UNIVERSIDAD DE MURCIA
Departamento de Ingeniería y Tecnología de Computadores

Simulación concurrente y elección dinámica de estrategias para la mejora de la entrada/salida de disco

Tesis propuesta para
la obtención del grado de

Doctor en Informática

Presentada por:
María Pilar González Férez

Dirigida por:
Antonio Cortés Roselló
Juan Piernas Cánovas

Murcia, febrero 2012

UNIVERSIDAD DE MURCIA
Departamento de Ingeniería y Tecnología de Computadores

Improvement of disk I/O by concurrent simulation and dynamic selection of strategies

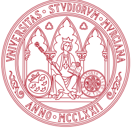
A dissertation submitted in fulfillment of
the requirements for the degree of

Doctor of Philosophy

By
María Pilar González Férez

Advised by
Toni Cortés
Juan Piernas

Murcia, February 2012



U N I V E R S I D A D
D E M U R C I A

DEPARTAMENTO DE INGENIERÍA Y TECNOLOGÍA DE COMPUTADORES

D. Juan Piernas Cánovas, Profesor Titular de Universidad del Área de Arquitectura y Tecnología de Computadores en el Departamento de Ingeniería y Tecnología de Computadores de la Universidad de Murcia

y

D. Antonio Cortés Roselló, Profesor Titular de Universidad del Área de Arquitectura y Tecnología de Computadores en el Departamento de Arquitectura de Computadores de la Universidad Politécnica de Cataluña

AUTORIZAN:

La presentación de la Tesis Doctoral titulada «*Simulación concurrente y elección dinámica de estrategias para la mejora de la entrada/salida de disco*», realizada por D.ª María Pilar González Férez, bajo su inmediata dirección y supervisión, y que presenta para la obtención del grado de Doctor por la Universidad de Murcia.

En Murcia, a 16 de enero de 2012.

Fdo: Dr. Juan Piernas Cánovas	Fdo: Dr. Antonio Cortés Roselló

UNIVERSIDAD DE
MURCIA

D. Antonio Fernando Gómez Skarmeta, Catedrático de Universidad del Área de Ingeniería Telemática y presidente de la Comisión Académica del Programa de Postgrado de Tecnologías de la Información y Telemática Avanzadas de la Universidad de Murcia, INFORMA:

Que la Tesis Doctoral titulada «*Simulación concurrente y elección dinámica de estrategias para la mejora de la entrada/salida de disco*», ha sido realizada por Dª María Pilar González Férez, bajo la inmediata dirección y supervisión de D. Juan Piernas Cánovas y de D. Antonio Cortés Roselló, y que la Comisión Académica ha dado su conformidad para que sea presentada ante la Comisión de Doctorado.

En Murcia, a 16 de enero de 2012.

Fdo: Dr. Antonio Fernando Gómez Skarmeta

A la querida memoria de mi tía, Pilar Férez Salmerón.

A Arturo y Pepita, mis padres, que me dieron mi *Tiempo*.

A *Sa*, mi hermana gemela.

Abstract

Over the years, advances in disk technology have been very important, and vast enhancements in disk drives have been made. However, memory and CPU performance has been improved at a much faster rate. Consequently, disk system performance usually dominates the overall throughput of a system, and limits the performance that many applications (specially, those that are data–intensive) can achieve.

In this thesis, we have focused our attention on improving the I/O performance, with the motivation that a better I/O performance will usually enhance the overall system performance. The three main contributions made to reach this aim are the following.

We firstly propose the *RAM Enhanced Disk Cache Project*, REDCAP, that enlarges the disk cache of a disk drive by using part of the main memory. To that end, we add a new level in the cache hierarchy, between the page and disk caches. We also introduce a prefetching technique that benefits from the read–ahead mechanism carried out by modern disk drives, and takes advantage of the read requests issued by applications. A mechanism to control the performance achieved by the new cache completes this approach. Thanks to REDCAP, we have been able to reduce the read I/O time by more than 80% in workloads with spatial locality, without downgrading the performance for other workloads.

As a way to achieve a self–monitoring and self–adapting I/O subsystem, we secondly propose an *in–kernel disk simulator* that is able to simulate both hard disk and solid state drives. Our disk simulator models a disk drive by using a dynamic table of I/O times, simulates a built-in disk cache too, and controls dependencies among requests and requests' thinking times to determine the request arrival order. The in–kernel simulator also has, as any other disk, an I/O scheduler that establishes the dispatch order among requests. This proposal allows us to compare, in real time, the behavior of different I/O system mechanisms, and to dynamically turn them on and off, depending on the expected performance. Our simulator has been successfully used in REDCAP to control its performance, and in our third contribution to implement a dynamic scheduling system. It also opens the door to new self–monitoring and self–adapting I/O mechanisms.

We finally present a *Dynamic and Automatic Disk Scheduling* framework (DADS). This mechanism selects the I/O scheduler that provides, for the current workload, the highest throughput. DADS compares any two Linux I/O schedulers by simultaneously running an instance of our in–kernel disk simulator for each scheduler. It then selects, at any time, the I/O scheduler providing the lowest service time for the same amount of requested data. By using DADS, the performance achieved is always close to that obtained by the best scheduler; system administrators are also exempted from selecting a suboptimal I/O scheduler which can provide a good performance for some workloads, but may downgrade the system throughput when the workloads change.

«Existe una cosa muy misteriosa, pero muy cotidiana. Todo el mundo participa de ella, todo el mundo la conoce, pero muy pocos se paran a pesar en ella. Casi todos se limitan a tomarla como viene, sin hacer preguntas. Esta cosa es el tiempo.

Hay calendarios y relojes para medirlo, pero eso significa poco, porque todos sabemos que, a veces, una hora puede parecernos una eternidad, y otra, en cambio, pasa en un instante; depende de lo que hagamos durante esa hora.

Porque el tiempo es vida. Y la vida reside en el corazón.»

Momo.
Michael Ende.

Agradecimientos

Si me pidieran definir el trabajo de todos estos años con una única palabra, elegiría *tiempo*. Tiempo de entrada/salida, de aplicación, de sistema, de sobrecarga, del disco virtual, de pruebas, de arranque... Todo tiene su *tiempo*, su instante, su momento. Ni un poco antes, ni un poco después: en su tiempo. Ahora es tiempo de agradecimientos.

A mi director de tesis Juan Piernas, por todo el trabajo compartido durante estos años, así como por su incansable esfuerzo, apoyo y dedicación, quien, además, se ha convertido en un entrañable amigo.

A mi director de tesis Toni Cortés, por el trabajo de todos estos años, así como por su paciencia y optimismo en los diferentes *tiempos* de este camino.

A mi buen amigo Antonio Flores, compañero de *tiempos* en parte de esta tesis. Antonio ha estado siempre cerca escuchando mis divagaciones sobre los resultados.

A todos mis compañeros del Departamento de Ingeniería y Tecnología de Computadores de la Universidad de Murcia. Para mi es un honor trabajar con ellos.

A mis amigos que han entendido mis *tiempos*, que han aceptado mis ausencias sin hacer reproches ni preguntas.

A Cáritas que me ayuda a poner los pies en el suelo y que me ha enseñado que mis *kernel panics* no son tan importantes, mis problemas no son problemas y mis *tiempos* pueden esperar.

A mis padres, Arturo y Pepita, a mis hermanos, Domingo, Rosario y Juan de la Cruz, a mi primo José y a mi tía María, por su amor, apoyo, compresión y paciencia. Mi vida no tendría sentido sin mi familia. Ellos son lo que soy: mi educación, mis valores, mi capacidad de esfuerzo y sacrificio... Sin vosotros no hubiese llegado hasta el «final». Os quiero.

A mi hermana, auténtica inspiración de todos estos *tiempos*.

To Keith, he has became a part of my family. Thanks for our meetings through Europe.

La mayoría de los tiempos de esta tesis han sido escritos con `vim`, medidos con `time`, buscados con `find` y `grep`, ordenados y organizados con `sort`, `tr`, `cut` y `uniq`, procesados con `awk`, etc. Sin estas órdenes, gran parte del análisis no se hubiese podido realizar.

De corazón: gracias a todos por vuestro *tiempo*.

María Pilar González Férez.
Febrero 2012.

Índice

Abstract xi

Agradecimientos xiii

Índice xv

Tabla de Contenidos xvii

Lista de Figuras xxi

Lista de Tablas xxvii

0. Resumen 1

1. Introducción 53

2. REDCAP: proyecto de caché de disco mejorada mediante RAM 61

3. Simulador de disco dentro del núcleo 111

4. DADS: selección automática y dinámica del planificador de E/S 179

5. Conclusiones y trabajo futuro 225

Bibliografía 229

Contents

Abstract xi

Agradecimientos xiii

Índice xv

Contents xvii

List of Figures xxi

List of Tables xxvii

0. Resumen de la tesis 1
- 0.1. Introducción 1
  - 0.1.1. Antecedentes 1
  - 0.1.2. Motivación 7
  - 0.1.3. Contribuciones de la tesis 8
- 0.2. Proyecto de caché de disco mejorada mediante RAM (REDCAP) 9
  - 0.2.1. Diseño e implementación de REDCAP 10
  - 0.2.2. Resultados Experimentales 15
  - 0.2.3. Conclusiones 21
- 0.3. Simulador de disco dentro del núcleo 23
  - 0.3.1. El disco virtual 23
  - 0.3.2. Caso de uso: REDCAP 27
  - 0.3.3. Resultados experimentales 28
  - 0.3.4. Conclusiones 36
- 0.4. Selección automática y dinámica del planificador de E/S 38
  - 0.4.1. Diseño de DADS 39
  - 0.4.2. Modificación del disco virtual 39
  - 0.4.3. Implementación de DADS 42
  - 0.4.4. Resultados experimentales 43
  - 0.4.5. Conclusiones 48
- 0.5. Conclusiones y trabajo futuro 49

1. Introduction 53
- 1.1. Background 53
  - 1.1.1. Hard Disk Drives 53
  - 1.1.2. Solid State Drives 56
  - 1.1.3. Hybrid Hard Disk Drives 58
- 1.2. Motivation 58

1.3. Thesis Contributions 59
1.4. Thesis Organization 60

2. REDCAP: The RAM Enhanced Disk Cache Project 61
2.1. Motivation 61
2.2. REDCAP overview 63
2.3. Design and Implementation 65
2.3.1. REDCAP cache 65
2.3.2. Prefetching technique 66
2.3.3. The activation–deactivation algorithm 70
2.4. Experiments and methodology 76
2.4.1. Hardware platform 76
2.4.2. Variations in the REDCAP cache configuration 76
2.4.3. Benchmarks 77
2.4.4. File systems 78
2.5. Results 79
2.5.1. Evaluation of the REDCAP segment size 80
2.5.2. Impact of the file system and cache size 87
2.6. Related Work 101
2.7. Conclusions 107

3. In–Kernel Disk Simulator 111
3.1. Motivation 111
3.2. In–Kernel Disk Simulator 113
3.2.1. Disk model 115
3.2.2. I/O schedulers for the virtual disk 119
3.2.3. Request management 120
3.2.4. Time control 124
3.2.5. Training the table 125
3.2.6. Avoiding the scheduler's queue congestion 126
3.2.7. Operation of the disk simulator 126
3.3. A use case: REDCAP 129
3.3.1. Active State 131
3.3.2. Inactive State 132
3.3.3. Management of the cache misses 133
3.4. Experiments and methodology 133
3.4.1. Hardware platform 133
3.4.2. Benchmarks 134
3.4.3. I/O schedulers of the experiments 135
3.5. Results 135
3.5.1. Accuracy of the virtual disk model 136
3.5.2. Performance of REDCAP with the virtual disk 141
3.6. Solid–State Drives 155
3.6.1. Viability of the virtual disk for SSDs 156
3.6.2. Experiments and methodology 157
3.6.3. Accuracy of the virtual disk model with SSDs 158

3.6.4. Performance of REDCAP on SSDs with the virtual disk 158
3.7. Related Work 167
3.8. Conclusions 175

4. DADS: Dynamic and Automatic Disk Scheduling framework 179
4.1. Motivation 179
4.2. DADS overview 181
4.3. Modification of the In–Kernel Virtual Disk 182
4.3.1. Disk model 184
4.3.2. I/O schedulers for the virtual disk 188
4.3.3. Thinking Time 189
4.3.4. Request management 190
4.3.5. Training the tables 190
4.3.6. Calculating the parameters of the simulated disk cache 190
4.3.7. Operation of the disk simulator 191
4.4. Implementation of DADS 193
4.4.1. Simulation process 194
4.4.2. Scheduler change 196
4.4.3. Performance control 196
4.5. Experiments and methodology 197
4.5.1. Hardware platform 197
4.5.2. Disk caches configurations 197
4.5.3. Benchmarks 199
4.5.4. I/O schedulers of the experiments 199
4.6. Results 200
4.6.1. Hard disk drives 201
4.6.2. SSD drives 215
4.7. Related Work 220
4.8. Conclusions 223

5. Conclusions and Future Directions 225

Bibliography 229

List of Figures

0.1. Comparativa del coste frente al tiempo de acceso de una memoria DRAM y un disco duro en los años 1980, 1985, 1990, 1995, 2000 y 2005. 3
0.2. Tendencias de la tecnología de los discos duros. 5
0.3. Jerarquía de cachés de REDCAP. 10
0.4. Manejo de REDCAP para una petición de lectura. 13
0.5. Mejora, en tiempo de aplicación, alcanzada por REDCAP sobre un núcleo de Linux sin modificar, al ejecutar el test *Lectura del núcleo de Linux*. 17
0.6. Mejora, en tiempo de aplicación, alcanzada por REDCAP sobre un núcleo de Linux sin modificar, al ejecutar el test *Lectura IOR*. 18
0.7. Mejora, en tiempo de aplicación, alcanzada por REDCAP sobre un núcleo de Linux sin modificar, al ejecutar el test *TAC*. 19
0.8. Mejora, en tiempo de aplicación, alcanzada por REDCAP sobre un núcleo de Linux sin modificar, al ejecutar el test *Lectura a saltos de 4 kB*. 20
0.9. Mejora, en tiempo de aplicación, alcanzada por REDCAP sobre un núcleo de Linux sin modificar, al ejecutar el test *Lectura a saltos de 512 kB*. 22
0.10. Las colas auxiliares, la cola del planificador y el modelo de tabla del disco virtual. 26
0.11. El disco virtual simulando un sistema «normal» cuando la caché de REDCAP está activa. 28
0.12. Diferencia, en porcentaje de tiempo de E/S, del disco virtual con respecto al disco real en el test *Todos los benchmarks seguidos*, usando el sistema de ficheros «nuevo» y el planificador CFQ, para 1, 8 y 32 procesos. 31
0.13. Celdas modificadas en la tabla de lectura una vez que el test *Todos los benchmarks seguidos* se ha ejecutado, para el sistema de ficheros «nuevo», el planificador CFQ y 1, 8 y 32 procesos. 33
0.14. Mejora, en tiempo de aplicación, alcanzada por REDCAP sobre un núcleo de Linux sin modificar, ejecutando los tests independientemente, en un sistema de ficheros «nuevo». 34
0.15. Mejora, en tiempo de aplicación, alcanzada por REDCAP sobre un núcleo de Linux sin modificar, ejecutando los tests independientemente, en un sistema de ficheros «envejecido». 34
0.16. Mejora, en tiempo de aplicación, alcanzada por REDCAP sobre un núcleo de Linux sin modificar, ejecutando los tests seguidos, en un sistema de ficheros «nuevo». 37
0.17. Mejora, en tiempo de aplicación, alcanzada por REDCAP sobre un núcleo de Linux sin modificar, ejecutando los tests seguidos, en un sistema de ficheros «envejecido». 37
0.18. Esquema general del diseño de DADS. 40
0.19. Resultados de DADS para las configuraciones *CFQ–Deadline* y *Deadline–CFQ*. 46

1.1. Cost versus access time for DRAM and hard disk in 1980, 1985, 1990, 1995, 2000, and 2005. . . . 55
1.2. Hard disk technology trends. . . . 57

2.1. REDCAP cache hierarchy. . . . 64
2.2. REDCAP management for a read request. . . . 67
2.3. An affected disk segment and its possible division into original and prefetched requests. . . . 68
2.4. REDCAP management for write requests. . . . 70
2.5. The activation–deactivation algorithm. . . . 71
2.6. Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel for the *Linux Kernel Read* benchmark depending on the RECAP segment size. . . . 81
2.7. Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel for the *IOR Read* benchmark depending on the RECAP segment size. . . . 82
2.8. Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel for the *TAC* benchmark depending on the RECAP segment size. 84
2.9. Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel for the *4 kB Strided Read* benchmark depending on the RECAP segment size. . . . 85
2.10. Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel for the *512 kB Strided Read* benchmark depending on the RECAP segment size. . . . 86
2.11. Improvement, in the I/O time, achieved by REDCAP over a vanilla Linux kernel for the *Kernel Compilation* benchmark depending on the RECAP segment size. . . . 87
2.12. Improvement, in the I/O time, achieved by REDCAP over a vanilla Linux kernel for the *TPCC* benchmark depending on the RECAP segment size. . . 88
2.13. Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel for the *Linux Kernel Read* benchmark depending on the file system used. . . . 90
2.14. Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel for the *Linux Kernel Read* benchmark depending on the file system used. . . . 92
2.15. Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel for the *TAC* benchmark depending on the file system used. . . . 93
2.16. Seeks of the read requests performed during the execution of the *TAC* benchmark run with 4 processes, the original kernel and the JFS file system. 96
2.17. Seeks of the read requests performed during the execution of the *TAC* benchmark run with 4 processes, the original kernel and the Ext3 file system. . . . 96
2.18. Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel for the *4 kB Strided Read* benchmark depending on the file system used. . . . 97

2.19. Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel for the *512 kB Strided Read* benchmark depending on the file system used. 99
2.20. Improvement, in the I/O time, achieved by REDCAP over a vanilla Linux kernel for the *Kernel Compilation* benchmark depending on the file system used. 100
2.21. Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel for the *TPCC* benchmark depending on the file system used. . . 102

3.1. Virtual disk routine. 114
3.2. Table disk model. 119
3.3. The auxiliary and I/O scheduler queues, and table model of the virtual disk. 123
3.4. Virtual disk routine and the auxiliary queues. 127
3.5. The virtual disk simulating a normal system when REDCAP is active. 131
3.6. The virtual disk simulating a REDCAP system when it is inactive. 132
3.7. Difference, in percentage of I/O time, of the virtual disk with respect to the real disk in the *All the benchmarks in a row* test, when the clean file system and CFQ are used, for 1, 8 and 32 processes. 137
3.8. Modified cells in the read table once all the benchmarks in a row are executed for the clean file system, CFQ, and 1, 8 and 32 processes. 140
3.9. Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed independently, on the clean file system and with the CFQ scheduler. 144
3.10. Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed independently, on the clean file system and with the AS scheduler. 144
3.11. Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel, when benchmarks are executed independently, on the aged file system and with the CFQ scheduler. 145
3.12. Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed independently, on the aged file system and with the AS scheduler. 145
3.13. Improvement, in the I/O time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed independently, on the clean file system and with the AS scheduler. 146
3.14. Improvement, in the I/O time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed independently, on the aged file system and with the AS scheduler. 146
3.15. Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed in a row, on the clean file system and with the CFQ scheduler. 150
3.16. Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed in a row, on the clean file system and with the AS scheduler. 150
3.17. Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed in a row, on the aged file system and with the CFQ scheduler. 151

3.18. Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed in a row, on the aged file system and with the AS scheduler. . . . 151
3.19. Improvement, in the I/O time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed in a row, on the clean file system and with the AS scheduler. . . . 152
3.20. Improvement, in the I/O time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed in a row, on the aged file system and with the AS scheduler. . . . 152
3.21. Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed at the same time. . . . 154
3.22. Difference, in percentage of I/O time, of the virtual disk with respect to the real disk in the *All the benchmarks in a row* test, for the SSD–160 disk, CFQ, and 1 (a), 8 (b) and 32 (c) processes. . . . 159
3.23. Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed independently, on the SSD–160 disk and with the CFQ scheduler. . . . 162
3.24. Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed independently, on the SSD–160 disk and with the Noop scheduler. . . . 162
3.25. Improvement, in the I/O time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed independently, on the SSD–160 disk and with the CFQ scheduler. . . . 163
3.26. Improvement, in the I/O time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed independently, on the SSD–160 disk and with the Noop scheduler. . . . 163
3.27. Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed independently, on the SSD–64 disk and with the CFQ scheduler. . . . 164
3.28. Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed independently, on the SSD–64 disk and with the Noop scheduler. . . . 164
3.29. Improvement, in the I/O time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed independently, on the SSD–64 disk and with the CFQ scheduler. . . . 165
3.30. Improvement, in the I/O time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed independently, on the SSD–64 disk and with the Noop scheduler. . . . 165
3.31. Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed in a row, on the SSD–160 disk and with the CFQ scheduler. . . . 168
3.32. Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed in a row, on the SSD–160 disk and with the Noop scheduler. . . . 168

3.33. Improvement, in the I/O time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed in a row, on the SSD–160 disk and with the CFQ scheduler. . . . 169
3.34. Improvement, in the I/O time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed in a row, on the SSD–160 disk and with the Noop scheduler. . . . 169
3.35. Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed in a row, on the SSD–64 disk and with the CFQ scheduler. . . . 170
3.36. Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed in a row, on the SSD–64 disk and with the Noop scheduler. . . . 170
3.37. Improvement, in the I/O time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed in a row, on the SSD–64 disk and with the CFQ scheduler. . . . 171
3.38. Improvement, in the I/O time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed in a row, on the SSD–64 disk and with the Noop scheduler. . . . 171
3.39. Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed at the same time for the SSD disks. 172

4.1. Improvements, in application time, achieved by the AS scheduler over the CFQ scheduler for the *Linux Kernel Read* and *IOR Read* benchmarks, the disks HD–250–32 and HD–320–16, and 1, 2, 4, 8, 16 and 32 processes. . . . 181
4.2. Overview of DADS. . . . 183
4.3. Overview of the disk simulator divided into RAS and the virtual disk. . . . 185
4.4. Configurations *AS–Deadline* and *Deadline–AS* for hard disk drives. . . . 202
4.5. Configurations *AS–Noop* and *Noop–AS* for hard disk drives. . . . 204
4.6. Configurations *CFQ–AS* and *AS–CFQ* for hard disk drives. . . . 206
4.7. Configurations *CFQ–Deadline* and *Deadline–CFQ* for hard disk drives. . . . 208
4.8. Configurations *CFQ–Noop* and *Noop–CFQ* for hard disk drives. . . . 210
4.9. Configurations *AS–Deadline* and *Deadline–AS* for SSD disks. . . . 216
4.10. Configurations *AS–Noop* and *Noop–AS* for SSD disks. . . . 217
4.11. Configurations *CFQ–Deadline* and *Deadline–CFQ* for SSD disks. . . . 218
4.12. Configurations *CFQ–Noop* and *Noop–CFQ* for SSD disks. . . . 219

List of Tables

0.1. Attributes of memory hierarchy components. 2
0.2. Precios y capacidades de varios dispositivos de disco. 4
0.3. Comparación de 3 discos duros de los años 2001, 2007 y 2011. 6

1.1. Attributes of memory hierarchy components. 54
1.2. Prices and capacities of disk drives. 56
1.3. Comparison among 3 hard disk drives from 2001, 2007 and 2011. 58

2.1. Specifications of the WD Caviar WD1200BB test disk. 76
2.2. Features of the file systems used. 79

3.1. Specifications of the test hard disks. 134
3.2. For the real disk and the four configurations of the virtual disk, average I/O time per request measured during the execution of the *All the benchmark in a row* test, for 1, 8 and 32 processes. 139
3.3. Average I/O time per request measured during the execution of the *All the benchmark in a row test* when the disk cache is off, for 1, 8 and 32 processes. 141
3.4. Average application time measured during the execution of the *All the benchmarks at the same time* test, for 1, 2, and 4 processes. 155
3.5. Specifications of the test SSD disks. 157
3.6. For the SSD–160 disk, average application time measured during the execution of the *512k–SR* test, for 1, 2, 4, 8, 16, and 32 processes. 166
3.7. For the SSD–160 disk, average I/O time measured during the execution of the *512k–SR* test, for 1, 2, 4, 8, 16, and 32 processes. 166

4.1. Main parameters of the four test hard disk drives. 198
4.2. Parameters of the simulated disk caches. 199
4.3. Average I/O time per request measured during the execution of the *8 kB Strided Read* test, for the CFQ scheduler with the original kernel, and the configurations *CFQ–Deadline* and *CFQ–Noop* of the DADS kernel, for 1, 2, 4, 8, 16, and 32 processes. 214

Chapter 0

Resumen de la tesis

0.1. Introducción

En la actualidad, los discos siguen siendo los dispositivos de almacenamiento secundario más ampliamente utilizados pese a que su tasa de transferencia normalmente determina, en gran medida, el rendimiento global del sistema. En esta primera sección discutiremos los problemas que presentan estos dispositivos y por qué, en nuestra opinión, los discos duros van a seguir dominando, al menos por un tiempo, los sistemas de almacenamiento, a pesar de las mejoras tan importantes y significativas que actualmente se están realizando en la tecnología de discos y de los nuevos dispositivos de almacenamiento que están surgiendo. Después describiremos cómo, desde nuestro punto de vista, el rendimiento de E/S de estos dispositivos se puede mejorar. Finalmente, haremos un resumen de las principales contribuciones de esta tesis.

0.1.1. Antecedentes

A lo largo de los años, la tecnología de los discos ha avanzado tremendamente y se han alcanzado mejoras muy relevantes. Sin embargo, el rendimiento de la memoria y la CPU ha mejorado a una velocidad mucho más rápida. Como consecuencia, el rendimiento del sistema de disco es un factor dominante en el rendimiento global del sistema, limitando el rendimiento de muchas aplicaciones, especialmente de las que realizan mucha entrada/salida (en adelante E/S). De hecho, el subsistema de E/S de disco normalmente se identifica como el mayor cuello de botella del rendimiento del sistema de muchos ordenadores.

Discos duros

Los discos duros[1] son en la actualidad el dispositivo de almacenamiento secundario más común, a pesar de su bajo rendimiento. El problema es que los discos duros son sistemas electro–mecánicos muy complejos[2], y sus operaciones mecánicas reducen considerablemente su velocidad comparada con la velocidad de otros componentes [1, 2, 3].

Una comparación de los componentes de la jerarquía de memoria nos permite ilustrar estas diferencias en rendimiento. La tabla 0.1 muestra los principales atributos de diferentes

[1] El término disco duro lo usamos para hacer referencia a los discos que usan tecnología de platos magnéticos.

[2] Nos gustaría hacer notar que en esta tesis no vamos a describir los componentes de un disco duro ni su funcionamiento. Para obtener una visión general sobre estos tópicos, remitimos al lector a Ruemmler y Wilkes [1], que dan un descripción detallada de cómo funciona un disco duro, y a Jacob *et al.* [2], que proporcionan una buena visión de los discos desde los principios de grabación física a su funcionamiento, e incluso su evolución en el tiempo.

Tabla 0.1: Attributes of memory hierarchy components. Source: "Modern Processor Design: Fundamentals of Superscalar Processors" [4].

				Cost ($) per	
Component	**Technology**	**Bandwith**	**Latency**	**Bit**	**Gigabyte**
Disk drive	Magnetic field	10+ MB/s	10 ms	$< 1 \times 10^{-9}$	< 1
SSD drive†	Flash memory	100+ MB/s	85 μs	$< 5 \times 10^{-9}$	< 5
Main memory	DRAM	2+ GB/s	50+ ns	$< 2 \times 10^{-7}$	< 200
On–chip L2 cache	SRAM	10+ GB/s	2+ ns	$< 1 \times 10^{-4}$	$<$ 100k
On–chip L1 cache	SRAM	50+ GB/s	300+ ps	$> 1 \times 10^{-4}$	$>$ 100k
Register File	Multiported SRAM	200+ GB/s	300+ ps	$> 1 \times 10^{-2}$ (?)	$>$ 10M (?)

†Destacar que hemos añadido a la tabla original la información sobre un disco SSD (disco de estado sólido, en inglés *Solid–State Drive*). El dispositivo seleccionado es un disco SSD Intel SSDSA2MH160G2C1, con un tamaño de 160 GB, del año 2009 y que hemos usado en nuestros experimentos.

componentes de memoria usando datos del año 2005, mientras que en la figura 0.1 se presenta una comparación de coste por GB y tiempo de acceso entre la memoria DRAM y los discos duros, con datos desde el año 1980 al 2005. Como podemos observar, los discos duros tienen un limitado ancho de banda efectivo y una latencia extremadamente larga. En ancho de banda hay una diferencia de dos órdenes de magnitud entre un disco magnético (10 MB/s) y la memoria RAM (2+ GB/s), mientras que, en tiempo de acceso, la diferencia es de cinco órdenes de magnitud (10 ms para los discos duros y 50+ ns para la memoria principal). Sin embargo, al mismo tiempo, los discos duros proporcionan el almacenamiento más efectivo en cuanto a coste y las mayores capacidades de todas las tecnologías comparadas. La diferencia en coste es de dos órdenes de magnitud (menos de 1$ por GB para el almacenamiento en disco frente a menos de 200$ por GB para la memoria).

Hoy en día, estas diferencias se siguen manteniendo casi idénticas. En la tabla 0.2 se resumen los principales atributos de varios discos modernos elegidos aleatoriamente, con diferentes tecnologías y que se encuentran actualmente en el mercado. Cuando comparamos los tres primeros discos duros [6, 7, 8] de esta tabla con, por ejemplo, un módulo de memoria DDR3 de 4GB [9], podemos concluir que:

- Un disco duro rápido tiene una tasa de transferencia sostenida máxima de 200 MB/s (que se adquiere cuando el disco transfiere datos secuencialmente), mientras que el ancho de banda de un módulo de memoria DDR3 es de 10,6 GB/s.
- La latencia de un disco para accesos aleatorios es aproximadamente de 10 ms, mientras que la latencia de la memoria RAM es de 6 ns.
- El coste por GB para discos duros es de menos de 0,10$, mientras que para el módulo de memoria es de unos 10$.

No obstante, como hemos dicho, se han alcanzado importantes mejoras en la tecnología de los discos duros. En la actualidad se siguen realizando propuestas bastante interesantes en esta tecnología, como la *grabación magnética perpendicular* [16] o la *grabación magnética por calor asistido* [17]. Por su parte, los fabricantes también están integrando cachés más grandes y controladoras más inteligentes a sus nuevos productos. Sin embargo, todas estas mejoras

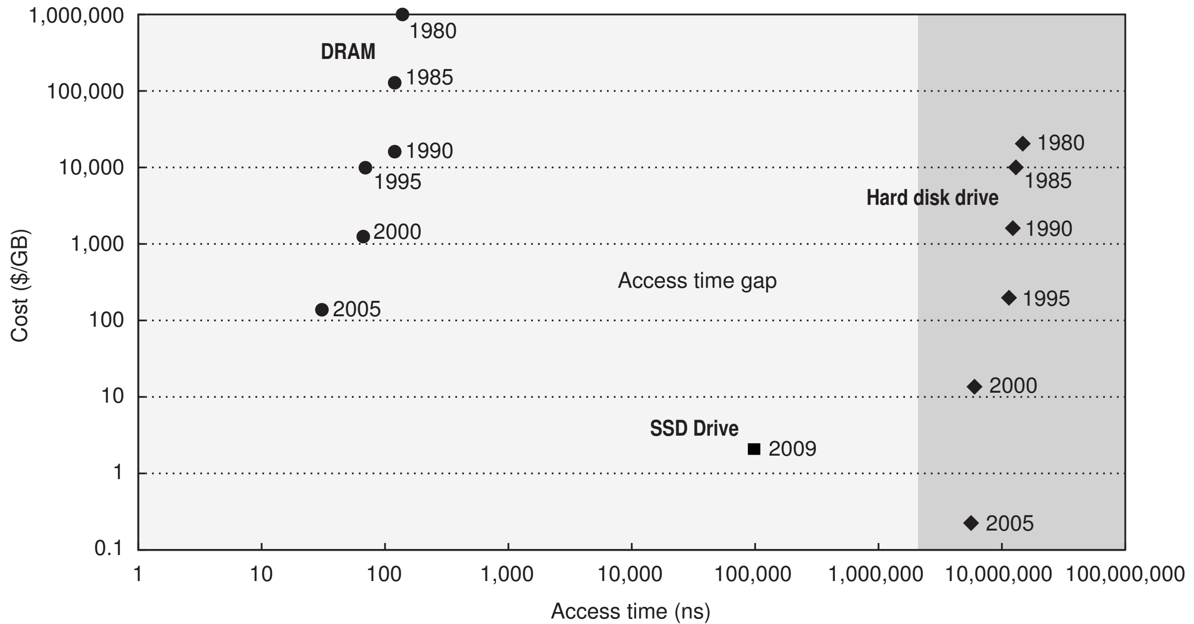


Figura 0.1: Comparativa del coste frente al tiempo de acceso de una memoria DRAM y un disco duro en los años 1980, 1985, 1990, 1995, 2000 y 2005. Destacar que entre los años 1990 y 2005 el coste por GB de los chips de DRAM no se mejoró mucho, mientras que el coste de los discos se mejoró de forma espectacular. Fuente: «Computer Architecture: A Quantitative Approach» [5]. Hemos añadido a la figura original datos sobre un disco SSD, que es el mismo de la tabla 0.1.

tienen un mayor impacto en la densidad del disco que en su rendimiento. Este desequilibrio se ilustra en la figura 0.2 y en la tabla 0.3. La figura 0.2 representa la tendencia de la tecnología de los discos duros a lo largo de los años. La tabla 0.3 compara las principales características de tres discos duros de 2001, 2007 y 2011, de los cuales los dos primeros los hemos usado en nuestros experimentos (ver las secciones 0.2.2 y 0.4.4)

Desde que en el año 1956 se introdujera el primer disco duro, la densidad de área de estos dispositivos se ha incrementado de forma espectacular, con un índice de crecimiento que varía desde el 25 % al 100 % [2, 18]. La historia evolutiva de esta mejora se resume en la figura 0.2(a). Puesto que la densidad de área determina la cantidad de datos que se pueden almacenar en un plato, también dicta la máxima capacidad de almacenamiento de un disco, y la misma increíble revolución se ha producido en la capacidad de los discos. Por ejemplo, Hitachi [19] introdujo el primer disco duro de 400 GB en el año 2004 [20], y Seagate [6] anunció el primero de 750 GB en 2006 [21], mientras que, en 2010, la capacidad máxima alcanzada fue 3 TB [22, 23], y 4 TB en 2011 [24, 25]. Es interesante destacar que un disco actual de 3 TB es un 400 % más grande que el disco de 120 GB usado en nuestros primeros experimentos (ver la tabla 0.3).

Lamentablemente, los valores de los tres principales componentes del rendimiento de un disco (tiempo de búsqueda, latencia rotacional y velocidad de transferencia) no se han mejorado de una forma tan significativa, e incluso algunos casi no se han modificado durante los últimos 15 años.

El tiempo de búsqueda ha decrecido con los años debido a que los componentes de los discos

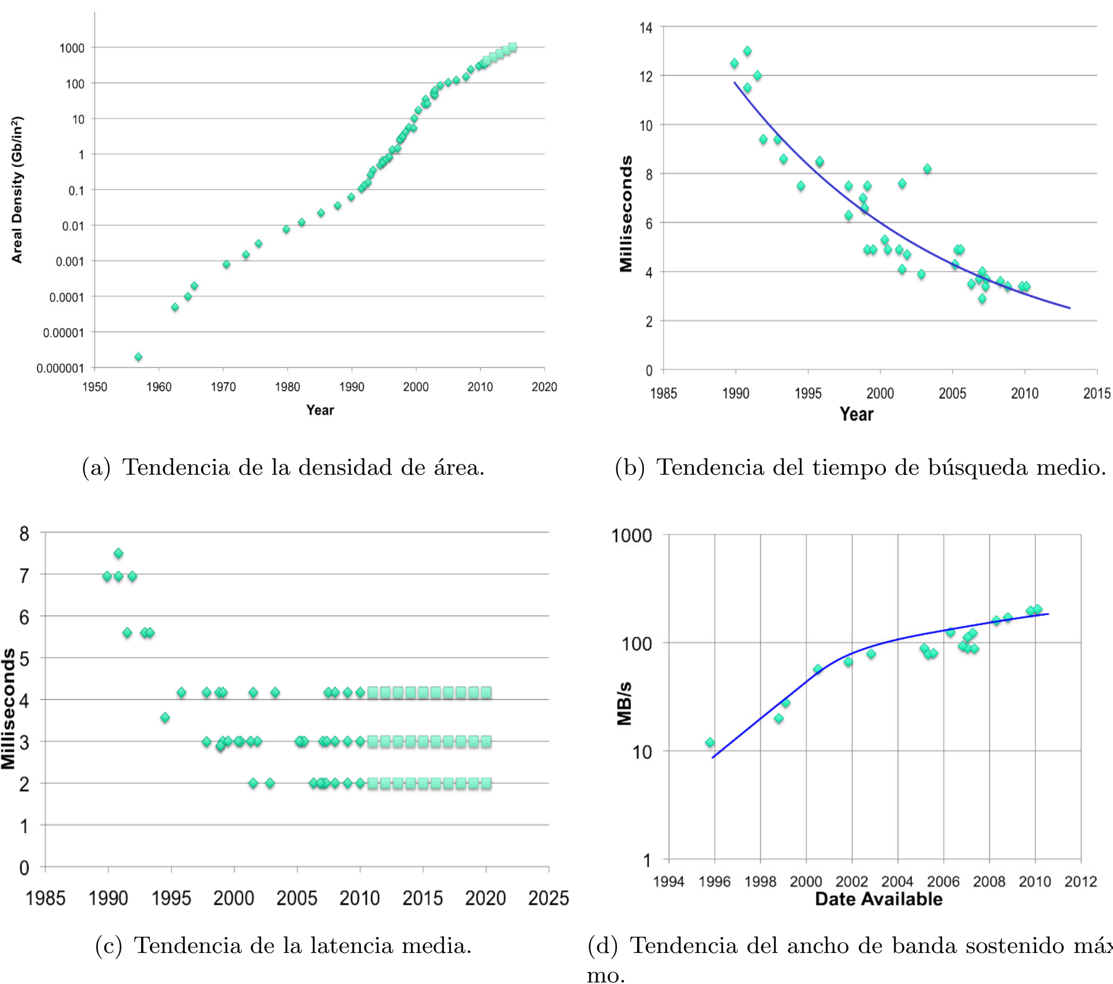
Tabla 0.2: Precios y capacidades de varios dispositivos de disco [6, 7, 8, 10, 11, 12, 13, 14, 15].

Tecno.	Modelo	Tamaño	Tamaño caché	Velocidad rotación	Tasa transferencia	Coste ($)
HDD	Seagate Barracuda XT® ST330005N1A1AS-RK	3 TB	64 MB	7200 RPM	138 MB/s (Max)	310
HDD	Hitachi Deskstar 7K3000 HDS723030ALA640	3 TB	64 MB	7200 RPM	207 MB/s (Max)	180
HDD	Western Digital Caviar Green WD30EZRX	3 TB	64 MB	IntelliPower	123 MB/s (Max)	250
HDD	Seagate Cheetah® NS 10k ST3600002SS	600 GB	16 MB	10000 RPM	82 MB/s (Min) 150 MB/s (Max)	530
HDD	Seagate Cheetah® NS 15k ST3600057SS	600 GB	16 MB	15000 RPM	122 MB/s (Min) 204 MB/s (Max)	670
SSD	Intel® SSD 710 Series	300 GB	–	–	270 MB/s (Read) 210 MB/s (Write)	1929
SSD	Samsung MZ–5PA256	256 GB	256 MB	–	250 MB/s (Read) 220 MB/s (Write)	370
SSD	Samsung MZ–7PC512D	512 GB	–	–	520 MB/s (Read) 400 MB/s (Write)	850
SSD	OCZSSD3-2VTX480G	480 GB	–	–	250 MB/s (Read) 215 MB/s (Write)	663
SSD	OCT1-25SAT3-512G	512 GB	512 MB	–	535 MB/s (Read) 400 MB/s (Write)	950
HDD + SSD	Seagate ® Momentus XT ST750LX003	750 GB 8 GB	32 MB	7200 RPM†	146,63 MB/s	240
HDD + SSD	OCZ RevoDrive Hybrid RVDHY-FH-1T	1 TB 100 GB	–	5400 RPM†	910 MB/s (Read) 810 MB/s (Write)	500

† Esta velocidad de rotación sólo es del disco duro interno.

son más pequeños y ligeros, especialmente por el menor diámetro de los platos. Sin embargo, debido a que este tiempo depende principalmente del movimiento de las cabezas del disco, las reducciones alcanzadas en la última década no son destacables (ver la figura 0.2(b) y la tabla 0.3). Además, es poco probable que se produzca un gran cambio en un futuro inmediato. En la actualidad, el tiempo de búsqueda medio típico de un disco duro para escritorio es todavía de unos 8 ms, el cual es bastante similar al de un disco del 2001, mientras que un disco duro para un servidor tiene un tiempo de búsqueda de unos 4 ms [2].

La latencia rotacional también ha mejorado, pero no de una manera significativa, porque es inversamente proporcional a la velocidad de rotación del disco. Es interesante hacer notar que, desde el año 2000, no se ha realizado ningún avance en este aspecto. La figura 0.2(c) muestra este hecho: durante la última década las latencias medias se han fijado a 2, 3 y 4,1 ms, que corresponden a 15000, 10000 y 7200 RPM, respectivamente. De hecho, los discos



(a) Tendencia de la densidad de área.

(b) Tendencia del tiempo de búsqueda medio.

(c) Tendencia de la latencia media.

(d) Tendencia del ancho de banda sostenido máximo.

Figura 0.2: Tendencias de la tecnología de los discos duros. Fuente: «GPFS Scans 10 Billion Files in 43 Minutes» [18].

duros para escritorio actuales normalmente giran a 7200 RPM, velocidad que se introdujo por primera vez para discos duros de servidores en 1994, y es la misma velocidad de rotación de los discos de la tabla 0.3. Los discos duros para servidores, por otro lado, llegan a girar a 15000 RPM, siendo 10000 RPM el valor más común [2].

La velocidad de transferencia media ha mejorado continuamente, y mucho más rápidamente que el tiempo de búsqueda y la latencia rotacional, debido al incremento de la densidad de grabación por bit. Esta evolución se presenta en la figura 0.2(d), que muestra cómo, desde 1996, el ancho de banda se ha incrementado de menos de 10 MB/s a más de 200 MB/s. De hecho, un disco actual ha multiplicado por dos la velocidad de transferencia sostenida máxima de un disco del año 2001 (ver la tabla 0.3).

Tabla 0.3: Comparación de 3 discos duros de los años 2001, 2007 y 2011 [8, 6, 7].

Modelo	Tamaño	Tamaño caché	Tiempo búsqueda	Velocidad rotación	Tasa transferencia	Año
WD Caviar WD1200BB	120 GB	2 MB	8,9 ms	7200 RPM	100 MB/s (Max)	2001
Seagate Barracuda ST3500630AS	500 GB	32 MB	$< 8{,}5$ ms	7200 RPM	100 MB/s (Max)	2007
Hitachi Deskstar 7K3000 HDS723030ALA640	3 TB	64 MB	8,2 ms	7200 RPM	207 MB/s (Max)	2011

Discos de estado sólido

Un competidor real y reciente de los discos duros son los discos de estado sólido (SSD, *Solid–State Drives*), que han evolucionado tremendamente en los últimos años. Los discos SSD superan en rendimiento a los discos *tradicionales* porque los nuevos dispositivos no presentan limitaciones mecánicas. Aún así su velocidad está todavía lejos del ancho de banda que alcanza la memoria RAM: el disco SSD más rápido de la tabla 0.2 transfiere a 535 MB/s, mientras que el módulo DRAM puede transferir a 10,6 GB/s.

Los SSDs también presentan varios inconvenientes. Un primer inconveniente es que todavía son muy caros comparados con los discos tradicionales. Aunque el precio de un disco SSD varía bastante dependiendo del fabricante (ver la tabla 0.2), todavía no pueden competir en precio con los discos duros. De hecho, los SSDs cuestan sobre 2$ por GB, mientras que los discos magnéticos cuestan menos de 0,10$ por GB. Un segundo inconveniente es que los discos duros proporcionan una capacidad muy superior a los SSDs. En la actualidad, es fácil encontrar un disco duro de 3 TB, mientras que los SSDs normalmente tienen una capacidad de 64 a 256 GB (no obstante existen en el mercado algunos discos SSD de 2 TB [26] bastante caros). Además, parece que esta diferencia en capacidad se va a mantener durante un tiempo, porque algunos autores señalan el gran reto que supone incrementar la capacidad de los discos SSD debido al tamaño de la litografía usada para hacer los chips [27, 28]. Finalmente, otro inconveniente es el tiempo de vida de los SSDs. El número del ciclo de escrituras de cualquier bloque de un dispositivo SSD es limitado, porque se tiene que borrar antes de poderse volver a escribir, y sólo es posible un número finito de borrados antes de que surjan errores de lectura/escritura [29, 30]. Los dispositivos tradicionales no presentan esta limitación.

Discos híbridos

Otra tecnología interesante es la propuesta para los discos duros híbridos (H–HDD, *Hybrid Hard Disk Drive*), que combina las tecnologías de almacenamiento magnético y de SSD. Los dispositivos H–HDD están compuestos por un dispositivo SSD y un disco duro. El primero mejora el rendimiento puesto que los datos más pequeños y accedidos con más frecuencia se almacenan en él, mientras que en el segundo se guardan los datos que se usan con menos frecuencia. Seagate Momentus XT (ver la tabla 0.2) es un ejemplo de un disco híbrido para portátiles. Este disco usa un algoritmo propio para monitorizar la actividad del dispositivo y determinar los datos que son los más óptimos para mantener en el disco SSD [14]. OCZ

RevoDrive Hybrid (ver la tabla 0.2) es otro ejemplo de H–HDD que también usa el dispositivo SSD como caché dedicada. De nuevo, un algoritmo controla dinámicamente el uso de ambos discos y decide qué datos se almacenan en cada uno [15].

Los dispositivos H–HDD poseen un par de características bastante interesantes. En primer lugar, los nuevos dispositivos son más baratos que los SSDs, ya que su precio está alrededor de 0,40$ por GB. En segundo lugar, dependiendo del patrón de acceso y de los datos almacenados en el SSD, los H–HDDs superan en rendimiento a los discos tradicionales, aunque, en el peor caso, obtienen el mismo rendimiento que su disco duro interno.

0.1.2. Motivación

A pesar del gran desequilibrio entre capacidad y rendimiento de los discos duros, todavía son los dispositivos dominantes de almacenamiento secundario en muchos ordenadores, y lo van a seguir siendo en un futuro inmediato [2]. Desde nuestro punto de vista, los discos SSD no van a reemplazar a los discos duros de una manera inmediata, aunque ambos van a coexistir durante un tiempo. Los nuevos dispositivos H–HDD también nos permiten pensar que los discos duros, o una versión mejorada de ellos, van a estar en uso por un tiempo. Además, muchas aplicaciones se caracterizan por sus enormes necesidades de almacenamiento, debido a la gran cantidad de datos que necesitan gestionar. En la actualidad, los discos duros son los únicos dispositivos que cumplen estas enormes necesidades de almacenamiento a un coste y a un rendimiento razonables.

Cuando el tiempo de E/S de una aplicación que realiza mucha E/S se reduce, el tiempo de ejecución también disminuye. En los subsistemas de E/S existen varios mecanismos, como las cachés, las técnicas de *prefetching* y los planificadores, que pueden reducir mucho el tiempo de E/S y, por consiguiente, mejorar el rendimiento obtenido. Por ejemplo, todos los discos duros tienen una caché interna integrada (llamada *caché de disco*) que mejora el rendimiento de E/S del dispositivo siempre que una petición de E/S se sirve directamente desde la caché, y no accediendo al disco.

Un problema de los mecanismos de E/S es que su rendimiento depende de varios aspectos (carga de trabajo, sistema de ficheros, modelo de disco, limitaciones técnicas, etc.). Por ejemplo, las cachés de disco no han sido tan efectivas como se esperaba debido a su pequeño tamaño comparado con la capacidad del disco. De hecho, un disco de 3 TB normalmente sólo tiene 64 MB de caché (ver la tabla 0.2). Puesto que varios autores apuntan que una caché más grande podría mejorar el rendimiento de E/S [31, 32], sería una buena idea proponer un mecanismo que agrandase de manera efectiva estas cachés de disco.

En otros casos, el comportamiento de un mecanismo de E/S puede llegar a degradar el rendimiento para una carga de trabajo concreta, o, incluso, puede haber otro mecanismo que, para las mismas condiciones, alcance un rendimiento mejor. Por lo tanto, para reducir el tiempo de E/S podría ser interesante activar/desactivar un mecanismo, o, dinámicamente, cambiar de un mecanismo a otro, dependiendo de las condiciones actuales. Esto nos permitiría conseguir un cierto objetivo, como incrementar el ancho de banda o reducir la latencia.

Teniendo en mente estas ideas, nuestra propuesta es un simulador de disco para, de una manera dinámica y *on–line*, simular el comportamiento de un disco real bajo diferentes condiciones. Basándonos en los resultados de esta simulación, el comportamiento de un mecanismo de E/S concreto podría ser modificado para mejorar su rendimiento. Nos gustaría hacer notar que los mecanismos diseñados para mejorar el rendimiento de E/S generalmente están imple-

mentados dentro del núcleo del sistema operativo. Por ello, pensamos que nuestro simulador de disco debe estar implementado también dentro del núcleo. De este modo, nuestro simulador podrá tener acceso a cualquier estructura del datos del núcleo o rutina relacionada con los mecanismos de E/S.

Una característica que queremos que tenga nuestro simulador es que no interfiera con el trabajo normal del sistema objetivo. Por suerte, hoy en día los ordenadores tienen una gran cantidad de memoria y una gran capacidad de cómputo, recursos que muchas veces están infrautilizados. Por ejemplo, los procesadores modernos normalmente tienen varios núcleos de procesamiento que, en ocasiones, no son totalmente explotados por las aplicaciones. En nuestra opinión, este entorno proporciona una plataforma interesante para mejorar el rendimiento de E/S. Esto es, nuestro simulador se puede implementar sin degradar el rendimiento del sistema, ni interferir en las peticiones de E/S realizadas por las aplicaciones, debido a los procesadores multinúcleo y a su gran potencia de cálculo.

0.1.3. Contribuciones de la tesis

El objetivo de esta tesis es la mejora del rendimiento de la E/S. Nuestra motivación es que el rendimiento global del sistema se beneficiará de forma significativa de tal mejora. A continuación se presentan las principales contribuciones de nuestra investigación.

Nuestra primera propuesta es agrandar la caché interna de los discos duros usando una parte de la memoria principal. El mecanismo, que nosotros llamamos **Proyecto de caché de disco mejorada mediante RAM** (*REDCAP*, *RAM Enhanced Disk Cache Project*) es una caché basada en RAM que imita el comportamiento de la caché de disco con el propósito de reducir el tiempo de E/S de las lecturas [33, 34]. Al hacer *prefetching* de los bloques adyacentes a los bloques de disco solicitados, nuestro enfoque se beneficia el mecanismo de lectura anticipada (*read–ahead*) realizado por los discos modernos, y aprovecha las peticiones de lectura lanzadas por las aplicaciones. Esta técnica implementa un mecanismo de control que activa o desactiva la nueva caché dependiendo del rendimiento de E/S conseguido. La sección 0.2 describe el diseño y la implementación de esta propuesta, incluyendo un análisis detallado de su comportamiento.

La segunda contribución de esta tesis es un marco para comparar, en tiempo real, diferentes elementos de E/S, y para activar/desactivar un mecanismo, o cambiar de un mecanismo o algoritmo a otro, dependiendo del rendimiento esperado. Presentamos el diseño y la implementación de un **simulador de disco dentro del núcleo de Linux** que es capaz de simular cualquier disco con una sobrecarga despreciable, y sin interferir con las peticiones de E/S regulares [35]. Nuestro simulador tiene en cuenta las posibles dependencias entre peticiones, el tiempo de *pensar* (tiempo que pasa entre que acaba una petición y llega la siguiente), los planificadores de E/S, etc. Esta propuesta nos permite alcanzar un comportamiento dinámico y mejorar el rendimiento global del sistema al simular diferentes mecanismos y algoritmos de E/S a la misma vez, y de forma dinámica activar y desactivar, o seleccionar entre diferentes opciones o políticas. En la sección 0.3 presentamos este simulador de disco, detallando el modelo de disco usado e incluyendo un estudio de su precisión. Como caso de uso describimos cómo el simulador puede ser usado para mejorar REDCAP.

Nuestra última contribución es un mecanismo que selecciona el planificador de E/S que obtiene el mayor rendimiento para la carga de trabajo actual. Es importante resaltar que no hay un planificador de E/S óptimo que consiga siempre el mejor rendimiento de E/S posible

para cualquier carga de trabajo [36]. Por esta razón, nosotros proponemos un **marco de planificación de disco automático y dinámico** (*DADS Dynamic and Automatic Disk Scheduling framework*), que compara dos planificadores de Linux cualesquiera, y selecciona el que está obteniendo el mejor rendimiento para la carga de trabajo actual [37]. Para realizar esta comparación, DADS usa nuestro simulador de disco. La sección 0.4 explica el diseño y la implementación de este sistema de planificación de E/S dinámico, proporcionando una evaluación de DADS.

0.2. Proyecto de caché de disco mejorada mediante RAM (REDCAP)

Hoy en día, la cache de disco, que todos los discos duros tienen integrada, actúa como *buffer* de aceleración y como caché de bloques [6, 38]. En el subsistema de E/S esta caché tiene un papel crucial, ya que reduce, en gran medida, el cuello de botella que supone, en muchos sistemas, el almacenamiento secundario debido a su bajo rendimiento si se compara con otros componentes, como la CPU o la memoria principal [3].

Los discos duros, normalmente, al terminar una operación de lectura, continúan leyendo, de forma secuencial, una serie de bloques consecutivos al último bloque de la petición de lectura, almacenando estos bloques en su caché. Con este comportamiento la caché de disco mejora los patrones de acceso secuencial. Cada vez que una nueva petición de lectura se puede servir desde la caché de disco, sin tener que leer los datos expresamente, se mejora el rendimiento de E/S. Por otro lado, la caché de disco también interviene en las operaciones de escritura, ya que, cuando llega una petición de escritura a disco, los datos se copian primero en ella y, posteriormente, se escriben en el disco. De esta forma, el disco no bloquea a los procesos esperando a que termine la petición de escritura. La controladora de disco escribirá los datos reduciendo el movimiento de las cabezas de disco, de forma que también mejora el rendimiento de E/S.

El tamaño de una caché tiene un efecto significativo en su rendimiento porque determina la tasa de aciertos [39], lo que hace que el tamaño sea uno de los aspectos más importante del diseño de una caché. Los diseñadores de sistemas generalmente consideran que una caché de disco debería suponer del 0,1 al 0,3 por ciento de la capacidad total del disco [31], y Hsu y Smith [32] sugieren que el rendimiento de E/S mejoraría con cachés de un tamaño de un 1 %, o incluso más, de la capacidad del disco. Aunque los fabricantes van incluyendo caché más grandes, su tamaño todavía está en el rango de 2 a 64 MB [40, 7], lo cual es bastante pequeño comparado con la capacidad total del disco. Por ejemplo, un disco de 1 TB normalmente tiene 32 MB de caché [6, 38], sólo el 0,003 % del tamaño total del disco. La tecnología actual de los discos duros indica que este desequilibrio entre los dos tamaños no va a cambiar a corto plazo [6, 38]. Las principales razones son, por un lado, las limitaciones de espacio en la controladora del disco y, por otro, la relación entre el tamaño de la caché y su coste (un incremento en el tamaño de las cachés conlleva un incremento considerable en su precio y, por tanto, en el precio de los discos duros). Estos reducidos tamaños han provocado que las caches de disco no sean tan efectivas como se esperaba.

Por tanto, si se espera que un incremento en la capacidad de las cachés de disco conlleve una mejora del rendimiento de los discos duros, consideramos interesante estudiar los beneficios potenciales de usar una pequeña parte de la memoria principal para ampliar la caché de disco.

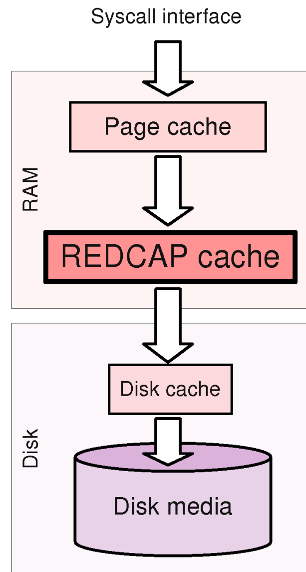


Figura 0.3: Jerarquía de cachés de REDCAP.

El propósito de este trabajo es presentar el *Proyecto de caché de disco mejorada mediante RAM*, llamado *REDCAP* (*RAM Enhanced Disk Cache Project*), que consiste en introducir una nueva caché de bloques de disco [33, 34].

0.2.1. Diseño e implementación de REDCAP

Las ideas esenciales de REDCAP son tres. La primera, usar la nueva caché como una extensión en la memoria principal de la caché de disco. La segunda, imitar el comportamiento de esta última leyendo de disco más datos de los inicialmente solicitados. La tercera y última, aprovechar, haciendo *prefetching* de bloques de disco, el mecanismo de lectura anticipada (*read–ahead*) que implementa la propia caché de disco.

La extensión de la caché de disco se realiza introduciendo una nueva caché de bloques de disco, llamada *caché de REDCAP*, entre la caché de páginas y la caché de disco, ampliando en un nuevo nivel la jerarquía de cachés. La figura 0.3 muestra la nueva jerarquía de cachés propuesta en esta tesis. Esta nueva caché se ubica en la memoria principal, usando para ello marcos de página reservados a tal efecto.

Para imitar el comportamiento de la caché de disco y aprovechar su mecanismo de *prefetching*, nuestra técnica lee por adelantado bloques de disco consecutivos que, *a posteriori*, podrían ser solicitados por futuras peticiones de lectura. Siempre que una petición de lectura pueda servirse desde la caché de REDCAP, sin tener que acceder a disco, se mejorará el rendimiento de E/S. Respecto a las peticiones de escritura, nuestra técnica no interviene en ellas, únicamente actualiza, si es necesario, los datos de su caché, y envía la solicitud a disco sin realizar ninguna modificación en la misma.

Además, nuestra primera propuesta implementa un *algoritmo de activación–desactivación* que estudia la mejora conseguida por la caché en todo momento. Si el algoritmo detecta que el rendimiento está empeorando porque nuestra caché no está siendo efectiva, el *prefetching* se

para y sólo los datos pedidos se leen del disco. En este estado, el algoritmo continúa analizando las peticiones recibidas y estudiando la mejora que la caché podría proporcionar, y cuando detecta que con la caché activa se mejoraría el rendimiento conseguido, el mecanismo de *prefetching* se inicia de nuevo.

Por tanto, nuestra primera propuesta consiste en tres partes:

1. una *caché* que amplía en RAM la caché del disco;
2. una *técnica de prefetching* para manejar la caché;
3. un *algoritmo de activación–desactivación* para controlar el rendimiento obtenido.

Caché de REDCAP

La caché de REDCAP es una caché de bloques de disco traídos por adelantado, en cada fallo de caché, mediante peticiones de lectura. La nueva caché tiene un tamaño fijo de C bloques de disco y, al igual que las cachés de disco, está dividida en N segmentos (segmentos de REDCAP), que son administrados de forma independiente y tienen un tamaño de S bloques (donde $C = N \times S$). REDCAP maneja el disco como una secuencia de bloques contiguos, a los que hace referencia mediante su número de bloque lógico, y también divide el disco en segmentos de bloques con el mismo tamaño que los segmentos de REDCAP. El primer segmento del disco empieza en el bloque de disco 0 y termina en el bloque de disco $S - 1$, el segundo en el bloque S y llega hasta el $2S - 1$, y así sucesivamente. Cada segmento de REDCAP tendrá un segmento de disco, es decir, S bloques de disco consecutivos. De esta forma, dada una petición, es sencillo saber si el segmento de disco afectado está o no en la caché.

Cuando todos los segmentos se están utilizando, la caché de REDCAP usa el algoritmo de reemplazo *menos recientemente usado* (LRU, *Least Recently Used*) para seleccionar el segmento a liberar. Este algoritmo es, sin duda, una política de reemplazo muy popular, es sencillo de implementar y, normalmente, genera buenos resultados.

REDCAP utiliza marcos de página de la memoria principal para almacenar los segmentos. Estos marcos de página se marcan como reservados para evitar que el sistema operativo los lleve a disco cuando necesite liberar espacio, haciendo *swapping* de ellos. Como el tamaño de los marcos de página y el de los bloques lógicos del sistema de ficheros es el mismo, 4 kB, cada bloque de disco estará en un marco de página.

Técnica de prefetching

La segunda parte es la *técnica de prefetching* que decide los datos que tienen que ser leídos desde el disco a la caché. La técnica implementada se puede considerar como una variante de la lectura anticipada que hace la caché de disco. REDCAP sólo realiza *prefetching* cuando recibe una operación de lectura y ésta produce un fallo de caché. Nunca se realiza ante un acierto de caché ni durante una operación de escritura. La técnica de *prefetching* es bastante simple pero efectiva, y es aplicable a cualquier sistema operativo, a cualquier sistema de ficheros e, incluso, a cualquier dispositivo de almacenamiento.

En un sistema normal, cuando una petición de E/S llega a la capa de bloques, se inserta directamente en la cola de peticiones del planificador de disco. Sin embargo, en un sistema REDCAP, la petición se gestiona por REDCAP.

Cuando REDCAP recibe una petición de E/S de lectura, primero calcula el número de segmentos de disco afectados y después los busca en su caché. Es importante destacar que una petición puede afectar a uno o más segmentos, dependiendo tanto del tamaño de la petición como del tamaño del segmento de REDCAP. En primer lugar vamos a describir la gestión que se realiza para una petición de lectura que afecta a un único segmento de disco, y después el caso de una petición que afecta a varios segmentos.

Si los bloques de disco solicitados están en uno de los segmentos de REDCAP, ocurre un acierto de caché. En este caso, los bloques se sirven directamente desde la caché, sin realizar ninguna lectura del disco. El proceso seguido es el siguiente: los datos se copian de los marcos de página de la caché a los marcos de página de la petición de E/S y, después, se finaliza la petición. Para la petición, el resultado final es el mismo pero más rápido, y sin acceder a disco; no sabe si los datos vienen de la caché de REDCAP o se han traído en ese momento del disco.

Sin embargo, si en la caché de REDCAP no están los datos pedidos ocurre un fallo de caché. En este caso, se leen de disco todos los bloques del correspondiente segmento de disco. Es interesante destacar que algunos de los bloques serán los solicitados por la petición de lectura original, mientras que los otros, leídos para completar el segmento, serán los bloques del *prefetching*. Por lo tanto, la cantidad de datos leídos por adelantado siempre depende tanto del tamaño de la petición como del tamaño del segmento de REDCAP. Con este procedimiento, nuestro método explota el principio de localidad espacial: si se hace referencia un bloque, también se hará referencia pronto a los bloques cercanos a él. Un esquema de esta gestión se presenta en la figura 0.4.

En el caso general, cuando una operación de lectura afecta a varios segmentos de disco, se divide en n peticiones parciales pequeñas, una por cada segmento de disco, que se gestionan de forma independiente. Todas las nuevas peticiones parciales se manejan de la misma manera, desde la primera a la última: calculando el segmento de disco afectado, buscándolo en la caché y copiando los datos, ante un acierto de caché, o leyendo los datos y los bloques para el *prefetching*, ante un fallo de caché. REDCAP controla cuándo acaban las peticiones parciales y, cuando todas ellas hayan terminado, finalizará la petición original.

Algoritmo de activación–desactivación

La última parte es el *algoritmo de activación–desactivación* que controla el rendimiento de REDCAP, y activa o desactiva su caché dependiendo de la mejora introducida.

En REDCAP se definen dos estados de trabajo, *activo* e *inactivo*. En el estado activo, la caché de REDCAP está funcionando y gestionando las peticiones de lectura que recibe, y el algoritmo está estudiando la posible mejora (o no) del tiempo de acceso que está suponiendo el uso de la caché. Cuando el algoritmo detecta que el tiempo de acceso está empeorando y el rendimiento es peor que si el sistema estuviese funcionando sin caché, cambia REDCAP al estado inactivo. En este estado inactivo, la caché no trabaja, las peticiones se envían directamente a disco y no se realiza *prefetching*. No obstante, el algoritmo estudia el posible beneficio de la caché, simulando que la caché de REDCAP está funcionando, y contabiliza los aciertos o fallos de caché de cada petición de lectura. Cuando el algoritmo detecta que la caché podría ser eficiente de nuevo, cambia REDCAP al estado activo, iniciando con ello el *prefetching* de la caché.

Supongamos inicialmente que la cache está activa. El algoritmo determina el rendimiento

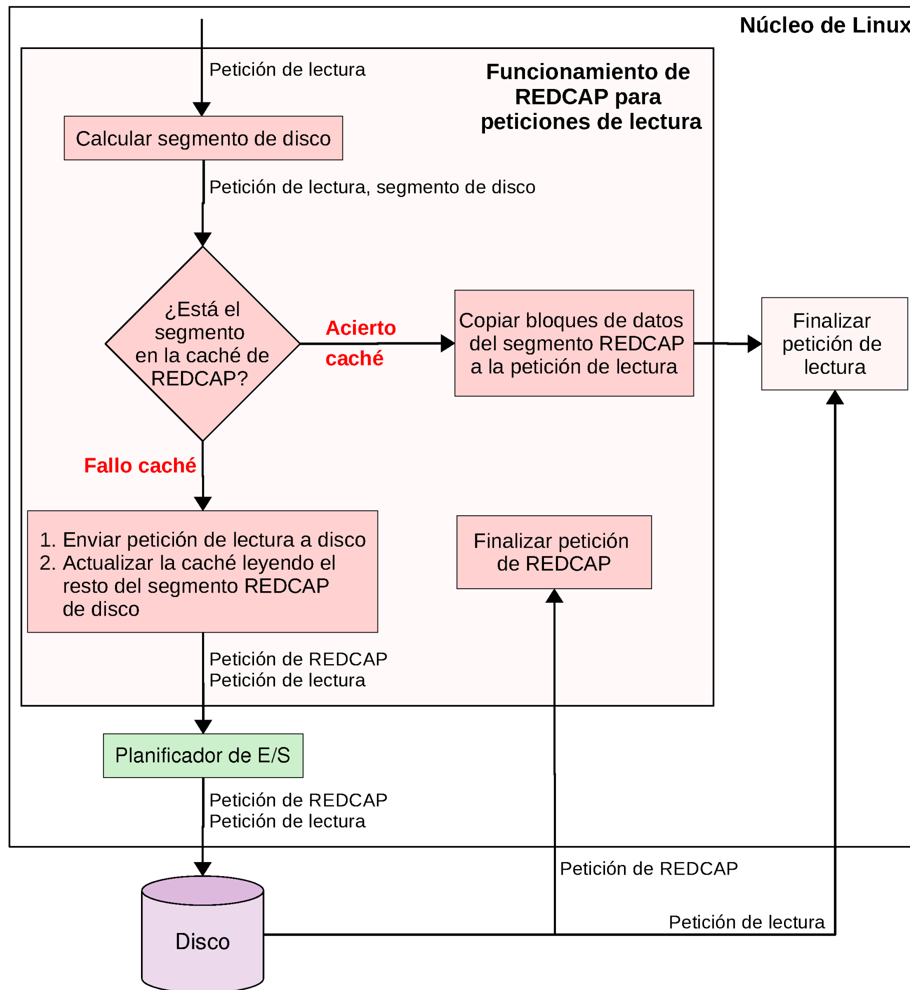


Figura 0.4: Manejo de REDCAP para una petición de lectura.

de la caché teniendo sólo en cuenta el tiempo de acceso a disco y el tiempo de copiar los datos de la caché de REDCAP a la petición recibida. Para realizar este análisis, REDCAP guarda los siguientes datos para cada petición de lectura que procesa:

- Para un acierto de caché (AC):
  - B_{AC} es el número de bloques de disco copiados desde la caché de REDCAP a la petición de lectura original cuando se produce un AC.
  - T_{AC} es el tiempo necesario para atender un acierto de caché, es decir, para copiar los B_{AC} bloques.
  - T_{Espera} es el tiempo que se está esperando a que los bloques solicitados lleguen de disco. Si una petición solicita bloques que ya se han pedido al disco por la caché de REDCAP, tiene que esperar que lleguen. En la mayoría de los casos este tiempo es cero, porque, normalmente, los datos ya han llegado.
- Para un fallo de caché (FC):

- B_{FC} es el número de bloques de disco de una petición original solicitados al disco porque no estaban en la caché.
- T_{Disco_FC} es el tiempo necesario para leer de disco B_{FC}.
- $B_{Prefetched}$ es el número de bloques de disco leídos por *prefetching* para la caché de REDCAP.
- $T_{Prefetched}$ es el tiempo necesario para leer de disco $B_{Prefetched}$.

Cuando la caché está activa, el tiempo de servir cada petición es:

$$T_{Activo} = T_{AC} + T_{Espera} + T_{Disco_FC} + T_{Prefetched}, \tag{0.1}$$

donde uno o más términos pueden ser cero, dependiendo de si la petición produce acierto y/o fallo de caché.

Por el contrario, si la caché hubiese estado desactivada, el tiempo de cada petición sería:

$$T_{Inactivo} = T_{Disco_AC} + T_{Disco_FC}, \tag{0.2}$$

donde T_{Disco_AC} es la estimación del tiempo que se necesitaría para leer de disco los B_{AC} bloques y se calcula como:

$$T_{Disco_AC} = \frac{T_{Disco_FC} * B_{AC}}{B_{Disco_FC}}. \tag{0.3}$$

El algoritmo de activación–desactivación establece que si el tiempo para servir una petición con REDCAP es menor o igual que el tiempo de servirla directamente de disco, REDCAP es efectivo y está mejorando el tiempo de acceso; en otro caso, REDCAP tiene que desactivarse. Usando las expresiones previas, el algoritmo dice que si la condición

$$T_{Activo} \leq T_{Inactivo} \tag{0.4}$$

se cumple, REDCAP debería estar activa, en otro caso debería desactivarse. Sustituyendo las expresiones (0.2) y (0.3) en la inecuación (0.4) y simplificando obtenemos

$$T_{AC} + T_{Espera} + T_{Prefetched} \leq T_{Disco_AC}. \tag{0.5}$$

Por lo tanto, si el tiempo que necesita la caché (es decir, el tiempo necesario para copiar los bloques de disco de la caché a la petición de lectura original más el tiempo de esperar que un segmento sea leído de disco más el tiempo necesario para el *prefetching* de los segmentos) es menor o igual que el tiempo estimado para leer los B_{AC} bloques de disco, la caché está siendo efectiva y no tiene que ser desactivada.

Cuando REDCAP está en el estado *inactivo*, para cada petición se calculan los mismos valores y se utiliza también la condición (0.5) para determinar si la caché tiene que ser activada de nuevo o no. La gran diferencia es que algunos de esos valores pueden ser calculados de forma exacta, mientras que otros tienen que ser estimados. Entre los primeros están B_{AC} y $B_{Prefetched}$, cuyos valores se conocen porque se está simulando el comportamiento de la caché. Y entre los segundos están T_{AC}, T_{Espera} y $T_{Prefetched}$, que se estiman usando los valores guardados la última vez que la caché estuvo activa.

Resumiendo, si REDCAP está en estado inactivo es porque la inecuación (0.5) no se cumple, es decir, el tiempo empleado por la caché es mayor que el necesario para leer B_{AC} bloques

directamente de disco. Cuando el algoritmo detecte que la condición (0.5) se vuelve a cumplir, la caché será efectiva de nuevo y REDCAP será activado.

Como *intervalo de comprobación* se define, en número de peticiones, cada cuánto tiempo se comprueba si REDCAP está mejorando el tiempo de acceso de un sistema normal o no. El valor de este intervalo se ha fijado en 100 peticiones.

0.2.2. Resultados Experimentales

Para analizar el rendimiento que obtiene esta propuesta hemos implementado REDCAP en un núcleo de Linux 2.6.14, al que llamaremos *núcleo REDCAP*. El estudio realizado evalúa el comportamiento de la caché de REDCAP y el impacto del tamaño del segmento (tamaño del *prefetching*) ejecutando varios *benchmarks*. Los resultados obtenidos con nuestro núcleo REDCAP los hemos comparado con los obtenidos con un núcleo de Linux 2.6.14 sin ninguna modificación, al que en esta sección denominaremos *núcleo original*.

Plataforma hardware

Los experimentos se han realizando en un sistema Pentium-III a 800 MHZ con 640 MB de memoria principal y dos discos duros. El primer disco duro es el disco de sistema, y es un disco Seagate ST-330621A [6] con el sistema operativo Fedora Core 4 que almacena las trazas para analizarlas. El segundo disco es en el que se realizan las pruebas y es un WD Caviar WD1200BB [38]. Este disco de pruebas tiene una capacidad de 120 GB y 2 MB de caché interna, y tiene una única partición, que usa todo el espacio del disco. El sistema de ficheros usado es Ext3 [41], cuyo tamaño de bloque lógico es 4 kB.

Configuración del sistema

Para realizar el estudio, el tamaño de la caché de REDCAP se ha establecido en 8 MB, cuatro veces mayor que el tamaño de la caché del disco. Es interesante destacar que estos 8 MB representan menos del 1,5 % de la memoria principal. Las pruebas se han realizado para cuatro configuraciones distintas del núcleo REDCAP:

- **256×32kB**. La caché se divide en 256 segmentos de 32 kB.
- **128×64kB**. En este caso, hay 128 segmentos de 64 kB cada uno.
- **64×128kB**. La caché se ha divido en 64 segmentos de 128 kB.
- **32×256kB**. El tamaño del segmento se ha establecido en 256 kB y hay un total de 32 segmentos.

El estado inicial de REDCAP es activo.

Las pruebas se han realizado usando el planificador de disco *Complete Fair Queuing* (CFQ) [42], elegido porque es el planificador de E/S por defecto en las últimas versiones «oficiales» del núcleo de Linux.

Benchmarks

A continuación se describen, en detalle, los *benchmarks* usados en la evaluación de REDCAP.

- *Lectura del núcleo de Linux (LKR, Linux Kernel Read)*. Este *benchmark* consiste en leer los ficheros fuente del núcleo de Linux 2.6.17 usando la siguiente instrucción:

```
find -type f -exec cat {} > /dev/null \;
```

  En el disco de pruebas hay un total de 32 copias de los ficheros del núcleo. Este test se ejecuta para 1, 2, 4, 8, 16 y 32 procesos, y cada uno de los procesos lee una de las 32 copias del núcleo de Linux.
- *Lectura IOR (IOR)*. El *benchmark* IOR se usa con bastante frecuencia para hacer pruebas en sistemas de ficheros paralelos usando un interfaz POSIX, MPIIO o HDF5 [43]. Sin embargo, nosotros aquí lo hemos usado para probar el comportamiento de REDCAP ante lecturas secuenciales en paralelo. La versión de IOR utilizada es la 2.9.1.
  La configuración establecida para este *benchmark* ha sido la API POSIX y un fichero por proceso. El tamaño de fichero es de 1 GB y el tamaño de las peticiones de E/S de 64 kB. IOR también se ha ejecutado para 1, 2, 4, . . . , y 32 procesos, cada uno leyendo su propio fichero.
- *TAC*. Este test lee hacia atrás ficheros usando la orden `tac` [44]. El *benchmark* se ha ejecutado para 1, 2, 4, . . . , y 32 procesos y, de nuevo, cada proceso lee su propio fichero. Es interesante destacar que se usan los mismos ficheros que en el *benchmark* IOR.
- *Lectura a saltos de 4 kB (4k–SR, 4 kB Strided Read)*. Este *benchmark* lee un fichero con un patrón de acceso a saltos. El test lee primero un bloque de 4 kB con un desplazamiento de 0, salta 4 kB, lee el siguiente bloque de 4 kB, salta otro bloque, etc. De nuevo, esta prueba se ejecuta para 1, 2, 4, . . . , y 32 procesos, y usa los mismos ficheros que IOR y TAC. Este test está escrito en C y usa las funciones POSIX `read` y `lseek` para leer y avanzar en el fichero.
- *Lectura a saltos de 512 kB (512k–SR, 512 kB Strided Read)*. Este test es similar al anterior, pero usa un patrón de acceso diferente. En este caso, cada proceso lee 4 kB, salta 512 kB, lee de nuevo 4 kB, salta 512 kB, etc. Cuando el proceso llega al final del fichero, realiza una nueva lectura con el mismo patrón de acceso, pero empezando en una posición distinta. En total la prueba realiza 4 series. La primera empieza en la posición 0, la segunda en la 4 kB, la tercera en la 8 kB y la cuarta, y última, en la 12 kB. Como en los casos anteriores, esta prueba se ejecuta para 1, 2, 4, . . . , y 32 procesos, y cada proceso lee su propio fichero. Los ficheros son los mismos que los usados en IOR, TAC y 4k–SR.

Resultados

Para cada una de las configuraciones del núcleo REDCAP y para el núcleo original, cada *benchmark* se ha ejecutado cinco veces. En las figuras los resultados que se muestran son la media de estas cinco ejecuciones. Además se incluyen, como barras de error, los intervalos de confianza para estas medias, con un nivel de confianza del 95 %. El equipo se reinicia después de cada prueba, por lo que todos los *benchmarks* se ejecutan con la caché de páginas y de REDCAP vacías.

Lectura del núcleo de Linux

En primer lugar, vamos a presentar la mejora conseguida en tiempo de aplicación por REDCAP con respecto al núcleo original en el *benchmark Lectura del núcleo de Linux*. Los

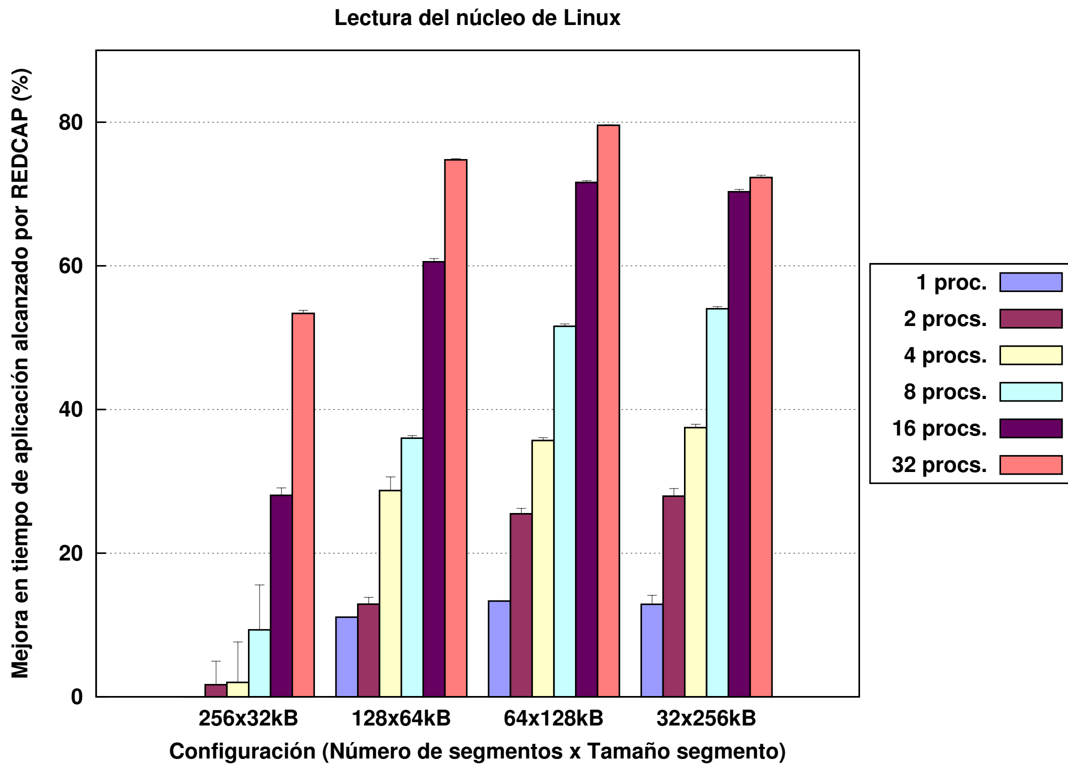


Figura 0.5: Mejora, en tiempo de aplicación, alcanzada por REDCAP sobre un núcleo de Linux sin modificar, al ejecutar el test *Lectura del núcleo de Linux*.

resultados obtenidos se presentan, en función de la configuración de REDCAP, en la figura 0.5.

Como se puede observar, los resultados de REDCAP son siempre mejores que los obtenidos con el núcleo original y, además, esta mejora se incrementa con el número de procesos. La configuración 64×128kB presenta el mejor rendimiento para 1, 16 y 32 procesos, mientras que para 2, 4 y 8 procesos el mejor rendimiento se consigue con la configuración 32×256kB. Aunque, la configuración 256×32kB presenta la mejora más pequeña, los resultados siguen siendo mejores que los del núcleo original para 8, 16 y 32 procesos, llegando a reducir el tiempo de aplicación en un 54 % para 32 procesos. Por otro lado, para 1, 2 y 4 procesos, REDCAP y el núcleo original obtienen, estadísticamente, los mismos resultados.

En este *benchmark*, el comportamiento de REDCAP depende, en gran medida, del tamaño de *prefetching*, es decir, el tamaño del segmento, y la mejoras se incrementan al aumentar el número de bloques leídos mediante el *prefetching*.

Una explicación por estos buenos resultados la podemos encontrar en la forma en que este *benchmark* lee el gran número de ficheros pequeños que tiene el núcleo de Linux. El proceso de lectura se realiza directorio a directorio. En un sistema de ficheros Ext3, los ficheros regulares de un mismo directorio se guardan todos juntos en el disco en el mismo grupo asignado al directorio, o en grupos cercanos si el correspondiente grupo está lleno [41]. El núcleo original no es capaz de detectar este patrón de accesos a bloques de disco cercanos (sólo hace *prefetching* de los bloques de disco de un fichero específico cuando detecta un acceso secuencial al mismo). Sin embargo, como ya mencionamos en la sección 0.2.1, REDCAP

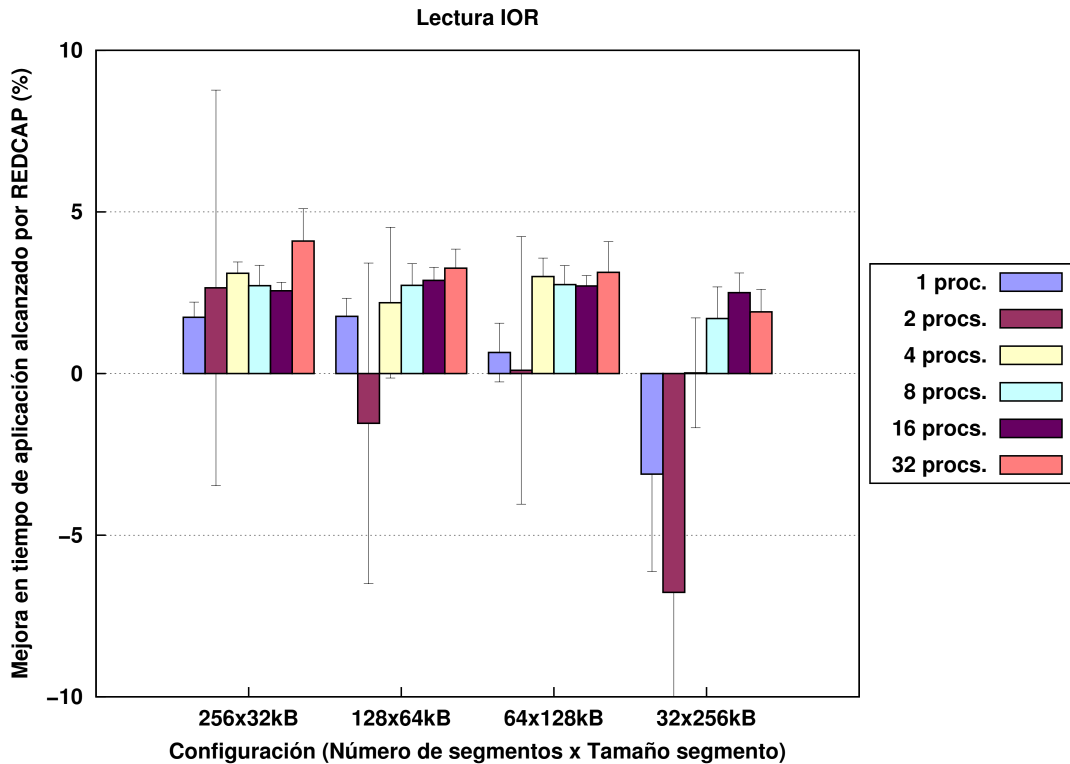


Figura 0.6: Mejora, en tiempo de aplicación, alcanzada por REDCAP sobre un núcleo de Linux sin modificar, al ejecutar el test *Lectura IOR*.

explota el principio de localidad espacial, y en este *benchmark* se aprovechan casi todos los bloques traídos a la caché, de hecho, la caché de REDCAP está casi todo el tiempo activa.

Lectura IOR

En la figura 0.6 se presenta la mejora en tiempo de aplicación conseguida por REDCAP con respecto al núcleo original, en función de la configuración de REDCAP, para el *benchmark Lectura IOR*.

Para esta prueba, el comportamiento de REDCAP es muy similar al del núcleo original, aunque en algunos casos llega a conseguir pequeñas mejoras. La excepción está en la configuración 32×256kB con 1 y 2 procesos, donde el rendimiento de REDCAP es ligeramente peor que el del núcleo original. En estos casos, el algoritmo activa y desactiva la caché varias veces, a pesar de que el mejor rendimiento se consigue cuando la caché está inactiva. El problema es que algunas veces, cuando REDCAP está inactivo, se reciben una serie de pequeñas peticiones, causadas por lecturas de *metadatos*, que hacen que la caché se active antes de que las siguientes peticiones le indiquen que la caché debería permanecer desactivada.

En este test se da un patrón de acceso secuencial y las técnicas de *prefetching*, tanto del núcleo original como de la caché de disco, están optimizadas para este tipo de patrón. La mayoría de las peticiones emitidas tienen también un tamaño de 128 kB, que es el tamaño máximo de una petición de disco permitida por el sistema de ficheros. Todo esto hace que

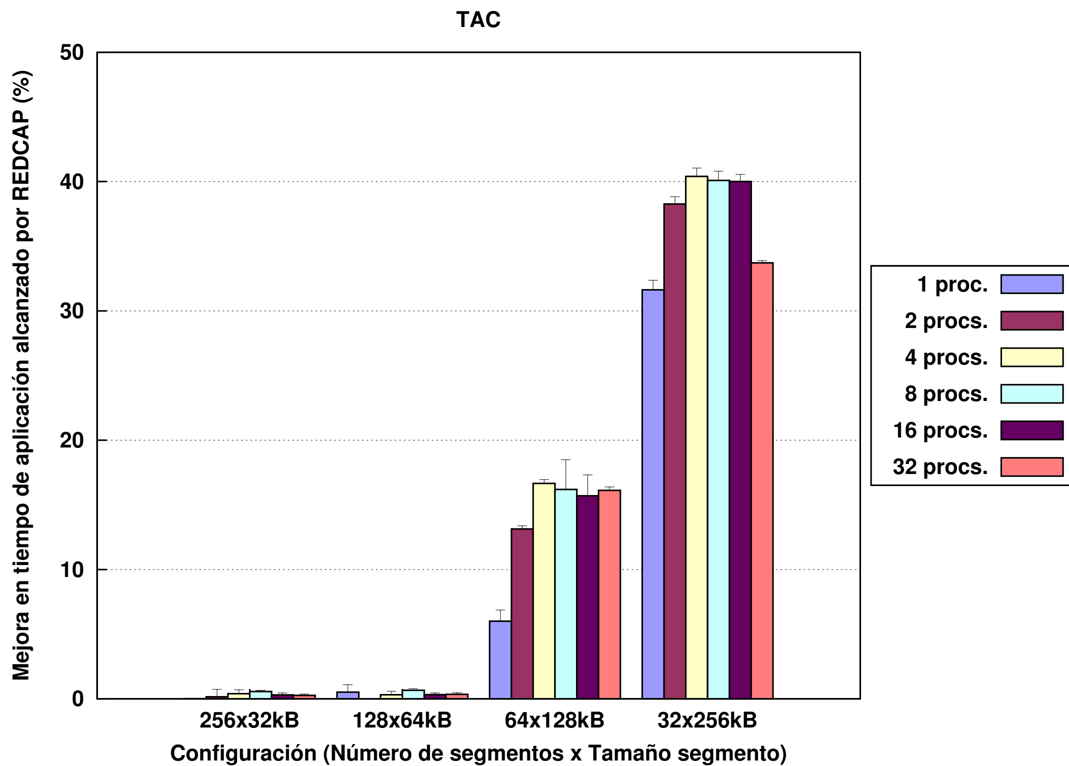


Figura 0.7: Mejora, en tiempo de aplicación, alcanzada por REDCAP sobre un núcleo de Linux sin modificar, al ejecutar el test *TAC*.

la contribución de nuestro método sea bastante pequeña, e incluso se está incluyendo un tiempo de copia en cada acierto de caché. El algoritmo de activación–desactivación detecta este comportamiento y la caché está desactivada casi durante toda la ejecución.

TAC

A continuación, en la figura 0.7, presentamos los resultados obtenidos para el tiempo de aplicación comparando los de REDCAP con los del núcleo original para el *benchmark TAC*.

Las configuraciones 64×128kB y 32×256kB, que mejoran de forma significativa el tiempo de aplicación, presentan un comportamiento cualitativamente similar pero cuantitativamente diferente, que depende, en gran medida, de la configuración usada, esto es, del tamaño del segmento.

El mejor rendimiento se obtiene para la configuración 32×256kB, con mejoras de hasta un 40 % para 4, 8 y 16 procesos. En este caso, la caché siempre está activa y el algoritmo nunca la desactiva.

Respecto a la configuración 64×128kB, REDCAP reduce el tiempo de aplicación con respecto el núcleo original en todos los casos. El mejor rendimiento se alcanza para 4 y 16 procesos, con una reducción del 16 %. La caché está casi todo el tiempo activa y sólo en raras ocasiones se desactiva.

El porqué de los resultados obtenidos con estas dos configuraciones se encuentra en que el

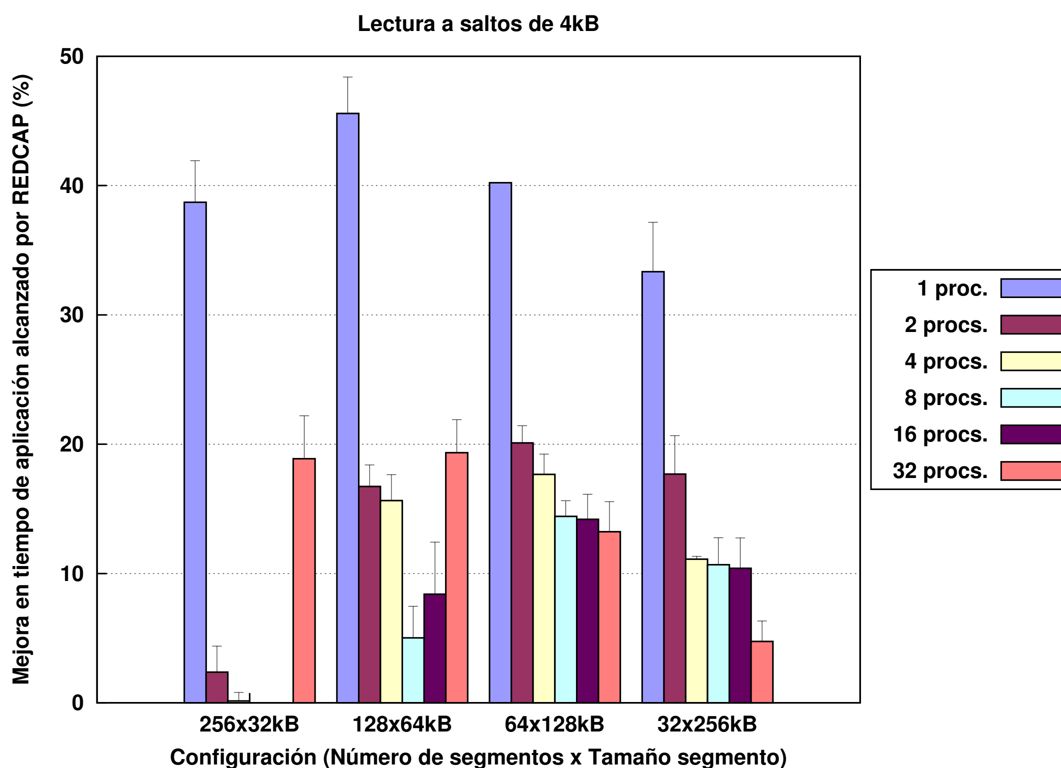


Figura 0.8: Mejora, en tiempo de aplicación, alcanzada por REDCAP sobre un núcleo de Linux sin modificar, al ejecutar el test *Lectura a saltos de 4 kB*.

sistema de ficheros Ext3 intenta reservar todos los bloques de un fichero regular juntos en disco, de tal manera que se optimice el acceso secuencial al mismo [41]. Como se dijo con anterioridad, esta localización de bloques beneficia la estrategia de *prefetching* que implementa REDCAP. Por otro lado, el núcleo original no es capaz de detectar el patrón de acceso hacia atrás, de forma que no realiza ningún *prefetching*.

Para las configuraciones 256×32kB y 128×64kB, los resultados conseguidos por REDCAP son muy similares a los obtenidos por el núcleo original. Esto es porque la orden `tac` lee hacia atrás un fichero con peticiones de 64 kB y el número de bloques leídos por adelantado en estas dos configuraciones es muy pequeño, debido al tamaño de segmento (32 kB y 64 kB, respectivamente), con lo que la contribución que proporcionan es muy pequeña o casi nula. El algoritmo detecta que la caché REDCAP no está siendo efectiva y la mantiene desactivada casi todo el tiempo.

Lectura a saltos de 4 kB

La figura 0.8 muestra los resultados conseguidos por REDCAP para el tiempo de aplicación comparados con los del núcleo original en el *benchmark Lectura a saltos de 4 kB*.

En esta prueba, el comportamiento de REDCAP depende de su configuración y del número de procesos. Para las configuraciones 128×64kB, 64×128kB y 32×256kB nuestra propuesta siempre consigue mejores resultados que el núcleo original. La mayor disminución en tiempo

de aplicación se consigue para 1 proceso con reducciones de un 45 %, 40 % y 33 % para cada configuración. A pesar de que las mejoras alcanzadas por las configuraciones 64×128kB y 32×256kB disminuyen al incrementar el número de procesos, para 32 procesos, el tiempo todavía se reduce en un 13 % y un 4,7 % respectivamente. Para la configuración 128×64kB, el peor rendimiento se consigue con 8 procesos, pero sigue siendo mejor que con el núcleo original, reduciendo el tiempo en un 5 %.

Por último, para la configuración 256×32kB con 1, 2 y 32 procesos los resultados obtenidos mejoran los del núcleo original, mientras que para 4, 8 y 16 procesos no se consigue ninguna mejora y los resultados son similares a los del núcleo original.

Para esta prueba, el algoritmo de activación–desactivación no es capaz de decidir el estado adecuado y la caché se activa/desactiva muchas veces. El problema, que aparece sólo con esta configuración, es que el disco duro detecta un patrón de acceso secuencial cuando la caché de REDCAP está activa, y activa su mecanismo de «*read–ahead*». Entonces, las peticiones originales que provocan fallo de caché aprovechan el *prefetching* que realiza el propio disco, y consumen muy poco tiempo, por lo que el algoritmo decide que el coste de la caché es mayor que leer directamente los datos del disco, y la desactiva. Sin embargo, cuando la caché está inactiva, como las peticiones no son secuenciales, el disco no activa su mecanismo de «*read–ahead*» y las peticiones originales tardan más tiempo en servirse. El algoritmo decide de nuevo que con la caché activa los resultados serían mejores, y la vuelve a activar. Este proceso de activar/desactivar la caché se repite sucesivamente. Por ello, los resultados no muestran un comportamiento sistemático como en los casos previos.

Por otro lado, es interesante destacar que el sistema operativo no detecta este patrón de acceso, y no implementa ninguna técnica para mejorar el rendimiento de esta carga de trabajo.

Lectura a saltos de 512 kB

En este apartado vamos a discutir los resultados de REDCAP en el *benchmark Lectura a saltos de 512 kB*. En la figura 0.9 se muestra el comportamiento de REDCAP en función de su configuración.

En esta prueba REDCAP presenta un comportamiento cuantitativamente similar en todas las configuraciones, aunque no mejora los resultados obtenidos con el núcleo original. Los peores resultados se dan con 1 y 2 procesos, con una pérdida del $-3,6\,\%$ (para todas las configuraciones) y del $-3,5\,\%$ (para la configuración 32×256kB) respectivamente. El problema es que en este test el tiempo de aplicación para 1 y 2 procesos es muy pequeño, y aunque la caché de REDCAP está casi todo el tiempo desactivada, inicialmente, cuando todavía está activa, pierde un tiempo que no es capaz de recuperar después. Para 4, 8, 16 y 32 procesos la pérdida se puede considerar despreciable y, fundamentalmente, se debe al tiempo que se necesita para simular el comportamiento de la caché cuando está inactiva.

Tal como sucede con el *benchmark Lectura a saltos de 4 kB*, el sistema operativo no implementa ninguna técnica de *prefetching* para este patrón de acceso.

0.2.3. Conclusiones

En esta sección hemos presentado REDCAP, que es una caché de disco basada en RAM capaz de reducir de forma significativa el tiempo de E/S de las peticiones de lectura de disco al usar una pequeña cantidad de memoria principal. Para algunas cargas de trabajo, REDCAP

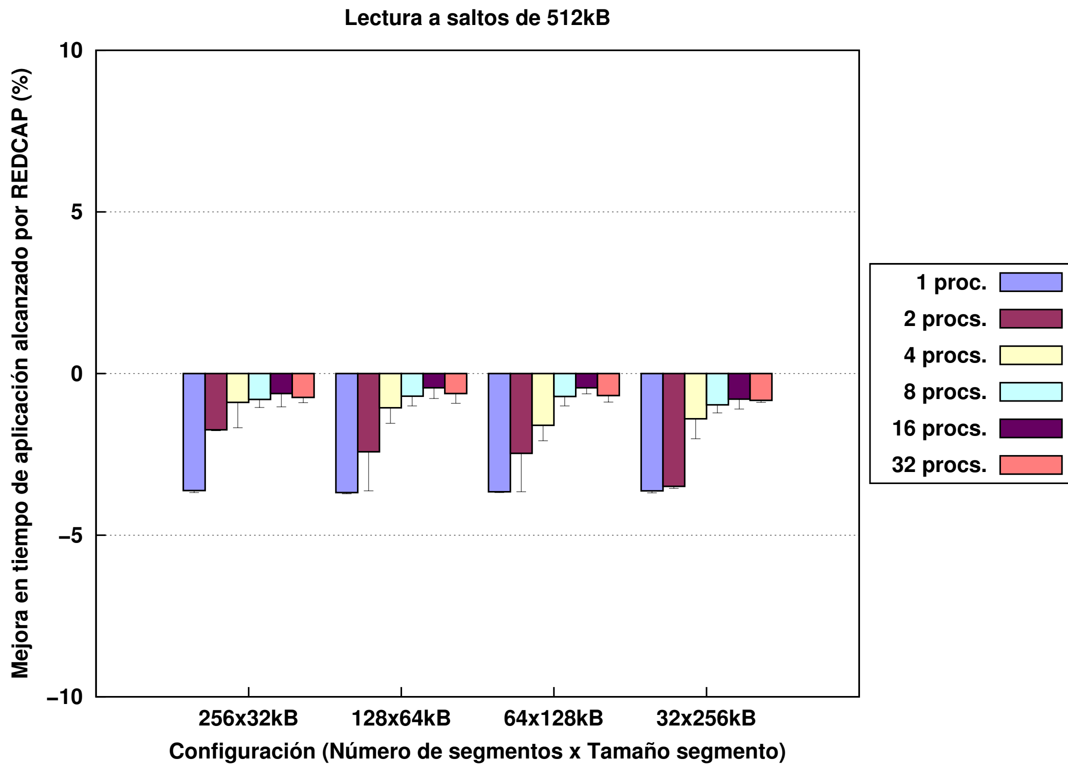


Figura 0.9: Mejora, en tiempo de aplicación, alcanzada por REDCAP sobre un núcleo de Linux sin modificar, al ejecutar el test *Lectura a saltos de 512 kB*.

mejora el rendimiento del núcleo original de Linux (sin ninguna modificación) en hasta un 80 %, mientras que para otras cargas de trabajo, para las que es difícil obtener una mejora, REDCAP iguala los resultados obtenidos por el núcleo original.

Analizando los resultados podemos afirmar que la mejora que proporciona REDCAP está estrechamente ligada con el tamaño de su segmento, esto es, el tamaño del *prefetching* que realiza. Así, las configuraciones 256×32kB y 128×64kB son las que menor beneficio aportan porque leen por adelantado menos datos. Sin embargo, las configuraciones 64×128kB y 32×256kB son las que alcanzan mejores resultados, siendo 64×128kB la que, en general, mejor comportamiento presenta.

Es importante resaltar que REDCAP tiene varias características que lo hacen único. En primer lugar, REDCAP es eficiente en tiempo de E/S, al ser capaz de aprovechar las peticiones de lectura de disco emitidas por una aplicación para realizar *prefetching* de los bloques adyacentes. En segundo lugar, es capaz de traducir cargas de trabajo con cientos de peticiones pequeñas en cargas de trabajo con peticiones grandes y secuenciales, que son óptimas para el funcionamiento del disco. En tercer lugar, REDCAP implementa un algoritmo de activación–desactivación que le permite ser dinámico, al activar o desactivar su caché dependiendo de la mejora conseguida durante su gestión. Las pruebas realizadas demuestran que el algoritmo, a pesar de ser bastante sencillo, es muy efectivo para un amplio rango de cargas de trabajo. Y cuarto y último, REDCAP es independiente del dispositivo utilizado. El algoritmo de activación–desactivación no tiene en cuenta ninguna característica física del disco, y sólo usa los

tiempos obtenidos experimentalmente para realizar sus cálculos y, en función de ellos, decidir un posible cambio de estado.

0.3. Simulador de disco dentro del núcleo

A lo largo de los años, la tecnología aplicada a los discos duros ha avanzado considerablemente introduciendo avances muy importantes. Sin embargo, el subsistema de E/S de disco aún sigue siendo el mayor cuello de botella del rendimiento de muchos ordenadores, debido a que las operaciones mecánicas de los discos duros reducen su velocidad de forma drástica y los hacen lentos en comparación con otros componentes como son la CPU o la memoria.

Son varios los mecanismos que juegan un papel importante en el rendimiento del subsistema de E/S: la caché de páginas o la caché de *buffers* del sistema operativo, la caché del disco duro, los mecanismos de *prefetching* del sistema operativo y del propio disco duro, los planificadores de E/S, etc. Sin embargo, aunque estos mecanismos pueden reducir significativamente el tiempo de E/S, no son óptimos, y la mejora que introducen depende de la carga de trabajo en un momento dado. Por otra parte, normalmente todos tienen un caso en el que pueden degradar el rendimiento de E/S.

Por lo tanto, sería una buena idea activar/desactivar un mecanismo, o cambiar de uno a otro, dependiendo de la carga de trabajo y del rendimiento esperado. Para conseguir este comportamiento dinámico, se necesita un medio de evaluar varias estrategias de E/S de forma simultánea.

Como un primer paso para implementar este sistema de simulación general, presentamos el diseño y la implementación de un simulador de disco dentro del núcleo, que cumple los requisitos previamente establecidos. Este simulador ha sido implementado en el núcleo de Linux, creando un *disco virtual* (también llamado DV, o VD de *Virtual Disk*) que captura el comportamiento de un disco duro [35].

0.3.1. El disco virtual

Para simular el comportamiento de un disco duro (en adelante llamado *disco real*, RD, o RD de *real disk*), hemos implementado, dentro del núcleo de Linux, un disco virtual que trabaja como un controlador de dispositivo y como un disco en sí. Al igual que un disco normal, el disco virtual tiene su propio planificador de E/S para ordenar las peticiones de E/S que recibe. Estas peticiones son una copia de las peticiones enviadas al disco real: antes de insertar una petición en el planificador del disco real, se crea una nueva petición «virtual» con los mismos parámetros básicos que la petición original. Las peticiones virtuales son insertadas en una cola auxiliar para su posterior procesamiento por el disco virtual.

El controlador del disco virtual crea un hilo del núcleo que, después de la fase de inicialización y creación, ejecuta una rutina que continuamente realiza las siguientes tareas:

1. Mueve las peticiones de la cola auxiliar a la cola del planificador de E/S.
2. Recoge la siguiente petición de la cola del planificador.
3. Obtiene, de un modelo basado en tablas del disco real, el tiempo de E/S estimado para atender la petición.
4. Duerme este tiempo estimado para simular que la operación de disco se está realizando.
5. Después de despertar, termina la petición y la borra de la cola del planificador.

Modelo de disco

Para modelar el dispositivo de almacenamiento hemos usado una tabla dinámica. Dada una petición, el modelo basado en tablas recibe una serie parámetros de entrada y devuelve el tiempo de E/S necesario para atender la petición. El modelo de disco propuesto es entrenado con los tiempos de E/S de las peticiones enviadas al disco real, sin tener en cuenta ninguna característica específica del disco. Por lo tanto, nuestro método es capaz de modelar el comportamiento de cualquier disco duro que pudiera usarse en la práctica.

Aunque el tiempo de una operación de E/S puede depender de varios factores [1, 45], nuestro modelo sólo usa, dada una petición, su tipo (lectura o escritura), su tamaño y la distancia entre peticiones [45] (distancia lógica en sectores de una petición con respecto a la petición anterior) para predecir su tiempo de E/S. La sección 0.3.3 muestra que estos tres parámetros, junto con el comportamiento dinámico de las tablas, son suficientes para modelar de forma precisa un disco duro.

Popovici *et al.* [46] afirman que la distancia lógica entre dos peticiones y el tipo de la petición son suficientes para predecir el tiempo de posicionamiento. Sin embargo, nosotros también consideramos el tamaño de la petición por dos razones. Primero, porque el tiempo de transferencia es proporcional a la longitud de la petición [1], especialmente para peticiones con una distancia pequeña con la petición anterior para las que este tiempo de transferencia es el factor dominante en el tiempo de E/S. Segundo, nosotros también tenemos en cuenta la caché del disco duro y, para una petición que es acierto de caché, su tiempo de servicio depende del tamaño de la misma. La sección 0.3.3 muestra que el tamaño de la petición es fundamental para nuestro modelo de disco.

Puesto que las operaciones de lectura y escritura tienen diferentes tiempos de E/S [1, 45], nuestro modelo maneja dos tablas diferentes: una para peticiones de lectura y otra para peticiones de escritura. En las tablas, las filas representan el tamaño de la petición. Cada tabla tiene treinta y dos columnas, representando tamaños desde 1 bloque (4 kB) hasta 32 bloques (128 kB). Estos tamaños se corresponden con los tamaños de petición de disco mínimo y máximo permitidos por el sistema de ficheros, respectivamente.

Las columnas representan las distancias entre peticiones. No obstante, debido al gran número de posibles distancias en los discos modernos por su gran capacidad, hemos asignado a las columnas rangos de distancias entre peticiones. La primera columna representa una distancia de 0 kB, representando principalmente aciertos de caché. Desde la columna 2ª a la 19ª, la tabla guarda valores para pequeñas distancias entre peticiones para simular, con una mayor precisión, el comportamiento del disco y el efecto de su *prefetching* y su caché. La columna n (con n de 2 a 19) guarda distancias mayores e iguales que $4 \cdot 2^{n-2}$ kB y menores que $4 \cdot 2^{n-1}$ kB. Las distancias entre peticiones más grandes están representadas por el resto de las columnas. Por ejemplo, la columna número 20 guarda distancias desde 1 GB a menos de 2 GB, la 21 desde 2 GB a menos de 3 GB, etc. Es interesante destacar que la capacidad del disco es la que determina el número de columnas.

Para resumir, dada una petición, su tipo selecciona la tabla, su tamaño la fila y su distancia con la petición anterior la columna. El valor de la celda es el tiempo de E/S necesario para servir esa petición.

Entrenamiento de la tabla

Las tablas pueden ser inicializadas *on–line* u *off-line*. Para la inicialización *off–line*, hemos implementado un programa de entrenamiento. Este programa tiene que ser ejecutado sólo una vez para cada disco que se vaya a usar y, debido a las operaciones de escritura, antes de utilizar el disco por primera vez. El proceso de entrenamiento normalmente es «rápido». En nuestro sistema, para un disco de 400 GB el proceso tardó 80 minutos.

Aunque el programa se ejecuta en el espacio de usuario, las tablas se construyen dentro del núcleo de Linux, que, para cada petición que sirve, guarda su tamaño, su tipo, su distancia entre peticiones y su tiempo de E/S. El valor de cada celda es la media del tiempo de E/S de las muestras tomadas para la misma. Una vez calculadas, las tablas son obtenidas del núcleo mediante el sistema de ficheros virtual `/proc`, el cual es también usado posteriormente para proporcionar las tablas al disco virtual tras arrancar el sistema.

El programa de entrenamiento produce un patrón de acceso aleatorio que no aparece en muchas cargas de trabajo. Por lo tanto, para modelar el comportamiento del disco de una manera más precisa, *adaptar el modelo a la carga de trabajo actual* y *captar los efectos de la caché del disco*, se ha implementado un método dinámico. Durante el funcionamiento normal del sistema, las celdas son actualizadas con los tiempos de E/S de las peticiones servidas por el disco real. Cada celda almacena los *últimos* X tiempos de E/S medidos, y su valor es calculado haciendo la media de estos tiempos. Las tablas olvidan, de esta manera, los tiempos más viejos que corresponden con cargas de trabajo pasadas. La sección 0.3.3 analiza la sensibilidad del modelo de disco al número de valores usados para hacer la media por celda, y muestra que, gracias al enfoque dinámico, el comportamiento del disco virtual se ajusta en gran medida al comportamiento del disco real.

Con la configuración *on–line* no hay sobrecarga de entrenamiento, las celdas son simplemente inicializadas a cero y, a continuación, se van actualizando dinámicamente según las peticiones de disco son atendidas. Para una celda que está sin inicializar, el modelo devuelve la media de la correspondiente columna como tiempo de E/S, si este valor no es cero; en otro caso, devuelve la media de la columna más cercana con celdas inicializadas.

Nos gustaría hacer notar que nuestro modelo no considera de forma explícita varias características de los discos modernos, como la grabación por zonas, el desfase de pistas/cilindros, o el *remapeo* de sectores defectuosos. Sin embargo, el impacto de estas características se tiene en cuenta de forma indirecta a través de los tiempos de E/S obtenidos del disco real durante la actualización dinámica de las tablas.

Manejo de peticiones

El disco virtual tiene que servir las peticiones de un proceso en el mismo orden en el que el proceso las ha lanzado. Puesto que el disco virtual puede servir las peticiones más despacio que el real, las dependencias entre peticiones tienen que ser controladas para permitir que nuestro simulador sirva las peticiones en el orden correcto.

Una petición de lectura normalmente es asíncrona: la correspondiente aplicación se bloquea y no puede lanzar una nueva petición hasta que la petición actual no se sirva. Además, también hay dependencias entre peticiones de procesos relacionados. Un ejemplo es un proceso padre que ejecuta el siguiente trozo de código:

1. Lanza una operación síncrona de lectura (PLP_1).

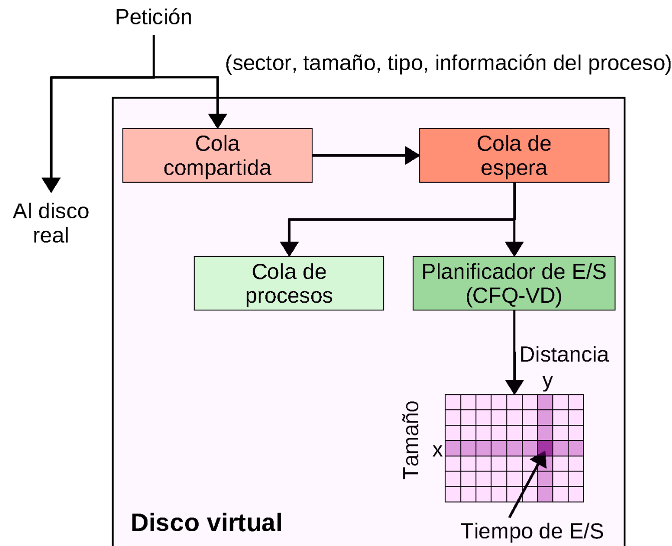


Figura 0.10: Las colas auxiliares, la cola del planificador y el modelo de tabla del disco virtual.

2. Crea un proceso hijo y espera a que este proceso hijo termine.
3. Lanza una segunda operación síncrona de lectura (PLP_2).

Si el proceso hijo lanza una petición de lectura síncrona (PLH), se establecen las siguientes dependencias: (a) PLH tiene que ser insertada sólo cuando PLP_1 haya terminado, y (b) PLP_2 no puede ser lanzada hasta que PLH no haya sido terminada.

Sin embargo, también hay peticiones de lectura asíncronas. Por ejemplo, el núcleo de Linux soporta *prefetching* de ficheros, y transforma pequeñas peticiones de lectura secuenciales en peticiones de *read–ahead* asíncronas más grandes que las originales. Por lo tanto, tenemos que distinguir entre operaciones *síncronas* y *read–ahead*.

Para mantener estas dependencias, además de la cola del planificador, se usan tres colas adicionales (ver figura 0.10). La primera es la *cola compartida*, que comunica el simulador con el sistema operativo. Cuando se lanza una nueva petición, justo antes de insertarla en el planificador del disco real, el sistema crea una *petición virtual* que inserta en esta cola compartida.

La segunda es la *cola de espera*, que guarda la peticiones que no pueden ser insertadas en el planificador porque tienen dependencias por resolver, y también mantiene el orden de llegada de las mismas. El disco virtual, después de servir una petición y antes de servir la siguiente, mueve las peticiones de la cola compartida a la de espera.

Una petición en la cola de espera se mueve a la del planificador si y sólo si ha resuelto todas sus dependencias. Para controlar las dependencias, hemos implementado la siguiente heurística:

- Las operaciones de escritura se insertan inmediatamente. Estas operaciones son normalmente asíncronas y no tienen dependencias.
- Una operación de lectura *síncrona* se insertará en la cola del planificador si no hay otra petición de lectura *síncrona* del mismo proceso en la cola del planificador o antes que ella en la cola de espera. Destacar que una petición de lectura *síncrona* se puede

insertar en la cola del planificador cuando ya hay peticiones de *read–ahead* del mismo proceso en esta cola.

- Una petición de *read–ahead* se inserta en la cola del planificador si no hay ninguna petición de lectura *síncrona* del mismo proceso delante de ella en la cola de espera.
- La primera petición de un nuevo proceso se inserta en la cola del planificador cuando la última petición del proceso padre, emitida antes de la creación del proceso hijo, ha sido servida. Por otro lado, si el proceso padre espera la finalización del hijo, ninguna nueva petición del proceso padre se insertará en el planificador mientras el hijo exista.

Aunque nuestra heurística no controla todas las dependencias, la mayoría de ellas sí se capturan. De hecho, el disco virtual procesa las peticiones en el mismo orden que el disco real lo hubiese hecho.

La última cola, la cola de procesos, simplemente controla qué procesos tienen peticiones pendientes en el planificador. El problema es que la cola del planificador no se puede usar para esta tarea porque Linux la gestiona como una caja negra que no se puede explorar para saber qué peticiones quedan pendientes.

Finalmente, es importante destacar que aunque una petición virtual pasa por diferentes colas, su tiempo de servicio se calcula de la misma manera que en el disco real: el tiempo que ha pasado desde que se insertó en la cola del planificador hasta que terminó.

Planificador de E/S

Hemos adaptado los planificadores de E/S *Complete Fair Queuing* (CFQ) [42] y *Anticipatory* (AS) [47] para trabajar con la cola de peticiones del disco virtual. El problema es que estos dos planificadores usan información sobre el proceso que insertó una petición para ordenar la cola y realizar la selección de peticiones a enviar a disco. Sin embargo, en el disco virtual, la peticiones son insertadas por el propio disco virtual y, de hecho, pertenecen al hilo del núcleo. Por lo tanto, los dos planificadores han sido modificados para guardar y usar la información del proceso de una manera diferente. Los nuevos planificadores, *CFQ–VD* y *AS–VD*, se comportan igual que los planificadores originales.

Los planificadores de E/S *Noop* y *Deadline* [42], por el contrario, no usan información de los procesos para realizar la planificación, por lo tanto, se pueden usar en el simulador sin ninguna modificación.

Otro aspecto interesante es que es posible cambiar el planificador de E/S del disco virtual en caliente, cuando se desee, sin necesidad de reiniciar el equipo, ya que el propio disco virtual aparece como un dispositivo de bloques regular.

0.3.2. Caso de uso: REDCAP

Con el objetivo de analizar la eficacia del simulador de disco implementado, hemos modificado REDCAP mejorando su algoritmo de activación–desactivación para que use el disco virtual. El simulador de disco implementa un modelo de disco más exacto, y proporciona estimaciones de tiempo de E/S más precisas que pueden mejorar y simplificar dicho algoritmo. Dependiendo del estado de REDCAP, el disco virtual simula el comportamiento de un disco real en un sistema normal (estado activo) o en un sistema REDCAP (estado inactivo).

Cuando la caché de REDCAP está activa, el disco virtual sirve las peticiones que son copia de las peticiones «originales», sin ninguna modificación. Esto nos permite simular un

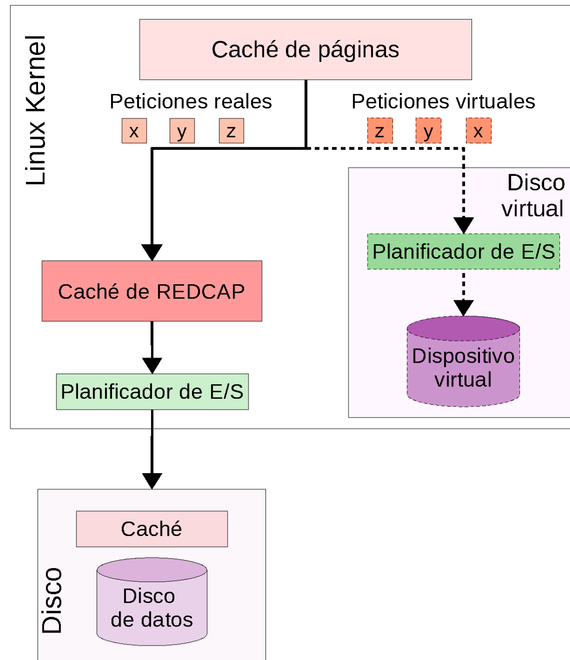


Figura 0.11: El disco virtual simulando un sistema «normal» cuando la caché de REDCAP está activa. Por simplicidad, hemos omitido las colas auxiliares del disco virtual.

sistema sin REDCAP. El tiempo que calcule el simulador del disco será el tiempo necesario para tratar las peticiones si REDCAP no estuviese funcionando. Sin embargo, cuando la caché está desactivada, el disco virtual procesa las peticiones para leer de disco los fallos de caché y las peticiones de *prefetching*. De este modo, el tiempo del disco virtual será el tiempo de los fallos de caché y del *prefetching* si REDCAP estuviese funcionando.

Esta nueva versión del algoritmo calcula los tiempos medios necesarios para servir un bloque de 4 kB en un sistema con REDCAP y en un sistema normal (sin REDCAP), y compara estos tiempos. Si el tiempo que necesita REDCAP es menor que el tiempo que necesita un sistema normal, REDCAP tiene que estar funcionando. En otro caso, debería desactivarse porque el sistema no está aprovechando el *prefetching* que realiza nuestro mecanismo.

0.3.3. Resultados experimentales

Para analizar el rendimiento que obtiene REDCAP cuando usa el disco virtual, hemos implementado los dos en un núcleo de Linux 2.6.23, al que llamamos *núcleo REDCAP–VD*. Además, hemos realizado una serie de experimentos para comparar el rendimiento que consigue el núcleo REDCAP–VD respecto a un núcleo de Linux 2.6.23 sin modificar, al que llamamos *núcleo original*.

La actividad de los discos de pruebas se ha registrado modificando los dos núcleos para que guarden información sobre cuándo empieza y acaba una petición, y cuando llega al planificador. El núcleo REDCAP–VD también almacena información sobre el comportamiento de su caché.

Plataforma hardware

Los experimentos se han realizando en un sistema Intel dual–core Xeon a 2,67 GHz, con 1 GB de memoria RAM y tres discos duros. Uno de los discos es el disco de sistema, que tiene como sistema operativo Fedora Core 8, y se usa para recoger las trazas. Los otros dos discos se usan para realizar las pruebas.

El primer disco de pruebas es un Seagate ST3400620AS [6] de 400 GB con una caché de 16 MB. Este disco tiene un sistema de ficheros Ext3, que sólo contiene los ficheros usados para hacer las pruebas. A este disco, a lo largo del documento, le llamaremos sistema de ficheros «nuevo».

El segundo disco es un Samsung HD322HJ [48] de 320 GB con una caché de 16 MB. Este disco contiene distintas particiones con sistemas de ficheros Ext3 envejecidos, que se han obtenido copiando sector a sector el disco del servidor de nuestro departamento. El sistema de ficheros que contiene los directorios *home* de los usuarios se ha seleccionado para realizar los tests. Este sistema de ficheros tiene 270 GB; en el momento de la copia estaba lleno al 84 %, y había estado en uso durante varios años. Los ficheros necesarios para realizar las pruebas se han creado en el mismo. A este disco, para distinguirlo del anterior, lo llamaremos sistema de ficheros «envejecido».

Benchmarks

En el estudio realizado se han ejecutado algunos de los *benchmarks* previamente usados para analizar el comportamiento de REDCAP (ver la sección 0.2.2). Los *benchmarks* utilizados, y ya descritos, son *Lectura del núcleo de Linux*, *Lectura IOR*, *TAC* y *Lectura a saltos de 512 kB*. Además, se ha modificado *Lectura a saltos de 4 kB* que ha pasado a ser *Lectura a saltos de 8 kB* y se han añadido dos nuevos *benchmarks*. Todos ellos se han ejecutado para 1, 2, 4, 8, 16 y 32 procesos. Los nuevos tests son:

- *Lectura a saltos de 8 kB (8k–SR, 8 kB Strided Read)*. Este test lee un fichero con un patrón de accesos a saltos: lee primero un bloque de 4 kB de la posición 0 del fichero, salta 8 kB, lee el siguiente bloque de 4 kB, salta otros 8 kB, etc. Esta prueba usa los mismos ficheros que IOR y TAC. 8k–SR está escrito en C y usa las funciones POSIX `read` y `lseek` para leer y avanzar en el fichero.

  Este test es diferente del usado inicialmente para analizar el comportamiento de REDCAP (ver sección 0.2.2). En ese primer estudio, se usaba un salto de 4 kB, pero el núcleo de Linux 2.6.23 es capaz ahora de detectar ese tamaño de salto y realiza *prefetching*. Puesto que nosotros queremos un patrón de acceso con saltos pequeños que no sea detectado por el sistema operativo, hemos cambiado el salto de 4 kB a 8 kB.
- *Lectura de directorios (DR)*. Este *benchmark* lee ficheros de unos directorios seleccionados del sistema de ficheros «envejecido» usando la orden:

  ```
  find -type f -exec cat {} > /dev/null \;
  ```

  Se han elegido 32 directorios *home*, uno por proceso, con tamaños comprendidos entre 1 GB y 3 GB. Este test sólo se ejecuta en el sistema de ficheros «envejecido», porque en el «nuevo» no están estos directorios.
- *Todos los benchmarks seguidos*. Los tests anteriores se ejecutan, uno detrás de otro, sin reiniciar el ordenador hasta que el último no ha terminado. Debido a que algunos de

estos tests usan los mismos ficheros, el orden de ejecución lo hemos establecido para tratar de reducir el efecto de la caché de *buffers*. En el sistema de ficheros «nuevo», el orden es: *TAC*; *512k–SR*; *8k–SR*; *LKR*; e *IOR*. En el «envejecido», el orden seguido es: *TAC*; *DR*; *512k–SR*; *8k–SR*; *LKR*; e *IOR*. El objetivo final de esta prueba es mostrar cómo el modelo de disco propuesto se adapta a los cambios que se producen en la carga de trabajo.

Precisión del modelo del disco virtual

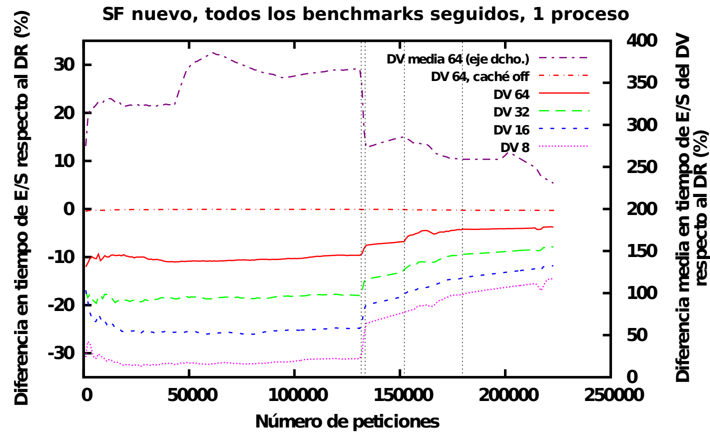
Para evaluar la exactitud del modelo de disco, hemos ejecutado el test *Todos los benchmarks seguidos* haciendo que tanto el disco real como el virtual sirvan las mismas peticiones, y hemos comparado los tiempos de E/S obtenidos por ambos discos. REDCAP no está activo ni está siendo simulado. Es importante destacar que el orden en el que cada disco sirve las peticiones puede ser distinto, puesto que, aunque ambos usan la misma política de planificación (el disco real CFQ y el virtual CFQ–VD), cada uno tiene su propia cola y, en un momento dado, pueden seleccionar una petición distinta.

Nos gustaría destacar que hemos seleccionado este test porque muestra cómo el disco virtual se adapta a los cambios en la carga de trabajo e, indirectamente, la precisión del modelo con todos los *benchmarks*. También se muestra cómo la adaptación dinámica de las tablas le permite al disco virtual seguir el comportamiento del disco real.

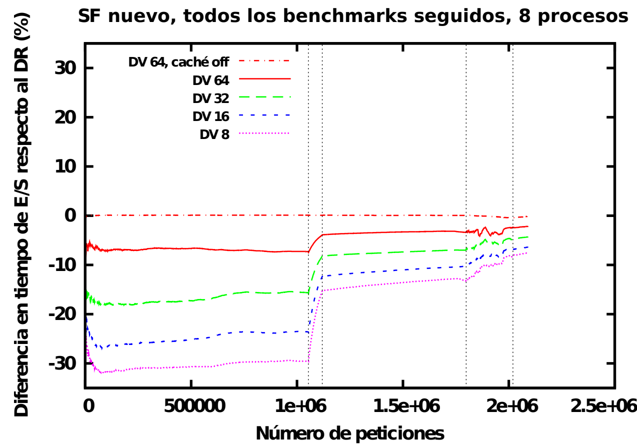
La figura 0.12 presenta la diferencia, en porcentaje de tiempo de E/S, del disco virtual con respecto al disco real para 1, 8 y 32 procesos, el sistema de ficheros «nuevo» y el planificador CFQ. También muestra la evaluación de distintas configuraciones del disco virtual basadas en el número de valores usados por celda para calcular la media: ocho («DV 8» en la figura 0.12), dieciséis («DV 16»), treinta y dos («DV 32») y sesenta y cuatro («DV 64»). Como podemos observar, cuando cada celda almacena la media de los últimos sesenta y cuatro valores, nuestro modelo presenta su mejor comportamiento, simulando el disco real de una manera más precisa.

Las mayores diferencias se observan al principio de la ejecución de la prueba, cuando el test TAC se ejecuta, debido a la caché del disco. Un patrón de acceso hacia atrás, como el producido por TAC, aprovecha la *lectura inmediata*[3] del disco duro, lo que produce un gran número de aciertos de caché. En un patrón de acceso secuencial sucede algo similar debido al *prefetching* realizado por la caché de disco. Sin embargo, ante un fallo, el tiempo que se necesita para leer los bloques solicitados es mayor en un acceso hacia atrás que en un acceso hacia delante, debido al proceso de búsqueda hacia atrás. Aunque la tabla de lecturas es actualizada con todos estos tiempos, la caché de disco tiene un notable impacto en los tiempos guardados en las celdas, haciendo que el disco virtual sea más rápido que el real. A pesar de este hecho, la configuración «DV 64» del disco virtual tiene un buen comportamiento, y su comportamiento es significativamente parecido al del disco real.

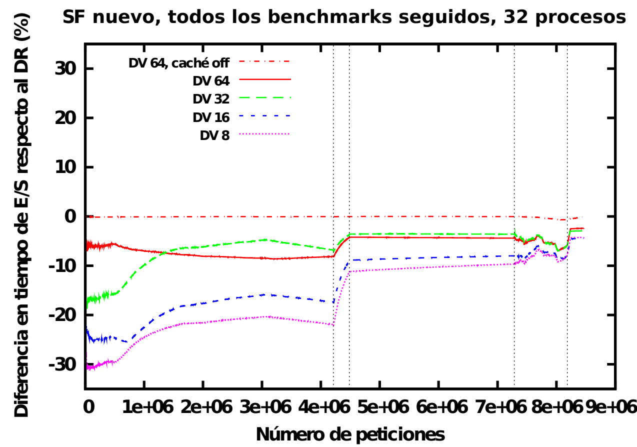
[3]Los discos duros usan una aproximación optimista para posicionar las cabezas del disco antes de una operación de lectura, e intentan leer tan pronto la cabeza está cerca de la pista correspondiente. Si el posicionamiento no se ha terminado y la cabeza está en una pista equivocada, no se ha perdido nada y simplemente se tiene que repetir la operación de lectura. Por el contrario, si la cabeza está bien posicionada y los datos se han leído correctamente, se ha ahorrado una vuelta completa [1]. Como consecuencia de este mecanismos, al que en esta tesis llamamos *lectura inmediata*, es posible que los bloques anteriores a los solicitados se lean y guarden en la caché de disco. El efecto es como si la controladora de disco hiciese *prefetching* hacia atrás, pero no lo está realizando y tampoco es capaz de detectar este patrón de acceso.



(a) 1 proceso.



(b) 8 procesos.



(c) 32 procesos.

Figura 0.12: Diferencia, en porcentaje de tiempo de E/S, del disco virtual con respecto al disco real en el test *Todos los benchmarks seguidos*, usando el sistema de ficheros «nuevo» y el planificador CFQ, para 1 (a), 8 (b) y 32 (c) procesos. Las líneas verticales marcan el final de un benchmark y el inicio del siguiente. El orden de ejecución de los test es: TAC, 512k–SR, 8k–SR, LKR e IOR.

Después de la ejecución del TAC, la diferencia entre los dos discos disminuye de una forma rápida. Para la configuración «DV 64», esta diferencia es de media menor de un 5 %. De hecho, podemos afirmar que *el disco virtual coincide en gran medida con el disco real*. Esta rápida adaptación se produce porque sólo un pequeño porcentaje de celdas de la tabla son usadas y actualizadas, incluso cuando la prueba se ejecuta para 32 procesos. Esto lo podemos observar en la figura 0.13, que muestra, para la tabla de lecturas, las celdas que se han modificado una vez que se ha terminado de ejecutar la prueba. El eje X muestra la distancia entre peticiones (las columnas de la tabla) y el tamaño de las peticiones es mostrado en el eje Y (las filas de la tabla).

Una razón por la que el comportamiento del disco virtual no es el mismo que el comportamiento del disco real es la dificultad de simular la caché del disco duro. Para analizar esta influencia, hemos ejecutado el mismo test, pero haciendo que la caché del disco esté desactivada, y sólo para la configuración «DV 64». Los datos se muestran en la figura 0.12 en la línea «DV 64, cache off». Como se puede observar, sin la caché de disco, el disco virtual iguala al real de una manera muy precisa, con una diferencia de menos del 0,2 %.

Finalmente, para mostrar que el tamaño de la petición es importante en nuestro modelo de disco, hemos ejecutado la misma prueba, pero sin tener en cuenta el tamaño de las peticiones. Dada una distancia entre peticiones, hemos usado la media de los valores para la columna correspondiente. Este estudio ha sido realizado sólo para 1 proceso (línea «DV media 64» en la figura 0.12(a)), pero esta única ejecución es suficiente para ver que si no tenemos en cuenta el tamaño de la petición, las diferencias entre el modelo de disco propuesto y el disco real son muy grandes.

Resultados

Para analizar el comportamiento de REDCAP cuando usa el disco virtual hemos realizado cinco ejecuciones para cada test y sistema de ficheros con ambos núcleos, REDCAP–VD y original. Los resultados mostrados son la media de estas cinco ejecuciones e incluimos, como barras de error, los intervalos de confianza para un nivel de confianza del 95 %.

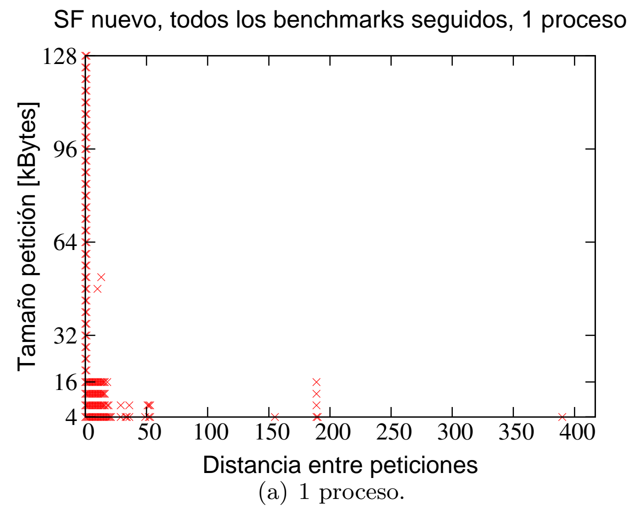
El equipo se reinicia después de cada ejecución. Las tablas obtenidas del entrenamiento *off–line* se dan al disco virtual cada vez que se inicializa el sistema. En cada celda se almacenan los últimos sesenta y cuatro valores.

El tamaño de la caché de REDCAP se ha establecido en 64 MB, dividida en 512 segmentos de 128 kB cada uno. En todos los tests el estado inicial de REDCAP es activo.

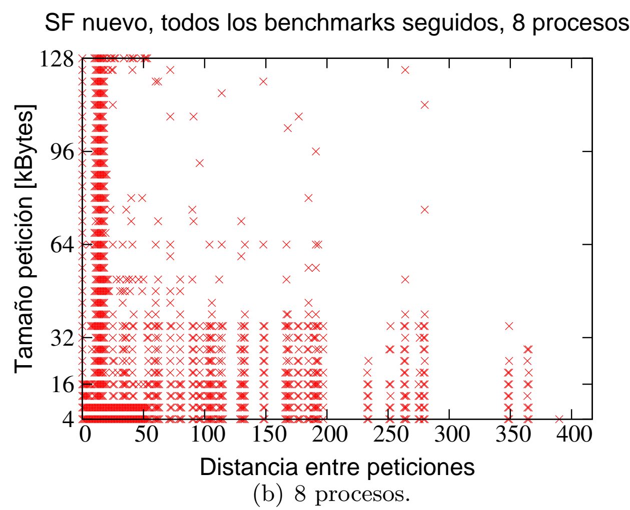
En las pruebas hemos usado el planificador CFQ en el disco real y el planificador CFQ–VD en el disco virtual (ver la sección 0.3.1).

Benchmarks ejecutados independientemente. Primero vamos a analizar los resultados de ejecutar los tests de forma independiente. Las figuras 0.14 y 0.15 muestran la mejora introducida por REDCAP con respecto al núcleo original para los sistemas de ficheros «nuevo» y «envejecido», respectivamente. Para facilitar la comparación, los test se han ordenado en las figuras en el mismo orden que se ejecutan en el test *Todos los benchmarks seguidos*.

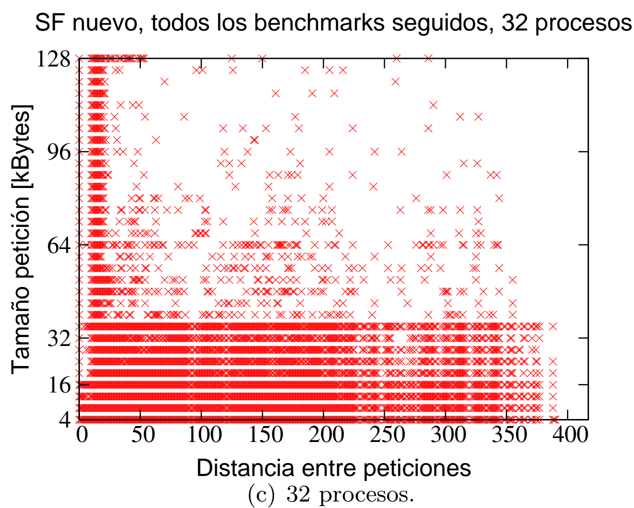
TAC. Con este test, REDCAP siempre mejora al núcleo original, obteniendo mejoras de hasta un 28,4 %. El sistema operativo no es capaz de detectar el patrón de acceso hacia



(a) 1 proceso.



(b) 8 procesos.



(c) 32 procesos.

Figura 0.13: Celdas modificadas en la tabla de lectura una vez que el test *Todos los benchmarks seguidos* se ha ejecutado, para el sistema de ficheros «nuevo», el planificador CFQ y 1 (a), 8 (b) y 32 (c) procesos.

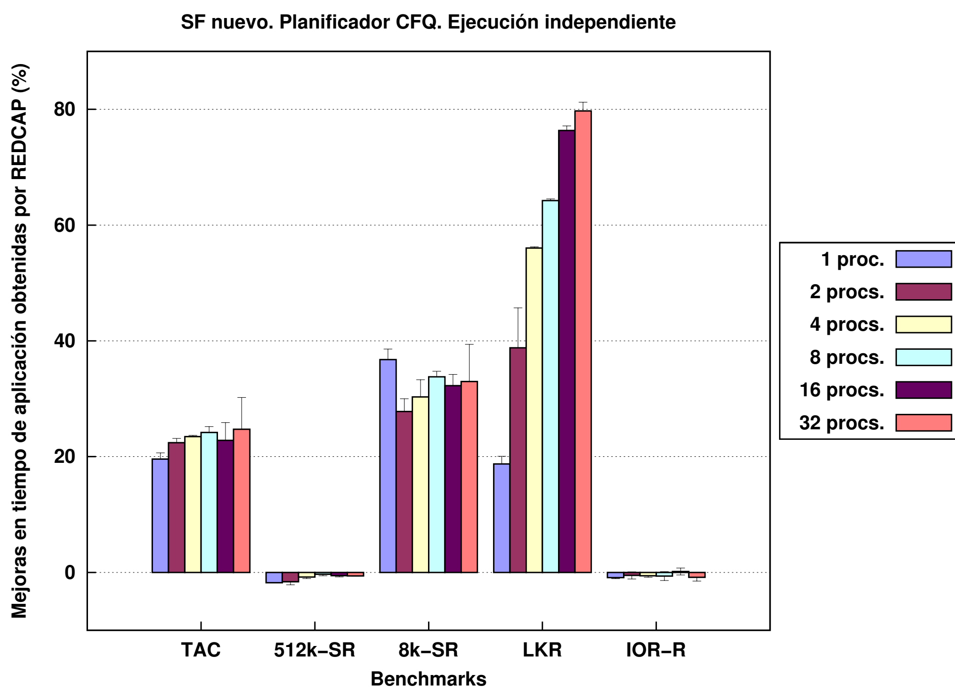


Figura 0.14: Mejora, en tiempo de aplicación, alcanzada por REDCAP sobre un núcleo de Linux sin modificar, ejecutando los tests independientemente, en un sistema de ficheros «nuevo».

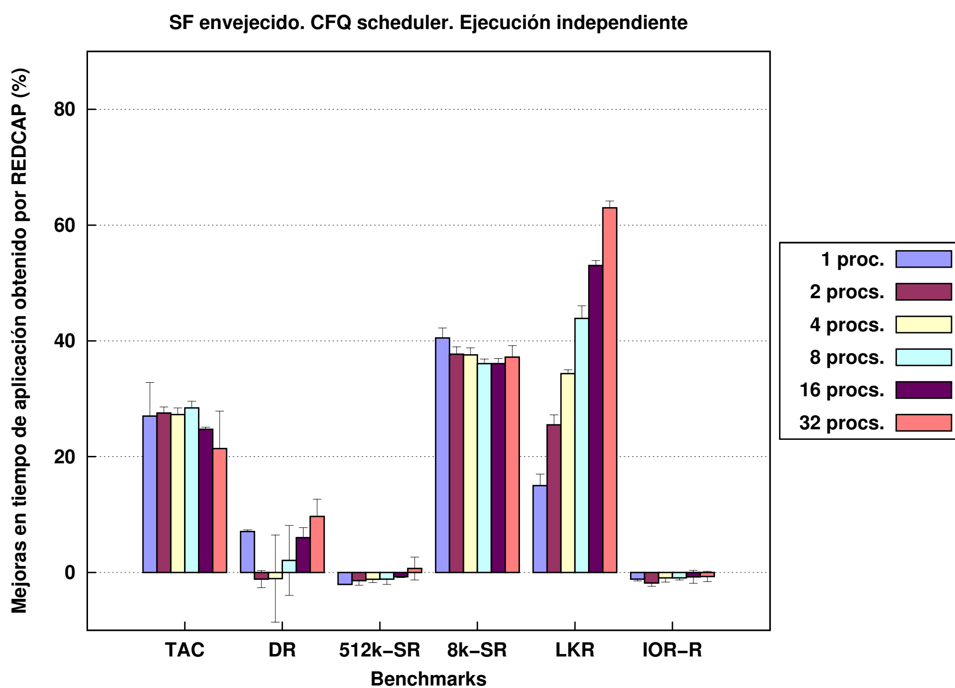


Figura 0.15: Mejora, en tiempo de aplicación, alcanzada por REDCAP sobre un núcleo de Linux sin modificar, ejecutando los tests independientemente, en un sistema de ficheros «envejecido».

atrás, y no realiza ningún tipo de *prefetching*. Sin embargo, REDCAP sí lo detecta y su caché está activa la mayor parte del tiempo que dura esta prueba.

Lectura de directorios. En esta prueba, sólo ejecutada en el sistema de ficheros «envejecido», nuestro método reduce el tiempo de aplicación hasta en un 9,7 %. La excepción está para 2 y 4 procesos, donde podemos decir que, debido a los intervalos de confianza, ambos núcleos presentan casi el mismo comportamiento.

Lectura a saltos 512kB. Para este *benchmark* REDCAP no realiza ninguna contribución porque su caché no es efectiva, siendo casi imposible aprovechar el *prefetching* que hace. El algoritmo de activación–desactivación detecta este hecho y desactiva la caché, la cual está desactivada durante todo el test. REDCAP se comporta casi igual que el núcleo original y, estadísticamente, ambos presentan el mismo rendimiento. Los peores resultados se obtienen para 1 proceso y el sistema de ficheros «envejecido» con una degradación de sólo el 2,1 %. Este pequeño incremento en el tiempo de aplicación se debe al tiempo perdido al principio de la ejecución, cuando la caché aún estaba activa.

Lectura a saltos 8 kB. En este caso, REDCAP siempre obtiene un mejor rendimiento que el núcleo original. Con este patrón de acceso, nuestra técnica alcanza sus mejores resultados cuando su caché está activa todo el tiempo, algo que sucede en la ejecución de este test. El sistema operativo no detecta este patrón de acceso y no implementa ninguna técnica de *prefetching*. Sin embargo, con nuestra técnica, la mayoría de las peticiones aprovechan el *prefetching* realizado por REDCAP, ya que 9 de cada 10 peticiones son acierto de caché. Por tanto, se alcanzan reducciones de hasta un 37 % y un 45 % para los sistemas de ficheros «nuevo» y «envejecido», respectivamente. Es importante resaltar que esta nueva implementación del algoritmo de activación–desactivación usando el disco virtual soluciona los problemas que aparecían en la primera versión con la prueba *Lectura a saltos 4 kB*. Ahora, el algoritmo es capaz de decidir el estado correcto de la caché, y REDCAP obtiene la máxima mejora posible con el test de *Lectura a saltos 8 kB*.

Leer núcleo de Linux. El método propuesto siempre mejora al núcleo original para este *benchmark*. Nuestra caché está todo el tiempo activa con ambos sistemas de ficheros, permitiendo esto que el tiempo de aplicación disminuya. Las reducciones que consigue REDCAP son muy significativas y, además, son mayores a mayor número de procesos. Para 32 procesos, el tiempo de aplicación se reduce hasta un 80 % y un 63 % para los sistemas de ficheros «nuevo» y «envejecido», respectivamente.

Lectura IOR. Puesto que este test tiene un patrón de acceso secuencial, y las técnicas de *prefetching* del sistema operativo y de la caché del disco duro están optimizadas para esta carga de trabajo, la contribución de REDCAP es muy pequeña. Además, debido a que los tiempos de E/S de ambos discos (real y virtual) son muy parecidos, algunas veces, el algoritmo no es capaz de determinar el estado correcto, y cambia sucesivamente el estado de activo a inactivo, y viceversa. Sin embargo, la mejor opción sería mantener REDCAP desactivado. Para ambos sistemas de ficheros, el comportamiento del núcleo REDCAP–VD es muy similar

al del núcleo original y, teniendo en cuenta los intervalos de confianza, los dos presentan el mismo rendimiento.

Todos los benchmarks seguidos. Las figuras 0.16 y 0.17 muestran la mejora introducida por REDCAP con respecto al núcleo original cuando se ejecuta el test *Todos los benchmarks seguidos*, para los sistemas de ficheros «nuevo» y «envejecido», respectivamente.

Los resultados son similares a los obtenidos cuando las pruebas se ejecutan independientemente. De hecho, podemos afirmar que el disco virtual se adapta bastante rápido a los cambios de la carga de trabajo. Sólo se observan pequeñas diferencias, debidas tanto a la caché de *buffers* como a la caché de REDCAP. Explicamos, a continuación, estas diferencias.

Sistema de ficheros «nuevo» y 1 proceso. Cuando el test TAC termina, una gran cantidad de bloques del fichero leído permanecen en la caché de *buffers*. Así, 512k–SR sólo tiene que leer un pequeño número de bloques del final del fichero. Después de la primera serie de lecturas, todos los siguientes bloques solicitados por las otras tres series están ya en la caché de *buffers* o en la caché de REDCAP. Sin embargo, con el núcleo original, se tienen que leer todos estos bloques. Por esta razón, REDCAP consigue, de forma inesperada, una mejora del 50 %.

Cuando 8k–SR se ejecuta, nuestro método lee más bloques del fichero que el núcleo original, de ahí que se reduzca la mejora de un 37 % a un 5 %. El problema es el tamaño de la imagen del núcleo: la imagen del núcleo REDCAP–VD es más grande que la del núcleo original. Por lo tanto, después de la ejecución de los dos primeros tests, con el núcleo REDCAP–VD, hay menos bloques del fichero en la memoria.

Al finalizar 8k–SR, 4 de cada 12 bloques del fichero están en la memoria RAM, y cuando LKR termina todos estos bloques están todavía en la memoria porque no han sido expulsados. Esto implica que la prueba IOR produce un patrón de acceso a saltos (siendo 20 sectores el tamaño de la petición más grande), lo que impide que el núcleo original realice *prefetching* lanzando peticiones más grandes. Por otro lado, el *prefetching* que realiza REDCAP se usa casi en su totalidad, con lo que se alcanza una mejora del 32 %.

Sistema de ficheros «nuevo» y 2 procesos. Cuando termina la ejecución de TAC, parte de los ficheros leídos por los dos procesos están en memoria. Pero, debido al tamaño de las imágenes del núcleo, con REDCAP hay menos bloques de los ficheros en memoria. Así, cuando se ejecuta 512k–SR, nuestra técnica tiene que leer más bloques que el núcleo original. Además, los datos que hay que leer ya no caben en la caché de REDCAP. Por ello, se produce una degradación del 5 %.

Sistema de ficheros «envejecido» y 1 proceso. Con este sistema de ficheros, hay un resultado inesperado para IOR y 1 proceso. La razón es la misma dada anteriormente. El núcleo original no puede lanzar peticiones de *prefetching*, mientras que el *prefetching* que realiza REDCAP se usa casi completamente, lo que hace que el tiempo de aplicación se reduzca en un 22 %.

0.3.4. Conclusiones

En esta sección hemos presentado el diseño y la implementación de un disco virtual dentro del núcleo de Linux que tiene varias propiedades interesantes: i) crea un simulador de disco

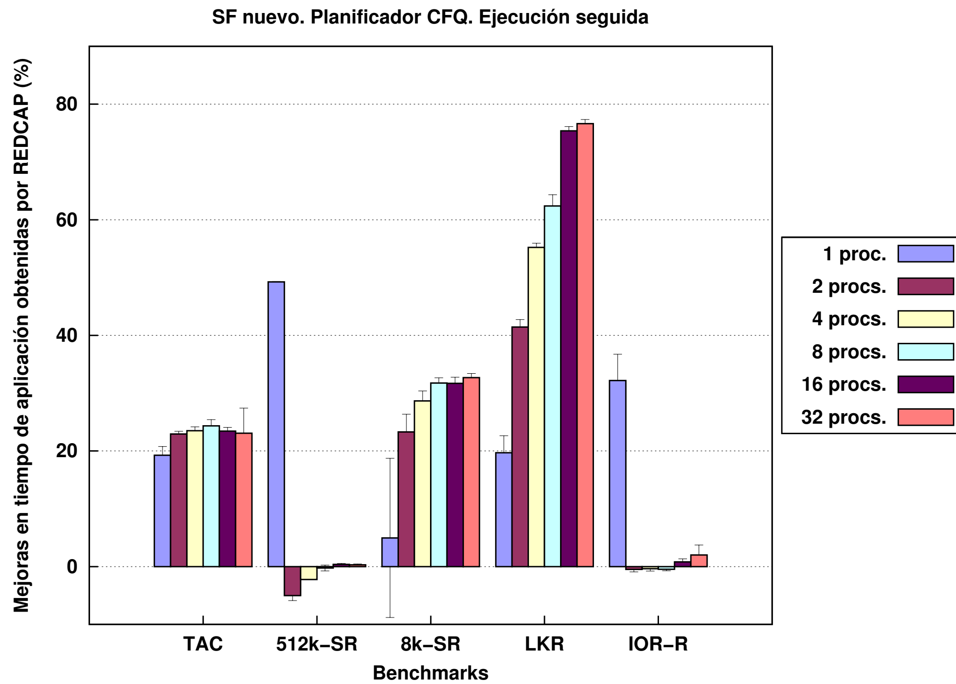


Figura 0.16: Mejora, en tiempo de aplicación, alcanzada por REDCAP sobre un núcleo de Linux sin modificar, ejecutando los tests seguidos, en un sistema de ficheros «nuevo».

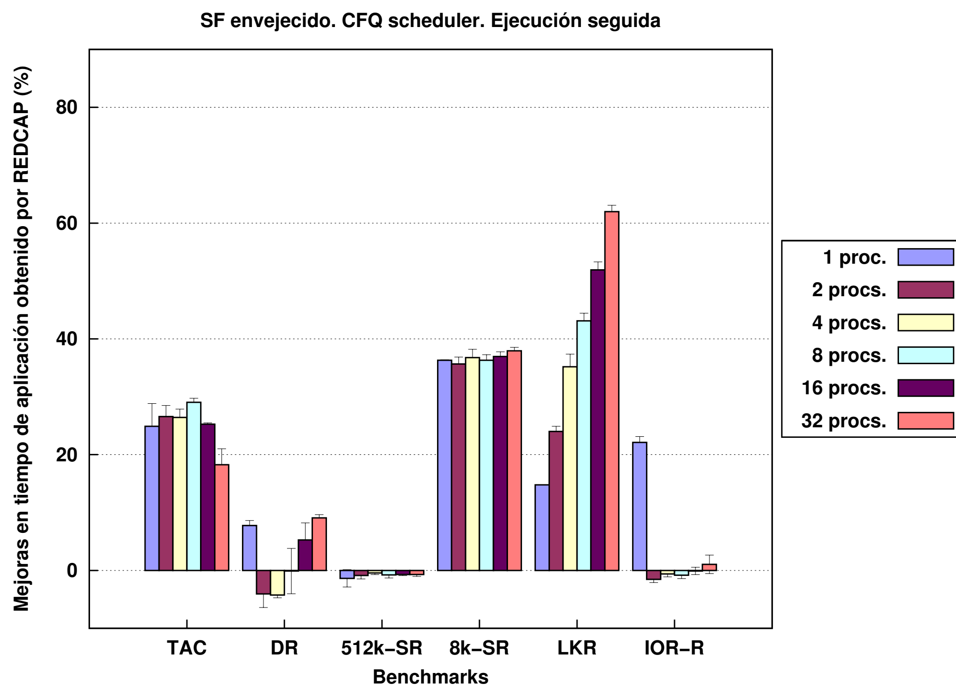


Figura 0.17: Mejora, en tiempo de aplicación, alcanzada por REDCAP sobre un núcleo de Linux sin modificar, ejecutando los tests seguidos, en un sistema de ficheros «envejecido».

que es capaz de simular cualquier disco usando una tabla de tiempos de E/S; ii) puesto que tiene el mismo interfaz que cualquier dispositivo de bloques, puede usar cualquier planificador de E/S con mínimas modificaciones; iii) no interfiere en el camino «regular» de las peticiones de E/S; iv) la sobrecarga que supone el simulador es despreciable; v) al actualizar la tabla de forma dinámica mientras las peticiones son atendidas por el disco real, el comportamiento del disco es modelado de una manera precisa; y vi) simula el orden de servicio de las peticiones en un disco real al considerar las dependencias entre ellas.

Nuestro disco virtual se puede usar para evaluar simultáneamente diferentes mecanismo de E/S, o para dinámicamente activarlos o desactivarlos dependiendo del rendimiento conseguido. Especialmente, hemos descrito cómo REDCAP puede utilizar el disco virtual para mejorar su algoritmo de activación–desactivación. Los resultados obtenidos muestran que usando el nuevo simulador, REDCAP alcanza las máximas mejoras posibles. Además, los nuevos resultados son consistentes con los obtenidos en el análisis previo (ver sección 0.2.2). REDCAP reduce significativamente el tiempo de E/S de las peticiones de lectura para cargas de trabajo con alguna localidad espacial. De hecho, el tiempo de aplicación se reduce hasta un 80 %. Por otro lado, el rendimiento de REDCAP es igual que el de un sistema tradicional (sin ninguna modificación) para cargas de trabajo con patrones de acceso aleatorios o de largas lecturas secuenciales.

0.4. Selección automática y dinámica del planificador de E/S

Los principales objetivos de un planificador de E/S son optimizar el tiempo de acceso a disco y maximizar el rendimiento. Con estos propósitos, el planificador decide el orden de las peticiones en su cola y cuándo cada petición es enviada al dispositivo. Sin un planificador de E/S, el sistema operativo enviaría las peticiones en el mismo orden en que las ha recibido, y el rendimiento de la E/S, e indirectamente del sistema, serían normalmente muy malos.

A lo largo de los años, han sido muchas las políticas de planificación que se han propuesto para mejorar el rendimiento de la E/S. Algunas de ellas tratan de minimizar el tiempo de búsqueda, otras propuestas tienen también en cuenta la latencia rotacional del disco, y otras incluso asignan a cada petición un límite de tiempo de servicio e intentan no violar este límite. Sin embargo, ninguno de los algoritmos de planificación actuales es óptimo, en el sentido de que los rendimientos que obtienen dependen de varios factores: características de la carga de trabajo, sistemas de ficheros, discos duros, etc.

Los sistemas operativos como Linux disponen de varios planificadores de E/S y, en tiempo de arranque o durante el funcionamiento del sistema, el administrador puede elegir uno, e incluso uno diferente para cada disco del sistema. Pero, elegir, para cada disco, el planificador que siempre proporcione el mejor rendimiento no es una tarea fácil. De hecho, la mayoría de las veces, los administradores no realizan ninguna selección, y se usa el planificador de E/S proporcionado por defecto, cuando uno diferente podría mejorar el rendimiento.

Motivados por estas ideas, presentamos el diseño y la implementación de un marco de planificación de disco automático y dinámico, llamado *DADS* (*Dynamic and Automatic Disk Scheduling framework*), que es capaz de seleccionar el mejor planificador de E/S de Linux comparando el rendimiento obtenido por cada uno de ellos en un momento dado [37].

El objetivo de DADS es incrementar el ancho de banda de E/S proporcionado a las aplicaciones por el sistema operativo. Este ancho de banda queda determinado por el disco duro

y, especialmente, por el planificador de E/S. El primero determina el tiempo de E/S de una petición. El segundo no sólo establece el orden en el que las peticiones van a ser atendidas (en el caso de un disco duro este orden podría determinar el rendimiento del servicio), sino que también añade un tiempo de espera en la cola del planificador que incrementa el tiempo de servicio de una petición (y, de hecho, reduce el ancho de banda). Por lo tanto, en nuestro caso, el mejor planificador es el que proporciona el mayor ancho de banda de E/S a las aplicaciones para un disco duro dado.

0.4.1. Diseño de DADS

DADS es un mecanismo que compara distintos planificadores de E/S de Linux y automáticamente y dinámicamente selecciona el que obtiene el mejor rendimiento de E/S. Por simplicidad, la primera versión implementada sólo compara dos planificadores, sin embargo, las modificaciones necesarias para soportar más planificadores son sencillas.

Para realizar el análisis, usamos una versión mejorada del simulador de disco presentado en la sección 0.3. DADS ejecuta, dentro del núcleo de Linux, dos instancias del nuevo simulador, que se configuran para imitar el comportamiento del disco real usado en las pruebas, y evalúa el rendimiento de cada planificador. Una de las instancias, llamada *VD_RD* (Virtual Disk of the Real Disk), tiene el mismo planificador de E/S que el disco real y simula, por tanto, su comportamiento. La otra instancia, llamada *VD_VD* (Virtual Disk of the Virtual Disk), tiene el planificador de E/S con el que se va a realizar la comparación, simulando, de esta manera, el comportamiento del disco real con un planificador diferente. La figura 0.18 muestra el esquema general del mecanismo de intercambio de planificador propuesto.

Al usar dos instancias, y no el disco real y una instancia del simulador de disco, se realiza una comparación más justa, que nos permite saber que las diferencias son debidas al rendimiento del planificador, y no a un error en la propia simulación del disco.

El tiempo de servicio de una petición se calcula como el tiempo que pasa desde que la petición es insertada en la cola del planificador de E/S hasta que termina. DADS mide el rendimiento de un planificador como la suma de los tiempos de servicio de todas las peticiones que sirve la instancia que tiene ese planificador. Por tanto, nuestra propuesta selecciona el planificador que optimiza el tiempo de servicio total.

0.4.2. Modificación del disco virtual

El nuevo simulador de disco, usado para implementar DADS, consiste en dos subsistemas: un *disco virtual*, que trabaja como un controlador de dispositivos y como un disco duro en sí, y un *simulador de la llegada de peticiones* (*RAS*, *Request Arrival Simulator*), que simula cómo llegan las peticiones al disco virtual y cómo se insertan en la cola del planificador.

El disco virtual se implementa usando un hilo de núcleo que continuamente ejecuta la siguiente rutina:

1. Recoge la siguiente petición de la cola del planificador.
2. Obtiene, del modelo de tabla del disco real, el tiempo de E/S estimado que necesita para atender la petición.
3. Duerme el tiempo estimado para simular que la operación de disco se están realizando.
4. Después de despertar, completa la petición, e informa a RAS.

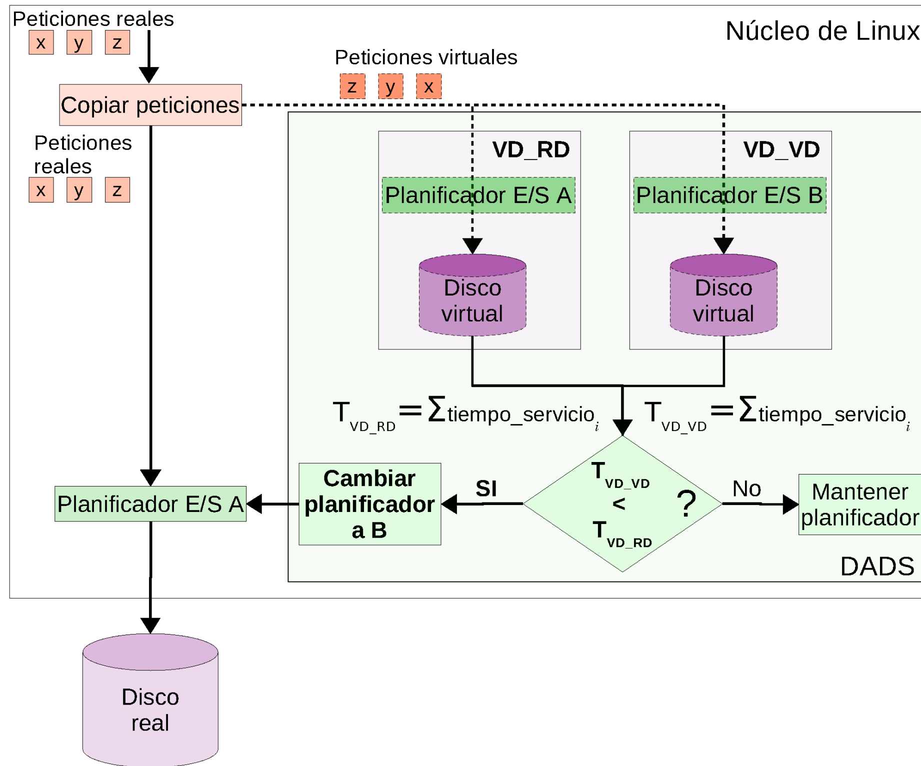


Figura 0.18: Esquema general del diseño de DADS.

RAS es también un hilo del núcleo que inserta las peticiones en la cola del planificador de E/S simulando la llegada de las mismas al controlador del dispositivo de bloques (ver la sección 0.4.3). RAS simula el tiempo de pensar (*thinking time*): el tiempo que pasa entre que se termina una petición y llega la siguiente lanzada por la misma aplicación. Puesto que las peticiones virtuales de un proceso tienen que ser servidas en el mismo orden que fueron emitidas, RAS también controla las dependencias entre peticiones y permite que el disco virtual las atienda en el orden «correcto».

Al modelo de disco basado en tabla presentado en la sección 0.3, le hemos añadido una nueva tabla para tiempos de E/S de lectura y una caché de disco simulada.

Puesto que las operaciones de lectura toman tiempos diferentes dependiendo de si son aciertos o fallos de caché, ahora se usan dos tablas de lectura: una para las peticiones que son fallos de caché y otra para las que son aciertos de caché. Por lo tanto, el nuevo modelo maneja un total de tres tablas. Las tres tablas tienen la misma estructura, la única diferencia es el tipo de valores que almacenan en cada celda: tiempos de E/S de escrituras, tiempos de E/S de lecturas de fallos de caché y tiempos de E/S de aciertos de caché.

Como ya hemos dicho, aunque el tiempo de E/S de una petición puede depender de varios factores [1], para predecir su tiempo de disco, nuestro modelo sólo usa su tipo, su tamaño, la distancia con la petición anterior y, para las operaciones de lectura, la información sobre si es un acierto o fallo de caché. Así, dada una petición, su tipo y la caché de disco simulada determinan la tabla a usar, su tamaño especifica la fila y su distancia con la petición anterior la columna. La correspondiente celda da el tiempo de E/S de la petición. Como en el modelo

original, los valores de la celda son dinámicamente actualizados mediante los tiempos de E/S proporcionados por el disco real cuando sirve las peticiones enviadas por las aplicaciones (dese cuenta que el simulador de disco nunca envía peticiones al disco real). Esto permite que los valores de las celdas se adapten a las características de la carga de trabajo. Especialmente, el valor de cada celda es calculado como la media de los últimas 64 muestras tomadas para la misma.

Es importante destacar que si sólo usásemos una tabla de lecturas, como en el modelo inicial, las estimaciones de los tiempos de lectura no serían tan exactas como necesitamos, puesto que el valor de una celda sería la media de tiempos de aciertos y de fallos de caché, y las diferencias entre estos tiempos normalmente son muy grandes. DADS, sin embargo, necesita los tiempos de aciertos y fallos de caché, porque la proporción de aciertos de caché producidos por un planificador determina, en gran medida, su rendimiento. Por tanto, necesitamos una caché de disco para realizar una simulación que nos diga cuándo una petición es acierto o fallo.

Respecto a la tabla de escrituras, sólo se necesita una tabla porque las cachés de disco normalmente usan las técnicas de *escritura diferida* y *respuesta inmediata* [49], y una petición de escritura se considera «hecha» tan pronto como llega a la caché.

Son muchas las propiedades y características que especifican el comportamiento de una caché de disco [1, 49], el problema es que la mayoría de ellas son consideradas «secreto comercial». De hecho, no es una tarea sencilla simular una caché de disco, especialmente si la caché tiene un comportamiento dinámico. Para reducir el número de posibilidades, hemos modelado una caché con *read–ahead* y *respuesta inmediata*, que está dividida en segmentos de igual tamaño, y que usa LRU como algoritmo de reemplazo. No hemos considerado una división dinámica de la caché.

Hemos implementado un modelo de caché que sólo realiza *read–ahead* en los fallos de caché o en los aciertos parciales. Además, hemos definido una política de *read–ahead* adaptativa que usa dos tamaños distintos: uno para los accesos secuenciales que, además, nos dará el tamaño máximo de *read–ahead*, y otro para los accesos aleatorios.

El número de segmentos y los tamaños de *read–ahead* son calculados con un programa de captura que hemos implementado usando las indicaciones dadas por Worthington *et al.* [45] y por Schindler y Ganger [50]. El tamaño de la caché lo obtenemos de las especificaciones dadas por los fabricantes.

Somos conscientes de que el modelo de caché propuesto no simula todas las propiedades de una caché de disco real y es sólo una aproximación. Pero nuestra intención es desarrollar una caché que se parezca lo suficiente para que nos permita estudiar el rendimiento de un sistema con diferentes planificadores. Los resultados obtenidos muestran que este modelo de caché cumple con estos requisitos.

El disco virtual tiene un planificador de E/S que maneja la cola de peticiones. Además, puesto que para el sistema operativo es un dispositivo de bloques más, se puede cambiar su planificador en caliente.

Usamos las adaptaciones de los planificadores CFQ [42] y AS [47] (*CFQ–VD* y *AS–VD*, respectivamente) para funcionar con el disco virtual. Ahora, la única diferencia es que las peticiones son insertadas por RAS. El problema que hay con estos dos planificadores es que usan información del proceso que envió la petición para ordenar la cola del planificador, y en el simulador todas las peticiones son insertadas por RAS. Es importante remarcar que los

nuevos planificadores se comportan como los originales, pero trabajando con el simulador de disco. Por otro lado, los planificadores Noop y Deadline [42] se usan sin ninguna modificación.

Finalmente, nos gustaría resaltar que el simulador de disco presentado aquí presenta importantes mejoras respecto a la primera versión del simulador de disco. Las diferencias principales son:

- En la implementación previa RAS no fue implementado, por lo que el propio disco virtual insertaba las peticiones en la cola del planificador y controlaba las dependencias entre peticiones. En la implementación actual, el disco virtual sólo simula el proceso de E/S y RAS gestiona la llegada de las peticiones.
- Gracias a RAS, la simulación de la llegada de las peticiones es más precisa: una petición se puede insertar en la cola del planificador mientras el disco virtual está atendiendo otra petición.
- El tiempo de cómputo de las peticiones no se tenía en cuenta en la primera versión: las peticiones se insertaban en la cola del planificador en el momento en que sus dependencias se resolvían.
- El modelo de disco para estimar los tiempos de E/S de cada petición también es diferente. Ahora se usa una tercera tabla para las peticiones de lectura que producen aciertos de caché, y la caché de disco se simula directamente. En la versión previa, sólo se usaba una tabla para las operaciones de lectura y el efecto de la caché se capturaba actualizando dinámicamente las tablas.

0.4.3. Implementación de DADS

Como hemos comentado anteriormente, DADS se ha implementado para seleccionar entre dos planificadores de E/S de Linux usando dos instancias del simulador de disco, y comparando sus tiempos de servicio. Ambas instancias sirven las mismas peticiones, que son una copia de las enviadas al disco real (ver la figura 0.18).

Una primera cuestión a solucionar es que los planificadores de los discos virtuales (y, de hecho, las dos instancias del simulador del disco) tienden a tener un comportamiento parecido. El problema es que el orden de llegada de las peticiones a las instancias *VD_VD* y *VD_RD* depende del orden en el que fueron atendidas las peticiones en el disco real.

Para evitar este problema de mimetización, el proceso de simulación se ejecuta en tres fases:

1. La primera fase recibe las peticiones del disco real. Durante un intervalo de tiempo, el sistema copia a RAS las peticiones emitidas por el disco real. Por cada petición virtual, RAS calcula su tiempo de cómputo y sus dependencias. Ninguna petición se encola en el planificador y no se realiza ninguna simulación.
2. La segunda fase corre la simulación propiamente dicha. RAS encola las peticiones en el planificador cuando sus dependencias han sido resueltas y su tiempo de cómputo ha pasado. Al mismo tiempo, el disco virtual sirve las peticiones simulando el comportamiento del disco real. Por cada petición servida, se calcula su tiempo de servicio. Durante esta fase, el sistema no copia ninguna petición nueva. La fase termina cuando las dos instancias han servido todas las peticiones recogidas en la primera etapa.
3. La última fase compara el tiempo total de servicio alcanzado por las dos instancias del simulador de disco. Si el rendimiento obtenido por *VD_VD* mejora el tiempo obtenido por *VD_RD*, es de esperar que el rendimiento del disco real también se mejore con un

cambio de planificador. Por lo tanto, los planificadores del disco real y de *VD_VD* son intercambiados. El planificador de *VD_RD* también se intercambia para que coincida con el que finalmente se queda en el disco real. Una vez que esta fase ha terminado, el proceso empieza otra vez por la fase inicial recogiendo nuevas peticiones.

Debido a que el proceso de cambio de planificador consume tiempo (la cola del planificador actual tiene que ser vaciada y la nueva inicializada), el cambio es hecho si, y sólo si, la mejora que alcanza *VD_VD* es mayor de un 5 %. Además, se considera una estimación del tiempo necesario para realizar el cambio. Así, si T_{VD_RD} y T_{VD_VD} son los tiempos de servicio de *VD_RD* y *VD_VD*, respectivamente, y T_{Change} es la estimación del tiempo necesario para realizar el cambio, el intercambio de planificador se hace cuando:

$$T_{VD_VD} + T_{Change} < 0{,}95 \cdot T_{VD_RD}. \quad (0.6)$$

0.4.4. Resultados experimentales

DADS y el nuevo simulador de disco han sido implementados en un núcleo de Linux 2.6.23 (al que llamaremos *núcleo DADS*). Hemos ejecutado una serie de experimentos comparando dos a dos los planificadores de Linux, aunque aquí sólo mostramos los resultados obtenidos por CFQ y Deadline. Los resultados han sido comparados con los obtenidos en un núcleo de Linux 2.6.23 sin modificar (al que llamaremos *núcleo original*) con los mismos planificadores.

Plataforma hardware

Hemos usado dos ordenadores con tres discos cada uno. Uno de los discos es el disco de sistema, que tiene como sistema operativo Fedora Core 8 y se usa para recoger las trazas para poder evaluar nuestro mecanismo. Los otros dos son discos de pruebas.

Un ordenador es un Intel dual–core Xeon a 2,67 GHz con 1 GB de RAM. Su disco de sistema es un disco Seagate ST3500630AS [6]. Su primer disco de pruebas es un Seagate ST3250310NS [6], con una capacidad de 250 GB y una caché de 32 MB. Tiene un sistema de ficheros nuevo que sólo contiene los ficheros para realizar las pruebas. Fue formateado y después los ficheros fueron creados. Durante la explicación de los resultados nos referiremos a este disco como «HD–250–32»[4].

El segundo es un disco Samsung HD322HJ [48], con una capacidad de 320 GB y una caché de 16 MB. Este disco contiene varias particiones con sistemas de ficheros envejecidos, obtenidos tras copiar sector a sector el disco del servidor de nuestro departamento. El sistema de ficheros que contiene los directorios de los usuarios ha sido seleccionado para realizar las pruebas, tiene un tamaño de 270 GB, ha estado en uso durante varios años y, en el momento de hacer la copia, estaba lleno al 84 %. En él hemos creado los ficheros para realizar los tests. Nos referiremos a este disco como «HD–320–16» durante la explicación de los resultados.

El segundo ordenador es un Intel dual–core a 1,86 GHz, cuyo disco de sistema es un disco Seagate ST3400620AS [6]. Su primer disco de pruebas es un Seagate ST3500630AS [6], con una capacidad de 500 GB y una caché de 16 MB. El segundo es un Seagate ST3500320NS [6], con un tamaño de 500 GB y 32 MB de caché. Ambos tienen un sistema de ficheros nuevo, que

[4] Notar que en el nombre «HD–250–32», «HD» viene de *Hard Disk*, «250» es la capacidad y «32» es el tamaño de su caché.

sólo contiene los ficheros para ejecutar los tests. A lo largo de la explicación nos referiremos a estos discos como «HD–500–16» y «HD–500–32», respectivamente.

Benchmarks

Hemos analizado el rendimiento de DADS ejecutando los *benchmarks* descritos en las secciones 0.2.2 y 0.3.3: *Lectura del núcleo de Linux*, *Lectura IOR*, *TAC*, *Lectura a saltos de 8 kB* y *Lectura a saltos de 512 kB*.

El orden de ejecución de los *benchmarks* se ha establecido teniendo en cuenta el rendimiento de cada planificador en cada prueba. Por lo tanto, los hemos ordenado intentando que se produzca un cambio de planificador de uno a otro. El orden de ejecución es: IOR, LKR, 512k–SR, TAC y 8k–SR. De esta manera, también se muestra cómo DADS se adapta a los cambios en la carga de trabajo, cambiando, cuando sea necesario, el planificador.

Las pruebas se han ejecutado para 1, 2, 4, 8, 16 y 32 procesos. Puesto que varios de los *benchmarks* usan los mismos ficheros, hemos establecido que hasta 16 procesos, cada test, excepto LKR, use ficheros que no hayan sido usados por el test previo. Sin embargo, para 32 procesos, como sólo hay 32 ficheros, no es posible cumplir esta restricción.

Resultados

Para el núcleo DADS, se han probado dos configuraciones: CFQ–Deadline y Deadline–CFQ. CFQ–Deadline significa que, inicialmente, el disco real usa CFQ, *VD_RD* usa CFQ–VD y *VD_VD* utiliza Deadline. También implica que CFQ es el planificador por *defecto*, esto es, el planificador que se selecciona cuando se produce un gran número de cambios y el mecanismo de cambio de planificador se desactiva para no degradar el rendimiento del sistema (el mecanismo se vuelve a activar después cuando el número de cambios decrece). Para el núcleo original se han usado también dos configuraciones: una con CFQ y otra con Deadline.

Las figuras muestran como DADS se adapta al mejor planificador y la mejora que alcanza cada configuración sobre la peor configuración. En concreto, las figuras muestran:

$$\frac{T_{conf}}{Max(T_{CFQ-Deadline}, T_{Deadline-CFQ}, T_{CFQ}, T_{Deadline})}, \tag{0.7}$$

donde $T_{CFQ-Deadline}$, $T_{Deadline-CFQ}$, T_{CFQ} y $T_{Deadline}$ son los tiempos de aplicación para las dos configuraciones del núcleo DADS y para las dos del núcleo original, respectivamente, y T_{conf} es uno de esos cuatro tiempos de aplicación. Para facilitar el análisis de las figuras, hemos representado los resultados del núcleo original con líneas y los del núcleo DADS con barras.

Los resultados que se muestran son la media de cinco ejecuciones. Los intervalos de confianza, para un nivel de confianza del 95 %, también han sido calculados, siendo menores de un 5 % de la media. No obstante, para hacer más claras las figuras, los hemos omitido.

El ordenador se reinicia después de cada prueba, por lo que las pruebas se han realizado con la caché de páginas limpia. Cada vez que el sistema se reinicia, le damos al simulador unas tablas que han sido obtenidas mediante un entrenamiento *off–line*. La duración de la primera fase de la simulación se ha fijado en 5 segundos.

Usando el programa de captura, las cachés simuladas para los discos de prueba han sido configuradas con los siguientes valores:

- «HD–250–32»: su caché de 32 MB se divide en 63 segmentos y el tamaño del *read–ahead*, tanto para acceso secuencial como aleatorio, es de 256 sectores.
- «HD-320-16»: sus 16 MB de caché de disco se dividen en 64 segmentos. Cuando se detecta una acceso secuencial, el tamaño del *read–ahead* es de 256 sectores, y para un acceso no secuencial es de 96 sectores.
- «HD–500–16»: hemos considerado que su caché de 16 MB se divide en 20 segmentos. El tamaño de *read–ahead* para un acceso secuencial es de 256 sectores y para uno no secuencial, es de 32 sectores.
- «HD–500–32»: los 32 MB de caché se dividen en 128 segmentos. El tamaño del *read–ahead* tanto para acceso secuencial como aleatorio es de 256 sectores.

La figura 0.19 presenta los resultados para los cuatro discos de pruebas usados. En la figura, el primer histograma sobre el título «TOTAL» muestra el tiempo de aplicación total de la prueba, calculado como la suma de los tiempos de aplicación de los cinco *benchmarks*. Este primer histograma resume el comportamiento de nuestra propuesta durante la ejecución completa de la prueba. Los otros cinco histogramas muestran los resultados individuales para el tiempo de aplicación de cada *benchmark*.

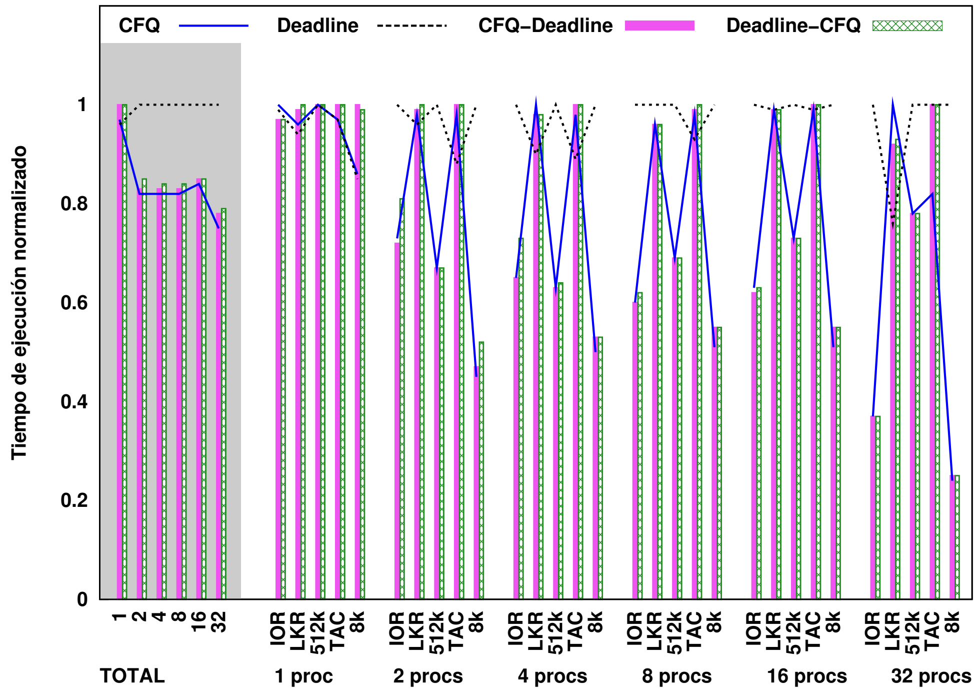
Tal como se puede observar, para un número dado de procesos, DADS sigue el mejor planificador, cambiándolo, si es necesario, cuando pasa de una prueba a la siguiente. La adaptación se puede ver más claramente en las figuras 0.19(a) y 0.19(d).

Es también importante resaltar que DADS obtiene mejores resultados que el mejor planificador en varios casos: para los discos «HD–500–16» y «HD–500–32», ver las figuras 0.19(c) and 0.19(d), nuestra propuesta mejora el rendimiento de CFQ en un 3,5 % para 32 procesos, siendo CFQ el planificador que obtiene el rendimiento más alto en ambos discos.

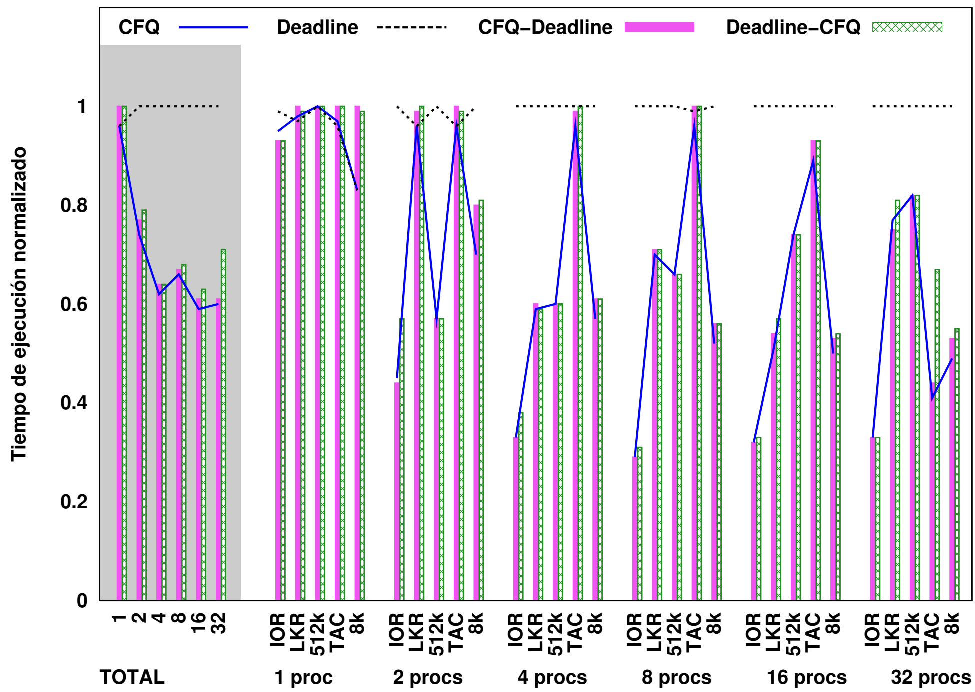
IOR. Para un patrón de acceso secuencial, DADS funciona como se esperaba y se adapta al mejor planificador. Sin embargo, para *Deadline–CFQ* y cualquier número de procesos excepto 32, hay una pequeña degradación con respecto al mejor planificador. El problema es que inicialmente se usa el peor planificador, que es *Deadline*. El cambio de planificador se realiza en la primera comprobación, pero al principio se pierde un tiempo, durante el primer intervalo, que después no es posible recuperar.

LKR. Con este test, nuestro mecanismo se adapta al planificador que presenta mejor rendimiento, aunque, en algunos casos, introduce una pequeña degradación con respecto al mejor. El planificador que proporciona el mejor comportamiento con esta prueba es distinto del planificador que presenta el mayor rendimiento con IOR (la prueba que se ejecuta justo antes), por lo tanto, inicialmente LKR tiene el peor planificador para su patrón de acceso. DADS detecta este hecho y realiza un cambio de planificador en la primera comprobación. Aún así, se produce un incremento en tiempo de E/S que perjudica el resultado final. Este problema aparece en todos los discos excepto en el modelo «HD–320–16».

512k–SR. El núcleo DADS presenta el mismo comportamiento que el mejor planificador, siendo sólo necesario destacar un caso. Con el disco «HD–500–16» y 32 procesos, DADS no es

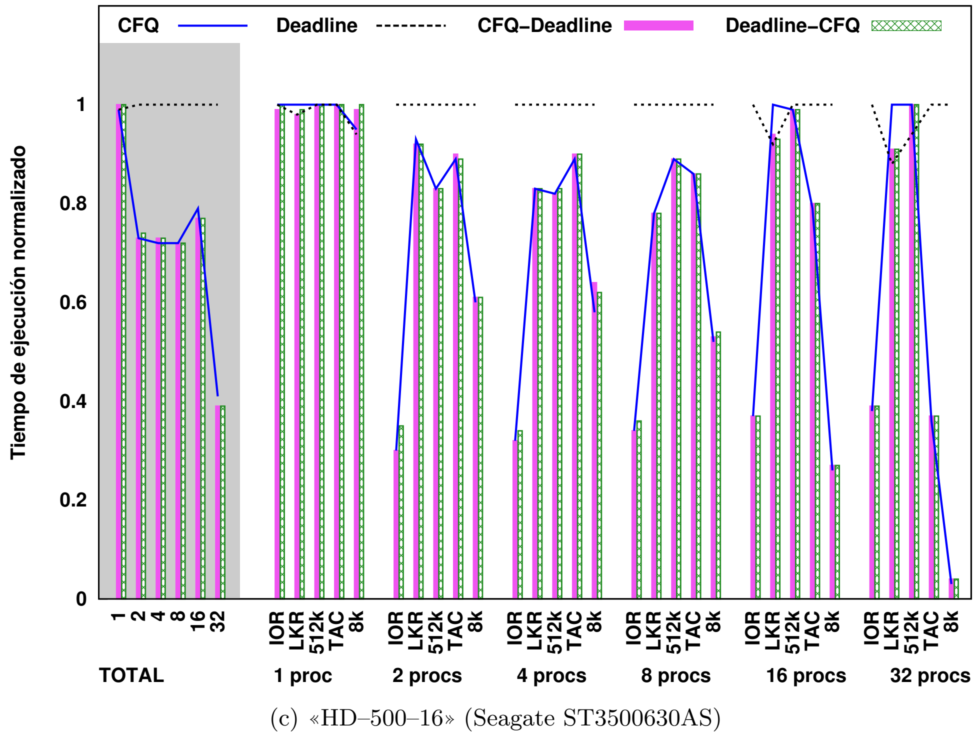


(a) «HD–250–32» (Seagate ST3250310NS)

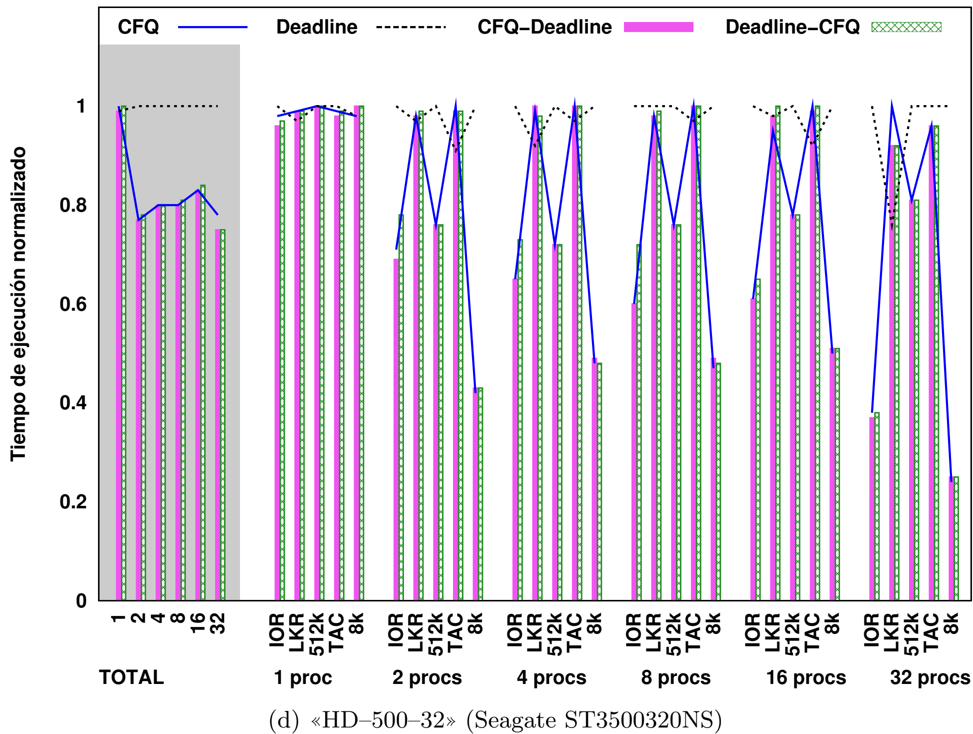


(b) «HD–320–16» (Samsung HD322HJ)

Figura 0.19: Resultados de DADS para las configuraciones *CFQ–Deadline* y *Deadline–CFQ*.



(c) «HD–500–16» (Seagate ST3500630AS)



(d) «HD–500–32» (Seagate ST3500320NS)

Figura 0.19: (Cont.) Resultados de DADS para las configuraciones *CFQ–Deadline* y *Deadline–CFQ*.

capaz de seleccionar el planificador Deadline, y pasa todo el tiempo con CFQ. El problema es que, en el núcleo original, la diferencia entre los tiempos de aplicación de ambos planificadores es menor de un 5,7 %, y el mecanismo propuesto no es capaz de detectar una diferencia tan pequeña.

TAC. Cuando se produce un patrón de acceso secuencial hacia atrás, el comportamiento de DADS depende del disco de pruebas que se use. Para el disco «HD–500–16», se selecciona el mejor planificador, y se alcanza el mismo rendimiento que el núcleo original. Sin embargo, para los otros tres discos, nuestro mecanismo no es capaz de hacer esta selección, e incrementa el tiempo de servicio con respecto al mejor resultado posible. Para cualquier número de procesos excepto 32, los planificadores obtienen casi el mismo rendimiento, y DADS no es capaz de detectar la pequeña diferencia que hay entre ellos. Incluso, introduce una pequeña sobrecarga que incrementa ligeramente el tiempo de aplicación, aunque el tiempo de E/S se mantiene igual. Para 32 procesos, DADS cambia alternativamente de un planificador a otro, y el mecanismo de control se activa para evitar que se produzcan un gran número de cambios seguidos. Por tanto, el rendimiento de cada configuración se aproxima al planificador por *defecto* de cada una.

8k–SR. DADS siempre selecciona el mejor planificador durante la ejecución de este test. Sin embargo, en algunos casos, se introduce una pequeña sobrecarga que incrementa ligeramente el tiempo de aplicación con respecto al núcleo original y el planificador que tiene el mejor rendimiento. Esta pequeña sobrecarga se nota más para 1 proceso, puesto que los tiempos de aplicación son más pequeños.

Un resultado inesperado en este test es que el núcleo DADS incrementa ligeramente el tiempo de E/S (y, por lo tanto, el tiempo de aplicación) con respecto al núcleo original cuando ambos usan el mismo planificador. La causa es la pequeña sobrecarga que se introduce al copiar una petición «real». Esta sobrecarga retrasa la llegada de la petición a la cola del planificador del disco real y, por tanto, al disco. El retraso es bastante pequeño, pero, sin embargo, lo suficientemente grande para hacer esperar al disco casi una vuelta completa, porque los sectores solicitados acaban de pasar y ya no pueden ser leídos. No obstante, es importante resaltar que este problema sólo afecta a las peticiones que son fallos de caché, y realizan un pequeño salto en sectores con respecto a la petición anterior. Este efecto también depende del modelo de disco, porque diferentes discos pueden tener una diferente distribución de sectores. Así, el problema no aparece en otros tests y es casi despreciable para los discos «HD–500-16» y «HD–500–32».

0.4.5. Conclusiones

DADS es una herramienta que automática y dinámicamente selecciona el mejor planificador de E/S de Linux para la carga de trabajo actual. La implementación presentada aquí evalúa el rendimiento de dos planificadores de E/S ejecutando, dentro del núcleo de Linux, dos instancias de un simulador de disco. Cada simulación calcula el tiempo de servicio empleado por su planificador para servir todas las peticiones. Al comparar estos tiempos de servicio, DADS determina si un cambio de planificador mejorará el rendimiento del sistema; en caso de que así sea, realiza un cambio de planificador en el disco real.

Además se ha modificado el simulador de disco propuesto en la sección 0.3 para que simule la caché de disco y los tiempos de pensar de cada petición.

Los resultados muestran que DADS selecciona el mejor planificador en cada momento, mejorando el rendimiento de la E/S. De hecho, gracias a usar DADS, los administradores de sistemas se librarán de elegir un planificador que no es óptimo y que, aunque para algunas cargas de trabajo puede proporcionar un buen rendimiento, para otras puede degradar el rendimiento del sistema de forma considerable.

0.5. Conclusiones y trabajo futuro

Vamos a concluir esta tesis haciendo un resumen de nuestras propuestas y proporcionando una breve visión de futuras direcciones de investigación. Nuestra motivación ha sido que un mejor rendimiento de E/S conllevará normalmente una mejora del rendimiento global del sistema. Las tres principales contribuciones realizadas para conseguir este objetivo han sido las siguientes.

Proyecto de caché de disco mejorada mediante RAM (REDCAP)

En primer lugar, la jerarquía de cachés se ha extendido introduciendo un nuevo nivel, la caché de REDCAP, entre la caché de páginas y la de disco. Una técnica de *prefetching* y un algoritmo para controlar el rendimiento alcanzado por la nueva caché completan esta primera propuesta. Usando una pequeña porción de memoria RAM, REDCAP puede reducir significativamente el tiempo de E/S de las peticiones de lectura, y también atenúa el problema del desalojo prematuro de bloques de la caché de páginas.

Para cargas de trabajo con algo de localidad espacial, nuestra técnica mejora el rendimiento hasta un 80 %. Para cargas de trabajo donde es difícil obtener una mejora en tiempo de E/S (principalmente, cargas de trabajo secuenciales y aleatorias), alcanza el mismo rendimiento que el obtenido por un sistema normal.

Debido a que REDCAP emula el comportamiento de una caché de disco (y se beneficia de su mecanismo de lectura anticipada), una conclusión que se puede extraer de nuestros resultados es que los discos deberían incluir cachés más grandes. Este aumento de tamaño mejoraría fácilmente el rendimiento del disco.

También hemos probado que el *prefetching* realizado tiene que ser dinámico, y es absolutamente necesario un mecanismo para activarlo/desactivarlo dependiendo de la mejora alcanzada. En otro caso, para algunas cargas de trabajo, un *prefetching* agresivo (sin ningún control) podría degradar significativamente el rendimiento de E/S y, consecuentemente, el rendimiento del sistema.

Simulador de disco dentro del núcleo

En segundo lugar, hemos implementado un simulador de disco dentro del núcleo de Linux que imita el comportamiento tanto de un disco tradicional como de un disco SSD. El disco se modela usando una tabla dinámica de tiempos de E/S direccionable por distancia de salto, tamaño de la petición y tipo de operación (lectura o escritura). La aproximación dinámica propuesta permite que nuestro modelo de disco se adapte rápidamente a los cambios que se produzcan en la carga de trabajo.

La versión inicial de nuestro simulador de disco controla el orden de llegada de las peticiones y las dependencias entre peticiones. También tiene un planificador de E/S que establece el orden en el que se envían las peticiones a disco. La segunda versión además tiene en cuenta el tiempo de pensar de las peticiones y simula una caché de disco.

El análisis de precisión realizado establece que para discos duros nuestro modelo presenta un buen comportamiento simulando el disco real de una manera bastante exacta, y que las diferencias entre ambos discos, real y virtual, se deben a la dificultad de simular el comportamiento de la caché de disco.

El simulador propuesto se puede usar para simulaciones *on–line* del rendimiento obtenido por diferentes mecanismos y algoritmos, y para dinámicamente activarlos o desactivarlos, o seleccionar entre diferentes configuraciones o políticas según los resultados. La primera versión se ha usado de forma satisfactoria para mejorar y simplificar el algoritmo de activación–desactivación de REDCAP. Es interesante destacar que, usando el disco virtual, REDCAP obtiene siempre las máximas mejoras posibles. La segunda versión ha sido utilizada en DADS para implementar un sistema de planificación dinámico.

Por lo tanto, podemos concluir que, a diferencia de otras propuestas teóricas, nuestro simulador de disco hace realidad un sistema de E/S que se auto–monitoriza y auto–adapta para obtener el mejor rendimiento.

Selección automática y dinámica del planificador de E/S

Nuestra tercera contribución es DADS, un mecanismo que realiza una comparación en tiempo real de dos planificadores distintos, y dinámicamente elige el planificador que obtiene el mejor rendimiento para la actual carga de trabajo. De forma simultánea ejecutamos una instancia de nuestro simulador de disco para cada planificador a comparar, y el planificador de E/S seleccionado es aquel cuya simulación proporciona el menor tiempo de servicio para la misma cantidad de datos solicitados.

DADS se adapta al mejor planificador en cualquier momento, y para discos duros, puede incluso superar el rendimiento de un sistema «normal», porque cambia el planificador para alcanzar el mayor rendimiento en cada momento.

Nuestro estudio confirma que no hay planificador de E/S que siempre proporcione el mejor rendimiento de E/S posible, puesto que el resultado depende de varios factores (carga de trabajo, dispositivo, etc.). Por lo tanto es necesario un mecanismo como DADS que sea capaz de seleccionar el mejor planificador en un momento dado. De hecho, usando DADS, los administradores de sistemas están exentos de elegir un planificador de E/S que no sea óptimo y que proporcione un buen comportamiento para algunas cargas de trabajo, pero que degrade el rendimiento del sistema cuando la carga de trabajo cambia.

Nos gustaría resaltar que tanto el simulador de disco como DADS han sido probados con discos SSD, obteniendo buenos resultados en ambos casos. Pero, debido a las limitaciones de espacio, esos resultados no han sido incluidos en este resumen.

Trabajo futuro

El trabajo presentado en esta disertación puede ser extendido en distintas direcciones puesto que varios puntos de investigación permanecen abiertos. Los siguientes son sólo unos pocos.

Dos cuestiones relacionadas con REDCAP merecen un análisis. Primeramente, su algoritmo de activación–desactivación podría ser mejorado para controlar el rendimiento obtenido en partes diferentes del disco, y no en todo el disco. Actualmente, el *prefetching* de REDCAP se activa o desactiva globalmente para un disco dado. Sin embargo, debido a que varios procesos pueden acceder al disco concurrentemente, la carga de disco resultante puede ser una mezcla de diferentes patrones de acceso. Por lo tanto, es posible que en algunas partes del disco REDCAP mejore el rendimiento, mientras que en otras partes no. El nuevo algoritmo podría analizar el rendimiento por grupo de segmentos de disco[5]. Dada una petición, decidiría si hacer *prefetching* o no dependiendo de la mejora ya alcanzada para el grupo correspondiente o grupos adyacentes. REDCAP estaría de este modo activada/desactivada por partes del disco.

En segundo lugar, debido a que REDCAP es capaz de obtener importantes mejoras para los discos, merece la pena investigar el desarrollo de REDCAP para sistemas RAID. En este caso, podría ser necesaria una nueva métrica para medir el rendimiento alcanzado porque un sistema RAID se presenta como un único disco aunque internamente el controlador maneja un grupo de discos.

En nuestro simulador de disco hemos modelado una caché de disco de una manera simple, porque hemos establecido un número fijo de segmentos. Sin embargo, muchas cachés tienen un comportamiento dinámico, y modifican el número de segmentos para mejorar el porcentaje de aciertos de caché [49]. Podríamos investigar cómo nuestro simulador podría capturar tal comportamiento dinámico, y calcular el número de segmentos de la caché analizando los patrones de las peticiones y los tiempos de E/S de las peticiones.

En este trabajo, DADS sólo compara dos a dos los planificadores de E/S de Linux. Queremos también extender DADS para que compare simultáneamente todos los planificadores disponibles. Además, un interesante aspecto es que todos los planificadores, excepto Noop, tiene varios parámetros configurables que se pueden modificar para asegurar un rendimiento óptimo. Pero, configurar los planificadores de manera manual para obtener mejor rendimiento de E/S no es una tarea sencilla. Por lo tanto, sería una buena idea que DADS seleccione no sólo el mejor planificador, sino también los mejores valores para los correspondientes parámetros.

Una continuación natural de este trabajo sería la evaluación de nuestras propuestas con los discos híbridos. En primer lugar, se debería probar si nuestro simulador de disco es capaz de simular el comportamiento de estos dispositivos. En segundo lugar, deberíamos evaluar el rendimiento que REDCAP podría alcanzar para los H–HDDs. En tercer lugar, se debería calcular el rendimiento de E/S de cada planificador bajo diferentes cargas de trabajo para estos discos, y analizar si DADS podría ayudar a elegir dinámicamente el mejor planificador.

Finalmente, otra dirección de investigación bastante prometedora es el diseño y la implementación de nuevos mecanismos, basados en nuestro simulador de disco, para mejorar el rendimiento de E/S. Los planificadores de E/S son una buena opción. Debido a que nuestro disco virtual también simula la caché de disco de un disco real, es viable implementar nuevos planificadores de E/S que tengan en cuenta los contenidos de la caché simulada para ordenar las peticiones en el disco real. Por ejemplo, un planificador podría servir una petición sin que sea su turno si los bloques solicitados por la petición van a ser expulsados de forma inmediata de la caché de disco.

[5] Recordar que REDCAP considera que el disco es una secuencia contigua de bloques, y que lo divide en segmentos del mismo tamaño que los segmentos de REDCAP.

Chapter 1

Introduction

Nowadays, disk drives are still the most widely used secondary storage devices, despite their throughput usually determines, to a large extent, the overall system performance. In this chapter, we discuss the problems that disk drives present, and why, in our opinion, hard disk drives will still be the dominant storage devices in the near future in spite of the significant and important improvements being currently made in storage technology. We then continue with a description of how, from our point of view, the I/O performance of these devices can be significantly improved. Finally, we summarize the main contributions of this thesis.

1.1. Background

Over the years, disk technology has advanced tremendously, and significant improvements have been achieved. However, memory and CPU performance has been improved at a much faster rate. As a consequence, disk system performance is a dominant factor in the overall system behavior, limiting the performance of many applications, specially of data–intensive applications. Hence, the disk I/O subsystem is usually identified as the mayor bottleneck for system performance in many computer systems.

1.1.1. Hard Disk Drives

Hard disk drives[1] are currently the most common secondary storage devices, despite their low performance. The problem is that a hard disk drive is a highly complex electro–mechanical system[2], and its mechanical operations considerably reduce its speed as compared to other components' [1, 2, 3].

A comparison of the components in a memory hierarchy illustrates these differences in performance. Table 1.1 shows the main attributes of different components by using data from 2005, while a comparison of cost per GB and access time between DRAM memory and hard disks, with data from 1980 to 2005, is plotted in Figure 1.1. As we can observe, hard disk drives present a limited effective bandwidth, and an extremely long latency. In bandwidth, there is a difference of two orders of magnitude between magnetic disk (10 MB/s) and RAM memory (2+ GB/s), whereas, for access times, it is of five orders of magnitude (10 ms for

[1]As hard disk drives we refer to disk drives that use technology of magnetic platters.

[2]Note that, in this thesis, we do not describe hard disk components or its operation. To get an overview on these topics, we refer the reader to Ruemmler and Wilkes [1], that give a detailed description of how a hard disk drive works, and to Jacob *et al.* [2], that provide a good view of disks, from physical recording principles to operation of disks, and even their evolution over the years.

Table 1.1: Attributes of memory hierarchy components. Source: "Modern Processor Design: Fundamentals of Superscalar Processors" [4].

				Cost ($) per	
Component	**Technology**	**Bandwith**	**Latency**	**Bit**	**Gigabyte**
Disk drive	Magnetic field	10+ MB/s	10 ms	$< 1 \times 10^{-9}$	< 1
SSD drive†	Flash memory	100+ MB/s	85 μs	$< 5 \times 10^{-9}$	< 5
Main memory	DRAM	2+ GB/s	50+ ns	$< 2 \times 10^{-7}$	< 200
On–chip L2 cache	SRAM	10+ GB/s	2+ ns	$< 1 \times 10^{-4}$	< 100k
On–chip L1 cache	SRAM	50+ GB/s	300+ ps	$> 1 \times 10^{-4}$	> 100k
Register File	Multiported SRAM	200+ GB/s	300+ ps	$> 1 \times 10^{-2}$ (?)	> 10M (?)

†The data about the SSD (*Solid–State Drive*) drive has been added to the original table by ourselves. The selected SSD device is an Intel SSDSA2MH160G2C1 SSD disk, with a capacity of 160 GB, from the year 2009 that we use in our experiments.

hard disks and 50+ ns for main memory). However, at the same time, hard disks provide the most cost–efficient storage, and the largest capacities of the compared technologies. The difference in cost is two orders of magnitude (less than \$1 per GB of disk storage against less than \$200 per GB of memory).

Today, these differences remain almost unaltered. The main attributes of a few randomly–chosen modern disk drives, with different technologies and currently found in the market, are summarized in Table 1.2. When comparing the first three hard disks [6, 7, 8] in this table with, for instance, a 4GB DDR3 desktop SDRAM module [9], we can conclude that:

- A fast hard disk has a maximum sustained bandwidth of 200 MB/s (which is achieved when the disk transfers data sequentially), whereas the bandwidth of the memory module is 10.6 GB/s.
- Disk latency is roughly 10 ms per random access, whereas RAM memory latency is 6 ns.
- Cost per GB is less than \$0.10 for hard disks, whereas around \$10 for the memory module.

As we have said, significant improvements have been achieved for hard disks. Current hard–drive technologies keep producing quite interesting proposals, such as perpendicular magnetic recording [16] or heat–assisted magnetic recording [17]. Manufacturers are also integrating larger caches and more intelligent controllers to their new products. However, all these improvements have a greater impact in disk density than in disk performance. This imbalance is illustrated in Figure 1.2 and Table 1.3. Figure 1.2 depicts disk technology trends over the years. Table 1.3 compares the main features of three hard disk drives from 2001, 2007 and 2011, of which the first two disks are used in our experiments (see Chapters 2 and 4).

Since the first hard disk was introduced in 1956, areal density of disk storage devices has increased dramatically with a growth rate that has varied from 25% to 100% [2, 18]. The evolutionary history of this improvement is summarized in Figure 1.2(a). Areal density consists of two components: recording density in the radial direction of a disk (numbers of recording tracks per inch), and recording density along a track (bits per inch). Although the number of tracks per inch has grown faster than the number of bits per inch, both have been

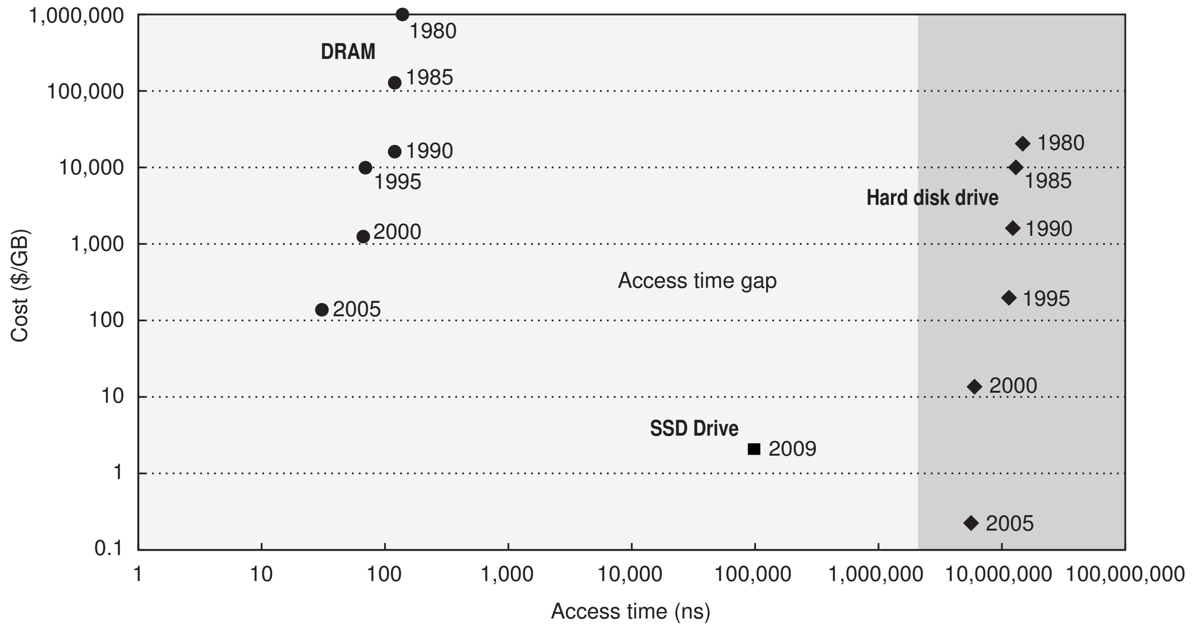


Figure 1.1: Cost versus access time for DRAM and hard disk in 1980, 1985, 1990, 1995, 2000, and 2005. Note that between 1990 and 2005 the cost per gigabyte DRAM chips made less improvement, while disk cost made dramatic improvement. Source: “Computer Architecture: A Quantitative Approach” [5]. The data about the SSD drive has been added to the original figure by ourselves, and it is the same as the one in Table 1.1.

significantly improved [18]. Because areal density determines the amount of data that can be stored on each platter, it dictates the total storage capacity of a disk. Therefore, the same incredible evolution has been done in disk capacity. For instance, Hitachi [19] introduced the first hard disk of 400 GB in 2004 [20], and Seagate [6] announced the first one of 750 GB in 2006 [21], while, in 2010, the highest disk capacity achieved was 3 TB [22, 23], and 4 TB, in 2011 [24, 25]. Note that, a current disk of 3 TB is 400% larger than the disk of 120 GB used in our first set of experiments (see Table 1.3).

Unfortunately, the values of the three major components of a disk's performance (seek time, rotational latency and transfer rate) have not been improved so significantly, and even, some of them remain almost identical for the last fifteen years.

Seek time has been decreasing over the years due to smaller and lighter drive components, specially shrinking disk diameters. However, since it mainly depends on the mechanical motion of the disk heads, reductions achieved are not significant in the last decade (see Figure 1.2(b) and Table 1.3), and an important change is not likely to occur in the near future. For today's desktop hard drives, the typical average seek time is still around 8 ms, which is quite similar to that of a 2001 disk, while it is around 4 ms for server hard drives [2].

Rotational latency has also been improved, but not significantly because it is inversely proportional to rotational speed. Note that, since the year 2000, no improvements have been made. Figure 1.2(c) shows this fact: during the last decade, average latencies have settled down to 2, 3, and 4.1 ms, which correspond to 15000, 10000, and 7200 RPM, respectively. Indeed, today's desktop hard drives usually spin at 7200 RPM, which was first introduced in server drives back in 1994, and is the spin speed as of the disks which appear in Table 1.3.

Table 1.2: Prices and capacities of disk drives [6, 7, 8, 10, 11, 12, 13, 14, 15].

Techno.	Model	Size	Cache size	Spin Speed	Transfer rate	Cost ($)
HDD	Seagate Barracuda XT® ST330005N1A1AS-RK	3 TB	64 MB	7200 RPM	138 MB/s (Max)	310
HDD	Hitachi Deskstar 7K3000 HDS723030ALA640	3 TB	64 MB	7200 RPM	207 MB/s (Max)	180
HDD	Western Digital Caviar Green WD30EZRX	3 TB	64 MB	IntelliPower	123 MB/s (Max)	250
HDD	Seagate Cheetah® NS 10k ST3600002SS	600 GB	16 MB	10000 RPM	82 MB/s (Min) 150 MB/s (Max)	530
HDD	Seagate Cheetah® NS 15k ST3600057SS	600 GB	16 MB	15000 RPM	122 MB/s (Min) 204 MB/s (Max)	670
SSD	Intel® SSD 710 Series	300 GB	–	–	270 MB/s (Read) 210 MB/s (Write)	1929
SSD	Samsung MZ–5PA256	256 GB	256 MB	–	250 MB/s (Read) 220 MB/s (Write)	370
SSD	Samsung MZ–7PC512D	512 GB	–	–	520 MB/s (Read) 400 MB/s (Write)	850
SSD	OCZSSD3-2VTX480G	480 GB	–	–	250 MB/s (Read) 215 MB/s (Write)	663
SSD	OCT1-25SAT3-512G	512 GB	512 MB	–	535 MB/s (Read) 400 MB/s (Write)	950
HDD + SSD	Seagate ® Momentus XT ST750LX003	750 GB 8 GB	32 MB	7200 RPM†	146.63 MB/s	240
HDD + SSD	OCZ RevoDrive Hybrid RVDHY-FH-1T	1 TB 100 GB	–	5400 RPM†	910 MB/s (Read) 810 MB/s (Write)	500

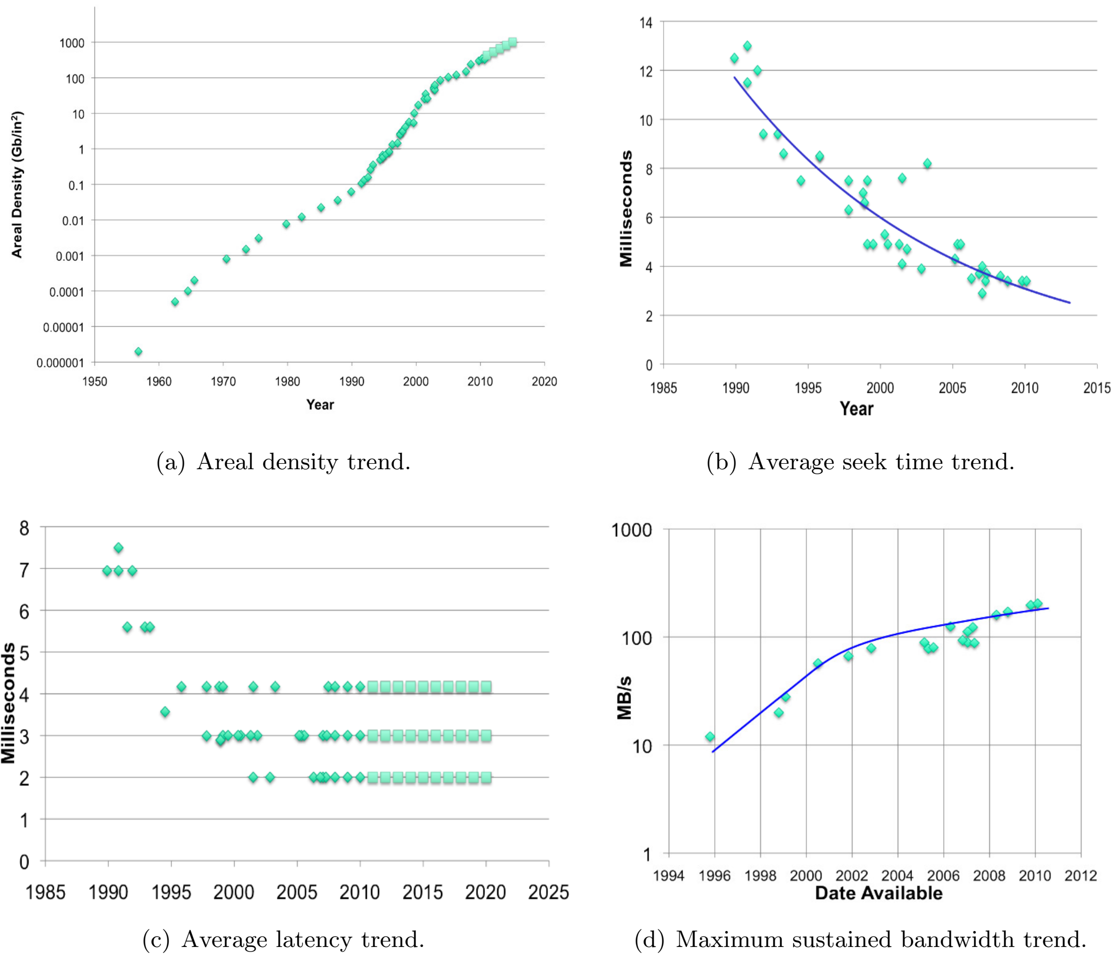
†This spin speed is only for the hard disk drive.

Server hard drives, on the other hand, run at 15000 RPM, with 10000 RPM being the most common [2].

As a consequence of increasing recording bit density, media transfer rate has also increased continuously and more quickly than seek time and rotational delay. This evolution is plotted in Figure 1.2(d), which shows that, since 1996, the bandwidth has increased from less than 10 MB/s to more than 200 MB/s. Hence, a current disk drive has multiplied by a factor of two the maximum sustained data transfer of a 2001 disk (see Table 1.3).

1.1.2. Solid State Drives

A recent real competitor to hard disk drives are Solid–State Drives (SSD), that have been tremendously enhanced during the last few years. Since SSD disks do not suffer the mechanical



(a) Areal density trend.

(b) Average seek time trend.

(c) Average latency trend.

(d) Maximum sustained bandwidth trend.

Figure 1.2: Hard disk technology trends. Source: "GPFS Scans 10 Billion Files in 43 Minutes" [18].

problems, they outperform *traditional* disks. But, their speed is still far away from the bandwidth achieved by RAM memory: the faster SSD disk that appears in Table 1.2 transfers at 535 MB/s, whereas a DRAM module can transfer at 10.6 GB/s.

SSDs also have several drawbacks. First, they are still expensive as compared to traditional disks. Although prices for SSDs are highly variable, and mainly depend on manufacturers (see Table 1.2), they cannot compete with the prices of traditional disks yet. Indeed, SSDs cost about $2 per GB, whereas magnetic disks cost less than $0.10 per GB. Second, hard drives provide larger capacity than SSDs. Currently, it is easy to find a hard disk of 3 TB, while SSDs are typically not larger than 64–256 GB (although there exist some expensive SSD drives of 2 TB [26]). Furthermore, it seems that this difference in disk capacity is going to remain for a while, since some authors point out that a great challenge for SSDs is to increase their capacity due to the size of the lithography used for making chips [27, 28]. Finally, another drawback is the lifetime of SSDs. The number of write cycles to any block of an SSD device is limited, because it must be erased before being re–written, and only a finite number of erasures are possible before read/write errors arise [29, 30]. Traditional devices do

Table 1.3: Comparison among 3 hard disk drives from 2001, 2007 and 2011 [8, 6, 7].

Model	Size	Cache size	Seek time	Spin Speed	Transfer rate	Year
WD Caviar WD1200BB	120 GB	2 MB	8.9 ms	7200 RPM	100 MB/s (Max)	2001
Seagate Barracuda ST3500630AS	500 GB	32 MB	< 8.5 ms	7200 RPM	100 MB/s (Max)	2007
Hitachi Deskstar 7K3000 HDS723030ALA640	3 TB	64 MB	8.2 ms	7200 RPM	207 MB/s (Max)	2011

not suffer this limit.

1.1.3. Hybrid Hard Disk Drives

Another interesting technology is that proposed for Hybrid Hard Disk Drive (H–HDD), which combines magnetic storage and SSD technologies. H–HDD drives are composed of an SSD and a hard disk drive. The former enhances the performance of the drive since smaller and more frequently accessed data is stored on it, while larger and less frequently used data is stored on the latter. Seagate Momentus XT (see Table 1.2) is an example of H–HDD for laptops, that uses a proprietary algorithm to monitor the drive activity and determine the optimum data to maintain in the SSD device [14]. OCZ RevoDrive Hybrid (see Table 1.2) is another example that also uses the SSD device as a dedicated cache. Again, an algorithm dynamically manages the use of both devices, and decides which data stays on each one [15].

H–HDD drives possesses a couple of interesting features. First, they are cheaper than SSDs, their price is around \$0.40 per GB. Second, depending on access patterns and data stored in the flash memory, H–HDDs can outperform traditional disks, although, under the worst scenario, they get the same performance as their internal hard disk drive.

1.2. Motivation

Despite the large imbalance between capacity and performance of hard disk drives, they are still the dominant storage device in many computer systems, and they will continue to be so for the foreseeable future [2]. From our point of view, SSDs are not going to replace hard disk drives any time soon, although they are going to coexist for a while. The new and emerging H–HDD devices also allow us to think that hard disk drives, or an enhanced version of them, will be in use for sometime. Moreover, most applications store and retrieve data on storage systems. Some of these applications are even characterized by huge storage requirements, since a large amount of data needs to be handled. Currently, hard disks are the only storage devices that can meet these huge capacity requirements at a reasonable cost and performance.

When the I/O time required by I/O-intensive applications is reduced, the execution time is also decreased. In the I/O subsystem, there exist several mechanisms, such as caches, prefetching techniques, and schedulers, that can greatly reduce the I/O time, and, consequently, improve the performance achieved. For instance, all hard disk drives have a built–in

cache (called *disk cache*) that improves the I/O performance of the device whenever an I/O request is served by the disk cache, and not by accessing the disk media.

A problem of the I/O mechanisms is that their performance depends on several aspects (workloads, file systems, disk models, technical limitations, and so on). For example, disk caches, built–in inside hard disks, have not been as effective as expected due to their small size compared to disk capacity. Indeed, a disk of 3 TB usually has only 64 MB of cache (see Table 1.2). Since some authors point out that larger disk caches could improve the I/O performance [31, 32], it would be a good a idea to propose a mechanism that effectively enlarges these disk caches.

In other cases, the behavior of an I/O mechanism can downgrade the performance for a given workload, or even there can be another mechanism that, for the same conditions, achieves a better throughput. Thus, to reduce the I/O time it would be interesting to activate/deactivate a mechanism as well as to dynamically change from one mechanism to another one depending on the current conditions. This would allow us to achieve a certain aim, such as increasing the bandwidth or reducing latency.

Bearing these facts in mind, we propose a disk simulator to dynamically and on–line simulate the behavior of a real disk under different conditions. Based on the results of this simulation, the behavior of a given I/O mechanism could be modified to improve its performance. Note that most of the mechanisms designed to enhance the I/O performance are generally implemented inside the kernel of the operating system. Hence, we have decided to also implement our disk simulator inside the kernel. Thereby, our simulator will be able to access to every kernel data structure or routine related to I/O mechanisms.

One important feature that we want for our on–line disk simulator is that it should not interfere with the regular working of the target system. Fortunately, today's computers have a quite large main memory and great computing capacity, which often remain underutilized. For instance, modern processors usually have several processing cores that, sometimes, are not totally exploited by applications. In our opinion, this environment provides an interesting platform for improving I/O performance, and these resources could be used for accelerating the I/O process. That is, due to multicore processors and their high computing power, our simulator can be implemented without degrading the system performance, or interfering with I/O requests submitted by applications.

1.3. Thesis Contributions

The improvement of the I/O performance is the goal of this PhD thesis. Our motivation is that the overall system performance will significantly profit from such an improvement. The contributions of our research are the following.

Our first proposal is to enlarge the disk cache of hard disk drives by using part of the main memory. The mechanism, that we call the **RAM Enhanced Disk Cache Project** (REDCAP), is a RAM–based cache that mimics the behavior of the disk cache with the purpose of reducing the read I/O time [33, 34]. By prefetching blocks adjacent to the requested disk blocks, our approach benefits from the read–ahead mechanism done by modern disk drives, and takes advantage of read requests issued by applications. This technique implements a control mechanism that activates or deactivates the new cache depending on the I/O performance achieved.

The second contribution of this thesis is a framework to compare, in real time, alternative approaches, and to activate/deactivate a mechanism, or to change from one mechanism or algorithm to another, depending on the expected performance. We present the design and implementation of a **disk simulator inside the Linux kernel** that is able to simulate any disk drive with a negligible overhead, and without interfering with regular I/O requests [35]. Our disk simulator takes into account possible dependencies among requests, thinking times (interarrival times of requests), I/O schedulers, and so on. This proposal allows us to achieve a dynamic behavior and to improve the overall system performance by simulating different I/O system mechanisms and algorithms at the same time, and dynamically turning them on and off, or selecting among different options or policies.

Our last contribution is a mechanism that selects the I/O scheduler that provides, for the current workload, the highest throughput. It is worth emphasizing that there is no an optimal I/O scheduler that always provides the best possible I/O performance for any workload [36]. For this reason, we propose a **Dynamic and Automatic Disk Scheduling framework** (DADS for short), that compares any two Linux I/O schedulers, and selects that achieving the best I/O performance for any workload at any time [37]. For this comparison, DADS uses our in–kernel disk simulator.

1.4. Thesis Organization

This thesis is organized as follows. Chapter 2 presents our first proposal, REDCAP, to improve the I/O performance of read operations. The design and implementation of the RAM–based disk cache is discussed by describing its cache, prefetching technique, and performance control. We also include a detailed analysis of its behavior under different variables.

Chapter 3 introduces the in–kernel disk simulator by describing the disk model, request management, I/O scheduler, and so on. We then explore the use of the disk simulator to tune our first approach; we also analyze the behavior of the modified REDCAP. The accuracy of the proposed disk model is evaluated in this chapter too.

Chapter 4 explores the dynamic I/O scheduling framework that we propose with DADS. We firstly discuss the modification of the in–kernel disk simulator that needs to be done for comparing I/O schedulers, and then describe the approach to achieve the automatic selection. In this chapter, an evaluation of our scheduler change mechanism is provided.

Finally, Chapter 5 concludes this thesis by briefly summarizing our findings and suggesting future directions in this field of research.

Chapter 2

REDCAP: The RAM Enhanced Disk Cache Project

The main subject of this chapter is the RAM Enhanced Disk Cache Project, REDCAP, a new cache of disk blocks that reduces the I/O time for reads. The REDCAP cache enlarges, by using a small portion of RAM memory, the built–in cache of disk drives, and prefetches consecutive disk blocks by taking advantage of read–ahead mechanism of disk drives. REDCAP is I/O–time efficient, and implements an *activation–deactivation algorithm* that dynamically turns the cache on/off depending on the improvement achieved.

We have implemented REDCAP in the Linux kernel, and analyzed its behavior under three different variables: i) segment size of its cache; ii) influence of the underlying file system; and iii) cache size.

The experimental results show that, with segments as large as the maximum request size allowed by the Linux operating system (128 kB), our mechanism reduces the application time by up to 80% for workloads which exhibit some spatial locality. Furthermore, also for this kind of workloads, it reduces the application time by more than 80% for file systems that split the disk into block groups, while for file systems that do not use this division the reduction is more than 55%. The experiments also show that the REDCAP cache size can determine the results depending on the file system and the number of processes. On the other hand, our proposal has the same performance as a traditional system for those workloads which have a random access pattern, or perform large sequential reads.

This chapter is organized as follows. We start by introducing the problem. Section 2.2 presents a first overview of our approach. The design and implementation of REDCAP is discussed in Section 2.3. The hardware platform, file systems, REDCAP cache configurations, and benchmarks used in the experiments are described in Section 2.4. A comparison of our results to those of a traditional system is performed in Section 2.5. Section 2.6 contains a brief description of previous work related to the proposed technique. Conclusions of this chapter are provided in Section 2.7.

2.1. Motivation

Nowadays, all disk drives have a built–in cache (called *disk cache*) that acts both as a speed–matching buffer and as a block cache [6, 38]. In all modern computers, this cache plays a crucial role in the I/O subsystem, because it reduces to a large extent the bottleneck that means the secondary storage in many systems due to its low performance as compared to other components, such as the CPU and the main memory [3].

The sequential access patterns are improved by prefetching data in the disk cache after a read request: the disk usually continues reading a sequentially numbered media blocks beyond

the end of a read operation. The goal is to minimize cache misses by anticipating future I/O data requests. The I/O performance is risen if a certain request can be satisfied "directly" from the disk cache, without seeking data and reading it off the disk, because accessing data from the cache is much faster than from the disk itself. The most common form of read caching in disk caches is read–ahead [1, 31, 49]. The read–ahead strategy improves the performance of a device, but its features extremely depend on the disk model. Indeed, aspects such as when is performed, where is began, how many blocks are read, or the type (adaptive or aggressive), vary significantly from one model to another, even within the same manufacturer.

On the other hand, write operations are also affected as data is written first to the disk cache and then to the disk itself. Indeed, depending on the write–to–disk policy, the disk cache could also influence the behavior of an application. For instance, a write–back policy used in conjunction with immediate reporting will not block the application on I/O operations because the request is considered "done" as soon as it is in the cache (but not necessarily in disk media). However, when a write–through policy is used, the cache and disk drive are updated at the same time, and an application has to wait for write operations, because they are not finished until data is on the disk media. Thus, the application will continue running sooner with a write–back policy than with a write–through one. Hence, a write–back policy improves the I/O performance because frees applications from waiting on data to reach the disk surface, and allows the disk controller to write data in an order that reduces the movement of the disk heads. We remark that both a write–back policy and an immediate reporting are normally used in disk caches [49].

The disk cache is usually split into separate segments of small size to allow prefetching on multiple sequential streams [1, 31, 49]. Since segments hold contiguous disk data, each I/O stream can be treated as having its own cache by assigning it to a segment. Moreover, the performance of the I/O subsystem can be efficiently improved when the number of concurrent streams is smaller than the number of cache segments. In contrast, if the number of streams exceeds the number of segments, a problem arises and a replacement algorithm is needed to determine which one should be evicted. There are several algorithms, such as LRU (Least Recently Used), FIFO (First In First Out) or Round–Robin, that can be used for determining the data to evict. Nevertheless, this information is considered trade secrets and the details about the used algorithm are hard to find or even unknown. It is interesting to mention that both Karelda [31] and Shriver [49] consider the LRU replacement algorithm as the most used in disk caches.

The size of a cache has a significant effect on its performance because it determines the hit rate [39], and it is one of the most important aspects in the design of a disk cache. System designers generally consider that a disk cache should be 0.1 to 0.3 percent of the total disk space [31], and Hsu and Smith [32] also suggest that sizes up to, and even larger than, 1% of the storage capacity would improve the I/O performance. Though the manufactures tend to integrate larger caches, their sizes are still in the range of 2 to 64 MB [40, 7], which is rather small compared to the disk capacity. For example, a disk of 1 TB usually has 32 MB of cache [6, 38], i.e., only 0.003 % of the total disk space. The current technology for hard disks also indicates that the imbalance between these two sizes will not change in the short term [6, 38]. The main reasons are a tradeoff between cache size and cost (large caches are expensive), and space limitations. As a consequence, their small sizes have caused disk caches to have not been as effective as expected.

Since computing systems usually have a large main memory, and it is expected that the larger the disk cache, the better the performance achieved; the potential benefits of using a small part of the main memory to enlarge the disk cache should be considered and investigated. Even more, it is reasonable to think that this could improve the access time of disk drives without downgrading the overall performance of the computing system. Thus, motivated by these ideas, we propose the RAM Enhanced Disk Cache Project, REDCAP: a new cache of disk blocks in RAM memory whose aim is to reduce the I/O time of read requests.

2.2. REDCAP overview

The essential ideas behind REDCAP are to extend the disk cache, to emulate its behavior with the purpose of improving the I/O time, specially the read I/O time, and to take advantage of its read–ahead mechanism by prefetching some disk blocks.

The extension of the disk cache is performed by introducing a new level in the cache hierarchy, referred to as *REDCAP cache*, between the page and disk caches. This new cache is stored in main memory. Figure 2.1 shows the hierarchy proposed in this work.

In order to emulate the behavior of the disk cache and to make use of its prefetching, our technique also prefetches consecutive disk blocks that could be served to a subsequent read I/O request later. Each time data requested by a read operation is not found in the REDCAP cache, new data is prefetched from disk to the cache. Thus, a cache miss causes blocks, that are contiguous to the requested ones, to be read into the new cache from disk. A cache hit read operation, however, does not perform any read, so the disk is not accessed and the requested data is copied from the new cache. I/O performance will be improved whenever a read operation can be satisfied from the REDCAP cache, without sending it to disk.

On the other hand, a write request can also modify the REDCAP cache because a cache–hit write operation updates the cache, whereas a miss does not modify it. It is interesting to remark that the proposed mechanism does not take part in writes: it only invalidates the appropriated disk blocks in the REDCAP cache in case of a hit, and sends write requests directly to disk without any modification. We have decided not to copy the affected disk blocks, because we do not expect any access to these blocks in the short term. Realize that, in Linux, write operations are deferred in the page cache, and submitted to disk by the `pdflush` thread under three different situations [51, 42, 52]. Firstly, dirty blocks are written to synchronize the page cache with the disk; after synchronization, blocks remain in the page cache from where they can be read again. Secondly, dirty blocks are written to regain memory when the amount of free memory is low; therefore it has no sense to make a copy of these blocks because if the page cache is saturated, the REDCAP cache will also be saturated, since the former is much larger than the latter. Finally, dirty blocks are written because they will be used no longer; therefore these blocks are unlikely to be requested again soon.

Obviously, due to the prefetching performed by REDCAP, the amount of blocks read is increased, and consequently the I/O time is also increased. Fortunately, the increase in IO time is not proportional to the number of blocks because, if we combine several small requests into a large request, seek times and rotational delays, which are the dominant factors in the IO time, are reduced, and a best I/O performance is usually achieved. For instance, to read 128 kB, a single request takes less I/O time than 2 requests of 64 kB each, or than 32 requests of 4 kB each. However, if part of the prefetched blocks are used by future reads, this increase

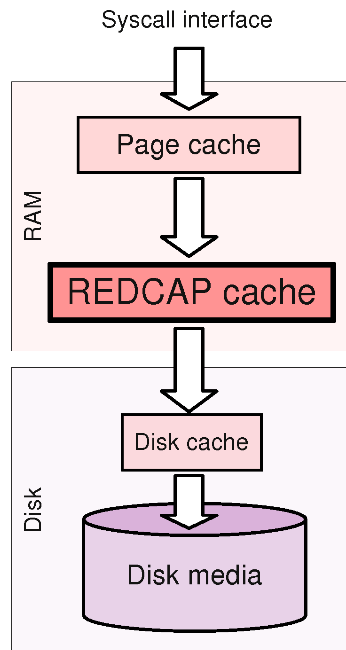


Figure 2.1: REDCAP cache hierarchy.

in blocks and time will be compensated, because many small requests will be replaced by a few large sequential reads that will take less time altogether. Indeed, the overall access time will be smaller than in a normal system without REDCAP. In contrast, if prefetched data is not used, this increase in the I/O time will not be compensated. For this reason, our technique also implements an *activation–deactivation algorithm* that continuously studies the evolution of its cache. If the REDCAP cache is not effective, and the performance is getting worse, the algorithm turns the cache off, prefetching is stopped, and only data requested by applications is read from disk. When the REDCAP cache is deactivated, the algorithm simulates its operation and studies the possible success of the REDCAP prefetching. When the algorithm detects a cache improvement and that the prefetching performed can be taken advantage of, it will turn the cache on again.

As we can see, two of the REDCAP's aims are to overlap computation and I/O requests, and to be I/O–time efficient. Hence, REDCAP only prefetches data on cache misses, always reads consecutive disk blocks, and converts workloads with thousands of small requests into workloads with disk–optimal large sequential requests [33, 34]. Let us remark that the prefetching implemented by many operating systems, and other prefetching techniques based on file access patterns [53], perform read requests of blocks that may not be consecutive on disk, increasing the overall I/O time.

Note that the enlargement of the *page cache* provided by the operating system could also be considered as a way to obtain a result similar to that provided by REDCAP. However, our proposal is based on a prefetching policy that is completely different from that performed by the page cache. REDCAP prefetches blocks that are adjacent on disk, while the page cache usually reads in advance data blocks of the same file, that could be fragmented on disk. In exploiting the principle of spatial locality (if a block is referenced, then nearby blocks may also

be accessed soon), our mechanism also takes advantage of the organization in block groups of some file systems, where data blocks and i–nodes of all the regular files in a directory are put together in the same group assigned to the directory (or in nearby groups if the corresponding group is full) [41]. As our results show, even with a small portion of main memory, REDCAP is able to obtain a performance that is much better than that obtained by the usual policies of the page cache.

2.3. Design and Implementation

Here, we discuss the design and implementation of REDCAP which consists of three parts:

1. a *cache*, that works as an extension in the main memory of the disk cache;
2. a *prefetching technique* to manage the cache;
3. and an *activation–deactivation algorithm* to control the performance achieved.

In the following sections, these components are described in detail.

2.3.1. REDCAP cache

As mentioned previously, one of the most important parts of REDCAP is its cache which represents a new level in the cache hierarchy, just between the page cache and the disk cache (see Figure 2.1). The aim of the REDCAP cache is to extend the disk cache in RAM memory and to emulate its behavior. It can be considered as a second cache of the hard disk in main memory.

The REDCAP cache is a cache of disk blocks prefetched by read requests on every cache miss. It is stored in RAM and has a fixed size of C blocks. Analogously to a disk cache, this new cache is split into N segments that are managed independently of each other and have a size of S blocks (where $C = N \times S$). The number of segments is fixed and we have not considered a dynamic division of the cache. The segments hold contiguous disk data, so a REDCAP segment has a group of consecutive disk blocks and it is the transfer unit used by the prefetching technique. Thus, data is allocated in the REDCAP cache in terms of cache segments. Section 2.5 investigates the impact of varying the segment size and the cache size in the REDCAP performance.

Once the cache is full, i.e., all the segments are in use, some data must be deleted whenever new data has to be added, and a cache replacement algorithm is required to make this task. The REDCAP cache uses a Least Recently Used (LRU) replacement algorithm to determine the segment that should be freed. This algorithm is one of the most popular replacement policies, and is the most used in disk caches [31, 49]. We have decided to use this algorithm because its implementation is straightforward and it generally obtains good results. Let us remark that we are aware that other algorithms, such as Round Robin, Least Frequently Used and Segmented LRU [31], could perform better for certain workloads.

Since modern drives tend to be optimized for sequential access and hide their physical geometry to operating system, it is rather difficult to know both the exact layout of the blocks on disk and the operation of the disk cache. Hence, REDCAP considers the disk as a contiguous sequence of blocks, referenced by their logical block numbers. In addition, it splits the disk into segments of the same size as the REDCAP ones: the first disk segment begins at disk block 0 and finishes at disk block $S - 1$, the second one is from S to $2S - 1$,

and so on. Every REDCAP segment corresponds to one disk segment, i.e., S consecutive disk blocks. In this way, the alignment of REDCAP segments with disk segments makes it easier and more efficient to search the requested blocks in the cache. For each I/O request coming in, it is straightforward to know whether the appropriate disk segment is in cache or not.

REDCAP uses several page frames of the main memory to store segments. These page frames are marked as reserved to prevent the operating system from swapping them out. Since the size of both page frames and logical blocks of the underlying file system are the same, 4 kB, every disk block prefetched will be in one page frame.

2.3.2. Prefetching technique

The second part of our proposal is the prefetching technique that decides the data to be read from the disk to the cache. The prefetching technique implemented could be considered as a variant of the read–ahead of the disk cache. Prefetching is performed only when a read operation takes places and a cache miss occurs, whereas it is not done on a cache hit or during write requests. The prefetching technique itself is quite simple yet effective, and it is applicable to any operating system, any file system and even any storage device.

In a normal system, when an I/O request arrives to the block device layer, it is inserted into the request queue of the I/O scheduler, and from there it will be dispatched to disk. In a REDCAP system, each request will be managed differently depending on its type. A read request is not inserted into the I/O scheduler, but the requested blocks are searched in the REDCAP cache. Two different options appear: i) if the request is a cache hit, blocks are copied from the cache, and ii) if it is a cache miss, the requested and prefetched blocks will be read from disk. In contrast, a write operation invalidates cache's blocks when necessary, and the request is inserted into the scheduler queue without any modification. Now, we explain in detail the prefetching technique for each operation type.

Read Operation

For each read I/O request issued, REDCAP first calculates the amount of affected disk segments and then searches those in its cache. Depending on the size of both requests and segments, it could be that the read operation affects not only one segment but several of them. In the latter case, all the affected segments will be contiguous on disk, and REDCAP manages the request as if it was n small partial requests, one for each disk segment. The new small partial read operations are handled as a regular ones.

Let us first describe the simple case in which a read I/O request only affects one disk segment. A scheme of this management is presented in Figure 2.2. The generalization to the case of a request affecting several segments will be provided later.

If the desired disk blocks are found in a REDCAP segment, a cache hit occurs, and the request is serviced from the cache by copying the requested data. No read is performed, and therefore, no prefetching is performed either. Thus, data is copied from page frames of our cache into page frames of the I/O request and then the operation is ended. For the operation, the result is the same but without accessing to disk. Applications do not know if data comes from the REDCAP cache or if the operation was sent to disk. Since disk has not been accessed, I/O time is close to the memory copy time.

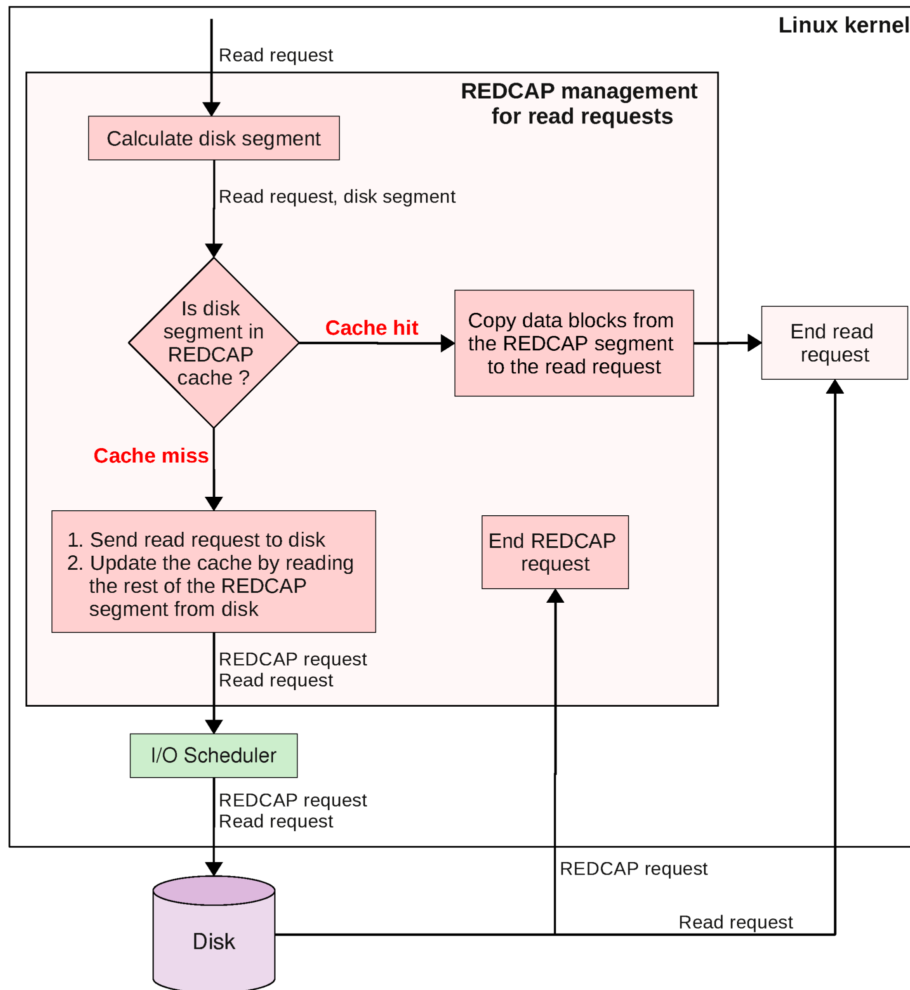


Figure 2.2: REDCAP management for a read request.

The other possibility is a cache miss that occurs when data is not in the REDCAP cache. This means that the requested disk segment is not in any REDCAP segment. In this case, all the blocks of the corresponding disk segment will be read from disk. It is important to remark that some blocks will be those requested by the original read operation, while others are read to complete the segment and will be the prefetched blocks. As a consequence, the amount of data to prefetch always depends on the size of the request and of the REDCAP segments. Note that, if the original read operation matches a whole REDCAP segment, no data will be prefetched. Now, for a cache miss, the I/O time is a function of the disk access time.

In order to avoid delays on the original read operation, REDCAP sends several independent requests to disk: the original one followed by one or more requests to complete the corresponding segment. The amount of requests needed depends on the number of blocks to prefetch and the position of the original request in the disk segment. Figure 2.3 illustrates the four possibilities that could be encountered. If the original request is exactly at

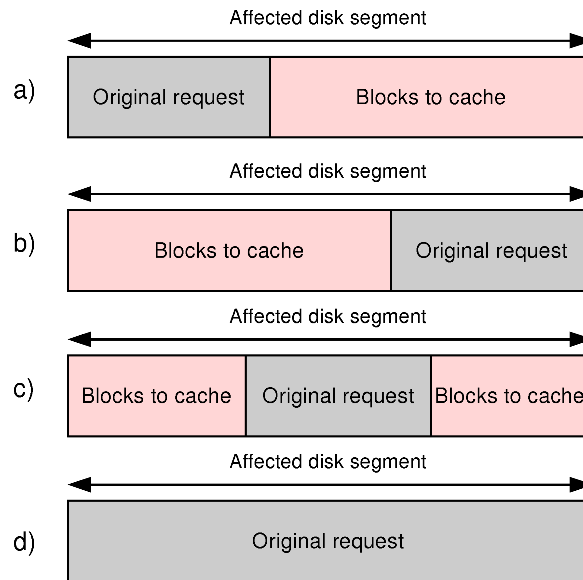


Figure 2.3: An affected disk segment and its possible division into original and prefetched requests. (a) A request beginning at a disk segment. (b) A request ending at a disk segment. (c) A request just in the middle of a disk segment. (d) A request matching a whole disk segment.

the beginning or end of the disk segment, only one request will be necessary to prefetch the blocks. Figures 2.3(a) and (b) depict these cases. The third possibility appears if the original request is in the middle of a segment, as it is illustrated in Figure 2.3(c). There is a fourth possibility that appears when the original request matches a whole segment, and the request and the segment are aligned. In this case, since no prefetching is performed, just a single request is needed, as we can see in Figure 2.3(d), and no segment really is used. Obviously, if the resulting requests are greater than the maximum request size allowed by the operating system, several smaller requests will be sent to disk.

Note that if the scheduler queue is empty, the original request can be immediately dispatched to disk. However, if the queue is not empty, since all the requests, original and prefetched, are contiguous in disk, the scheduler will merge all of them in a single request, which will be eventually issued to disk.

It is interesting to remark that, in order to achieve (at least partially) *exclusive caching* [54] between the operating system caches and the REDCAP cache, blocks requested by the original read operation are not stored in our cache, because they will be in the page cache. This is specially true for requests that affect whole segments. The idea is not to have duplicated disk blocks in both caches unnecessarily. Also note that the page cache is much larger than the REDCAP cache, and that blocks in the latter will be likely evicted before than those in the former.

For the more general case, a read operation including several disk segments is handled as n small partial requests, one for each disk segment. Since an I/O operation only requests consecutive blocks, the affected disk segments are also consecutive. All the new partial requests are managed in a similar way: calculating the affected disk segment, looking for it

in the cache, and copying the data on a cache hit or reading the data and the prefetching blocks on a miss. REDCAP controls the completion of the partial requests, and when all of them finish, the original one ends.

If an original read request spans two disk segments, there are four possibilities: two cache hits, two cache misses, a cache hit and a cache miss, or a cache miss and a cache hit. The number of possibilities increases as 2^n, with n being the number of affected disk segments. Nevertheless, all the inside partial requests affecting whole segments will usually be cache misses, whereas the partial requests at the ends could be cache hits or misses. The reason why the inside partial requests are cache misses is because the page cache is larger than the REDCAP cache, and if the blocks corresponding to these partial requests are not in the page cache, neither will be in the REDCAP cache. Furthermore, if some of these blocks have been read and already evicted from the page cache, the corresponding segment will also have been evicted from the REDCAP cache.

It is worth mentioning that when all the cache segments are busy attending ongoing requests, subsequent requests that cause cache misses are sent to disk directly without cache intervention. This procedure prevents REDCAP becoming a bottleneck when there are lost of processes and, therefore, more requests than segments.

Since the REDCAP cache stores data which is contiguous to the requested data, our prefetching technique exploits the spatial locality: if a block is referenced, then nearby blocks will also be accessed soon.

We remark that the prefetching performed by REDCAP is not carried out by operating systems, which usually only perform file prefetching, but not metadata prefetching or a limited one [51]. In contrast, part of the success of REDCAP is the prefetching performed with the metadata blocks. On the other hand, applications do not usually perform any type of prefetching, and they should be modified, something that it is not always possible, to perform any.

To summarize, prefetching is performed only when a read operation is a cache miss. The amount of data to prefetch depends on the size of both requests and REDCAP segments. Prefetching is not done on a cache hit.

Write Operation

As already indicated, for write requests, REDCAP neither performs any prefetching nor modifies requests, but it updates its own cache.

Therefore, in a REDCAP system, write operations are straightforward, the affected disk segments are calculated, and searched in the cache. In case of a hit, the corresponding segment is updated by invalidating the appropriated blocks (but not the whole segment). On a miss, no updates are made. Finally, the original request, without modifications, is issued to disk by queueing it in the I/O scheduler. The management scheme performed for writes is shown in Figure 2.4.

The idea of invalidating the data blocks being written to disk is based on the fact that they are already in the page cache, so it is highly unlikely that they are requested again from the REDCAP cache. Hence, it is not necessary to make a copy of those blocks. Furthermore, although our cache would make a copy, those blocks would have to be immediately written down to disk in order not to put the consistency of the file system at risk, for example.

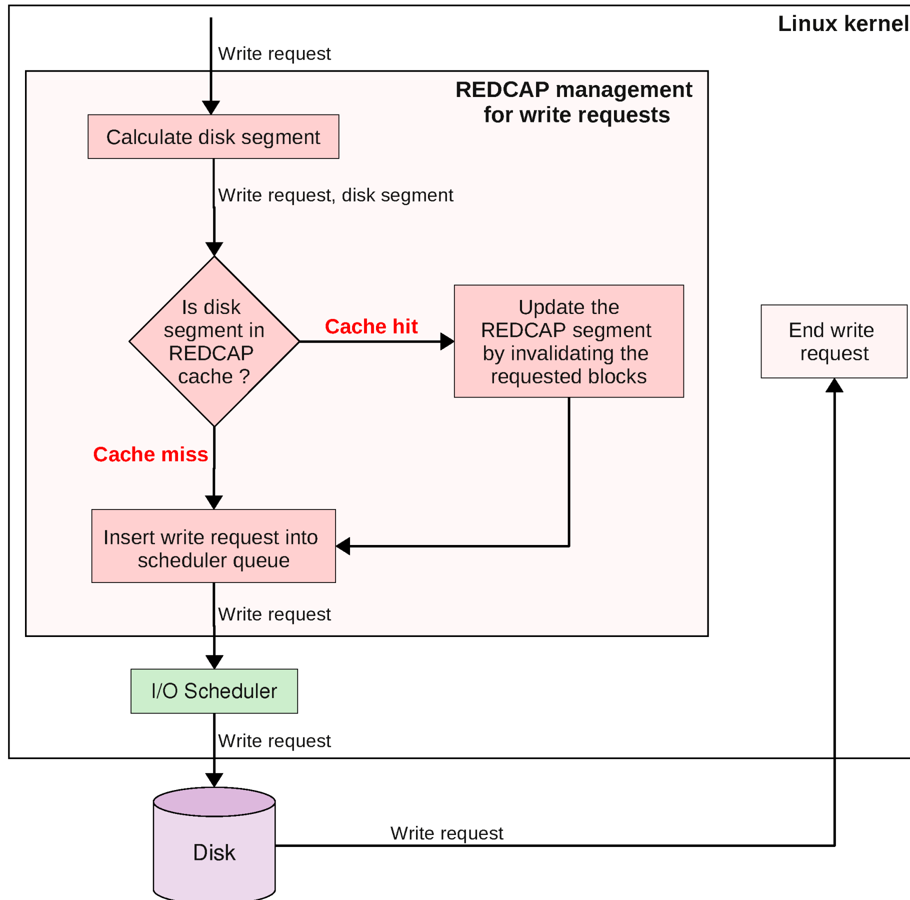


Figure 2.4: REDCAP management for write requests.

Note that if a read operation requests some blocks of a segment that have been invalidated by a write operation, only the invalidated blocks will be read from disk, and the other blocks will be copied from the segment.

2.3.3. The activation–deactivation algorithm

As aforementioned, REDCAP is completed with an *activation–deactivation algorithm*. Its duties are to control the performance obtained by its cache and to make the prefetching dynamic.

The REDCAP cache improves the I/O performance when it can frequently provide requested data without accessing the disk. In the worst case, when prefetched data is not used, the performance considerably decreases, because the amount of blocks prefetched by REDCAP means an increase in I/O time not compensated by cache hits. In order to avoid these cases and to control and improve the REDCAP performance, we have implemented an activation–deactivation algorithm [33, 55]. This algorithm continually analyzes the performance of REDCAP by comparing the time needed by its cache to process the requests with the estimated time to process them without it. The algorithm turns our cache on/off

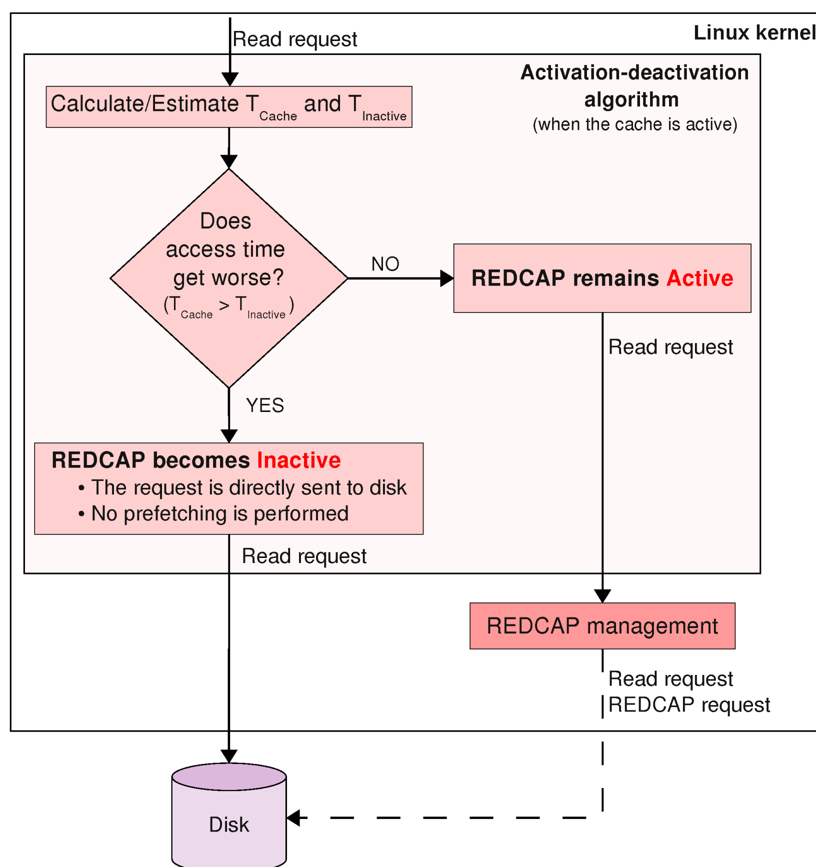


Figure 2.5: The activation–deactivation algorithm.

according to the obtained results. Operations forming the activation–deactivation algorithm are schematically presented in Figure 2.5.

In order to control the performance, REDCAP works in two main states: *active* and *inactive*. In the active state, the read requests are dispatched as it was previously explained, and the algorithm studies the performance achieved. If the algorithm detects that the access time is getting larger with than without cache, REDCAP will be turned off and moved to the inactive state. In this state, requests are not processed and no prefetching is performed, but the algorithm continues studying the possible success of REDCAP. The system simulates the tasks done by our cache as it would be switched on, and records the hits and misses for each read request. If the algorithm detects that our cache can be efficient, moves it to the active state again. In the next sections, we describe both states in detail, and how the algorithm works.

Active state

Let us first describe the operation of the algorithm in the active state. The explanation for the inactive case will be given later.

A metric is needed to measure the cost of keeping the REDCAP cache active, and therefore,

to determine the performance achieved by its prefetching. In the active state, the algorithm uses a quite simple mechanism based on disk access times for cache misses and copy times for cache hits. The algorithm calculates the time to serve a read request from its cache on a hit, and the time to read the request and prefetched data from disk on a miss. Furthermore, it also analyzes the performance of a "normal" system without REDCAP by estimating the time to serve each read request directly from disk. By comparing these two values, the proper state of REDCAP is decided, and changed when necessary.

For each read request, the following data is calculated and stored:

- In a cache hit:
  - B_{CHit} denotes the number of disk blocks that are a cache hit, i.e., blocks that are copied from the REDCAP cache to the original read request.
  - T_{CHit} denotes the time needed to attend the cache hit, i.e, to copy B_{CHit} blocks.
  - T_{Wait} is the time waiting for blocks to be read from disk. If the B_{CHit} requested disk blocks have already been asked to disk by a previous prefetching request of REDCAP, but they have not arrived yet, the system have to wait for them. Note that this time is almost always zero, because prefetched data normally has arrived yet.
- In a cache miss:
  - B_{CMiss} denotes the number of disk blocks of the original request asked to disk because they were not in the cache.
  - T_{CMiss} is the time needed to read B_{CMiss} blocks from disk[1].
  - $B_{Prefetched}$ denotes the number of disk blocks that are prefetched for this cache miss.
  - $T_{Prefetched}$ denotes the time needed to read $B_{Prefetched}$ blocks from disk.

When the cache is active, the time of each request is:

$$T_{Active} = T_{CHit} + T_{Wait} + T_{CMiss} + T_{Prefetched} \quad (2.1)$$

where one or more of these contributions could be zero depending on the cache hits and/or misses, and on the number of segments affected by the read request.

For each read request, the time required by the REDCAP cache is given by:

$$T_{Cache} = T_{CHit} + T_{Wait} + T_{Prefetched}. \quad (2.2)$$

Thus, the time needed by a request, given by Eq. (2.1), can also be expressed as:

$$T_{Active} = T_{Cache} + T_{CMiss}. \quad (2.3)$$

On the other case, in a system without REDCAP, and, therefore, with an inactive REDCAP cache, each read request would have to be read directly from disk, and the time of each request would be:

$$T_{Inactive} = T_{Disk_CHit} + T_{CMiss} \quad (2.4)$$

[1]For the sake of simplicity, the names of the variables have been changed with respect to the original publication [33], where B_{CHit}, T_{CHit}, B_{CMiss}, T_{CMiss}, and $T_{Disk_Estimated}$ were previously denoted as B_{Copy}, T_{Copy}, $B_{Disk_Original}$, $T_{Disk_Original}$, and T_{Disk_CHit}, respectively.

where T_{Disk_CHit} denotes the estimation of the time needed to read the B_{CHit} blocks from disk. Again, note that one of these terms could be zero depending on the cache hits and/or misses. The algorithm uses disk access times, calculated by REDCAP while cache miss requests are served, to estimate T_{Disk_CHit}, and its value is given by:

$$T_{Disk_CHit} = \frac{(T_{CMiss} + T_{CMiss_Inactive})B_{CHit}}{(B_{CMiss} + B_{CMiss_Inactive})} \tag{2.5}$$

where $T_{CMiss_Inactive}$ and $B_{CMiss_Inactive}$ are, respectively, the latest values of T_{CMiss} and B_{CMiss} calculated when REDCAP was last inactive. In order to reduce their importance and influence, these values are divided by 2 at every check. If the cache is active for a long time it holds that $B_{CMiss_Inactive} \approx 0$, and after several checks, the previous Equation (2.5) could be consider as equivalent to the following one:

$$T_{Disk_CHit} = \frac{T_{CMiss}B_{CHit}}{B_{CMiss}}. \tag{2.6}$$

Once we know how to calculate the time to serve each request on a REDCAP system and a normal system, these two times can be compared. The algorithm says that if the time to serve a request with REDCAP is less or equal to the time to serve it directly from disk, REDCAP is being effective. Using the previous expressions, the algorithm says that if the condition

$$T_{Active} \leq T_{Inactive} \tag{2.7}$$

is satisfied, REDCAP is improving access time and it should be active, otherwise, it has to be inactive. By replacing expressions (2.3) and (2.4) in Inequality (2.7) and simplifying it, we arrive at

$$T_{Cache} \leq T_{Disk_CHit}. \tag{2.8}$$

By using now the expression for T_{Cache} (2.2), it yields

$$T_{CHit} + T_{Wait} + T_{Prefetched} \leq T_{Disk_CHit}. \tag{2.9}$$

Hence, if the total time needed by the cache, i.e., the time needed to copy the disk blocks from the cache to the original read request, plus the time waiting on a segment to be read from disk, plus the time required to prefetch segments, is less than or equal to the estimated time to read from disk the blocks that are cache hits (B_{CHit}), REDCAP is effective, and its cache has to be active. Otherwise, REDCAP has to be turned off because its cache is degrading the performance with respect to a system without it.

Inactive state

Now, let's assume that REDCAP is inactive. In this case, each request is read from disk, and since the REDCAP cache is turned off, no prefetching is performed. However, the REDCAP system is continuously simulated, and for each read request, REDCAP records if it is a cache hit or a miss and, in the latter case, the blocks that would be prefetched to the REDCAP cache.

For each request, the algorithm computes the same times as in the active case, and it also

checks if condition (2.9) is satisfied to determine whether the cache should be turn on again or not. The main difference is that only some of these times can be exactly calculated, whereas others have to be estimated.

T_{Disk_CHit} is exactly obtained while the requests are served. B_{CHit} and $B_{Prefetched}$ are also known, because the cache is being simulated and the amount of both data copied and prefetched is calculated at any time. In contrast, T_{CHit}, T_{Wait} and $T_{Prefetched}$ are not calculated and are estimated by using values stored during the active state and the known values of B_{CHit} and $B_{Prefetched}$.

If B_{CHit_Active} specifies the number of disk blocks copied from the cache to the original read request when REDCAP was last active, and T_{CHit_Active} is the time required to copy B_{CHit_Active} blocks, then T_{CHit} is given by

$$T_{CHit} = \frac{T_{CHit_Active} B_{CHit}}{B_{CHit_Active}}. \tag{2.10}$$

If T_{Wait_Active} is the time waiting for B_{CHit_Active} blocks to arrive from disk when REDCAP was last active, we estimate T_{Wait} with the following expression

$$T_{Wait} = \frac{T_{Wait_Active} B_{CHit}}{B_{CHit_Active}}. \tag{2.11}$$

Analogously, if $B_{Prefetched_Active}$ blocks were prefetched when REDCAP was last on, and the time to read them is $T_{Prefetched_Active}$, then $T_{Prefetched}$ is calculated as:

$$T_{Prefetched} = \frac{T_{Prefetched_Active} B_{Prefetched}}{B_{Prefetched_Active}}. \tag{2.12}$$

REDCAP is kept inactive while condition (2.9) is false, i.e., while the time needed by the cache is more than the time needed to read B_{CHit} blocks. When the algorithm detects that the inequality given by expression 2.9 is satisfied, the cache is potentially effective again, and REDCAP is turned on.

To summarize, in the active state, the algorithm compares the cache time to process the requests with the estimated time to process them without cache. REDCAP cache is turned off if it is not being effective, that is, if the time required by our cache is worse than without it. In contrast, in the inactive state, to activate REDCAP again, the algorithm compares the times to read the requests and to serve them from cache, and it turns our cache on when it detects that can be effective again.

Intermediate states

Besides the active and inactive states, there are also two intermediate states: *pending–active* and *pending–inactive*. They are transition levels between the main states. The pending–active state changes from inactive to active, whereas the pending–inactive one from active to inactive.

When REDCAP state changes, requests previously issued to the I/O scheduler and still pending should be finished. REDCAP is firstly moved to the corresponding pending state where it remains until all these requests are completed. Once these pending requests are finished, and only then, the final change of state is carried out. During the pending state, the

new requests are managed as if the REDCAP system already were in the corresponding final main state. The duration of these intermediate states depends on the number of pending requests of the previous main state. It is important to note that a direct change from a main state to the other one is possible if there are not any pending requests.

In our first implementation of REDCAP, during these intermediate states, it was impossible to undo the state change. Since the activation–deactivation algorithm did not analyze the performance achieved in these temporal states, it was not possible to go back to the initial state till the final one was already reached. For benchmarks with many requests and modifications of the REDCAP state [33], these intermediate states damaged the performance because it took a long time to return to the proper state. The results and peculiarities of this first implementation are explained in Section 2.5.1.

In a second implementation, this problem has been solved by analyzing the performance of the REDCAP cache during temporal states [34]. The algorithm can now change back to the previous state when necessary. As it will be shown in Section 2.5.2, REDCAP is now able to achieve the maximum level of improvement, although its state is modified many times.

Algorithm's operation

In order to estimate the unknown I/O times, REDCAP does not use I/O times of individual requests, but it saves the data of the last 100 requests, and adds up these values to compute the unknown values. The idea is that I/O times can significantly vary from request to request, even for requests with the same type, size and seek. If we use the I/O time of a request to estimate the I/O time of another one, the estimation error can be large, and both positive and negative (i.e., the estimated I/O time can be much larger or smaller than the "real" value). However, when computing the I/O time of a large set of requests, many positive and negative errors cancel each other out, what minimizes the overall computation error. In addition, the REDCAP activation–deactivation algorithm also compares the average I/O times of the last 100 real requests and the last 100 simulated requests.

The check interval defines, in requests, how often is verified whether REDCAP is improving the access time of a normal system or not. The usual setting is 100 requests, but it is changed depending on the behavior of REDCAP. In our algorithm, the check interval is modified according to the following conditions:

- after four consecutive checks getting good results, the interval is doubled up to a maximum of 200, what makes checks less frequent;
- the first time that the algorithm detects that REDCAP is getting worse results than a normal system, the interval is reduced by half until a minimum of 50;
- if two consecutive checks say that REDCAP is producing bad results, the REDCAP cache is turned off and it becomes inactive;
- when REDCAP is inactive, the check interval is fixed to 100 requests and never changes; the first time the algorithm detects that REDCAP could improve the normal system, its cache is switched on and becomes active.

It is worth to remark that the proposed algorithm is independent of the underlying device, because it only takes into account the I/O time of the issued disk requests.

In spite of the fact that the disk model used is quite simple, our results show that it is effective, and works well for a wide range of workloads (see Section 2.5). Obviously, we could

Table 2.1: Specifications of the WD Caviar WD1200BB test disk.

Features	Values
Capacity	120 GB
Cache	2 MB
Read	Adaptive
Write	Yes
Average latency	4.2 ms
Rotational Speed	7200 RPM
Seek time	
Read	8.9 ms (average)
Track–to–track	2.0 ms (average)

consider a more accurate disk model [1, 49, 56, 57], or disk simulator [46, 58] to estimate I/O times, but the disk performance attributes for the disk model or simulator should be investigated, and the cost could increase and become larger than the cost of the calculations of our algorithm. Hence, Chapter 3 is devoted to the design and implementation of an in–kernel disk simulator which greatly matches the behavior of the real disk [35]. There, as a use case, we present a modified version of REDCAP using the disk simulator to evaluate the expected performance of its cache.

2.4. Experiments and methodology

In order to investigate and understand the behavior of REDCAP under several conditions, different aspects have been considered: prefetching size; performance obtained by a larger cache; different access patterns; and impact of a file system. This section discusses the hardware platform used in our experiments, REDCAP cache configurations tested, benchmarks run to carry out the analysis, and different file systems used.

2.4.1. Hardware platform

Our experiments are conducted on a 800 MHZ Pentium–III system with 640 MB of main memory and two disks. The first one is the system disk. It is a Seagate ST–330621A disk [6] which contains a Fedora Core 4 operating system, and collects the traces for a future study. The second one is the test disk, a WD Caviar WD1200BB disk [38], that has a capacity of 120 GB and 2 MB of built–in cache. Its features are listed in Table 2.1. On the test disk there is only one disk partition, and the file system used depends on the experiments done.

2.4.2. Variations in the REDCAP cache configuration

To investigate the impact of the segment and cache sizes on the improvement achieved by REDCAP, several configurations have been tested in our experiments.

Our first study considers four different configurations of the REDCAP cache, where the segment size (i.e., the prefetching size), varies, but the cache size is always the same. The REDCAP cache size has been fixed to 8 MB, which is four times as large as the size of the built–in disk cache of the test disk, and its memory utilization is less than 1.5% of the main memory. We have studied the following segment sizes: 32, 64, 128, and 256 kB. Since these configurations have the same total cache size, their numbers of segments are also different: 256, 128, 64, and 32, respectively. Section 2.5.1 presents the experimental results for these configurations and their analysis.

The second study investigates how the variation of the cache size affects the performance of REDCAP by configuring it with two different cache sizes: 8 and 16 MB. The former size is the one used in our first study. The latter one is eight times larger than the disk cache of the test disk, and less than 3% of memory utilization. In both cases, the segment size has been fixed at 128 kB. This choice is justified by the following two reasons. First, it showed the best behavior in our early tests with different segment sizes and number of segments [33] (see Section 2.5.1). Second, 128 kB is the maximum request size allowed by the file system. The results of these experiments are discussed in Section 2.5.2.

2.4.3. Benchmarks

In order to analyze the impact of the access pattern on the REDCAP behavior and the activation–deactivation mechanism, seven benchmarks have been used in our evaluation. Five of them are I/O–bound. The other two are CPU–bound in our system, but they have been included because they are frequently used for testing purposes. This section describes all of them in detail.

The benchmarks have been selected trying to cover several access patterns: traversal of a directory tree with small files; sequential read; backward read; and two strided access patterns, one with small strides and another with large strides.

The I/O–bound benchmarks are:

- *Linux Kernel Read (LKR)*. It reads the sources of the Linux kernel 2.6.17 by using the command:

  ```
  find -type f -exec cat {} > /dev/null \;
  ```

  This benchmark performs a traversal of a directory tree with small files. It is executed for 1, 2, 4, 8, 16, and 32 processes, each working on its own copy of the Linux kernel source tree. We remark that, in the test disk, there are 32 copies of the kernel files, one for each process.
- *IOR Read (IOR)*. It is commonly used for benchmarking parallel file systems using POSIX, MPIIO, or HDF5 interfaces [43]. Here, version 2.9.1 is used for testing the behavior of REDCAP in parallel sequential reads.

  IOR has been configured with the POSIX API for I/O, and one file per process. File size is 1 GB, and 64 kB is the size to be transferred in a single I/O call. Analogously to the previous benchmark, this test is run for 1, 2, 4, . . . , and 32 tasks, each one reading its own file. Before running the test, files were created in parallel with the IOR benchmark too.
- *TAC*. This benchmark reads files backward with the command `tac` [44]. The test is executed for 1, 2, 4, . . . , and 32 processes, reading, each process, its own file. Files are the same as those from the *IOR Read* benchmark.

- 4 *kB Strided Read*. This test reads a file with a strided access pattern with small strides. The benchmark reads a first block of 4 kB at offset 0, skips a block of 4 kB, reads the next 4 kB block, skips another block, and so on. Again, it is executed for 1, 2, 4, . . . , and 32 processes, and each process reads its own file. These files are the same as in *IOR Read* and *TAC* benchmarks. It has been written in C, and uses the POSIX `read` and `lseek` functions.
- 512 *kB Strided Read*. This benchmark is similar to the previous one, but has a larger stride, thus, the access pattern is different. In this case, every process reads 4 kB, skips 512 kB, reads 4 kB again, skips 512 kB, and so on. When the end of the file is reached, a new read with the same access pattern starts again at a different offset. There are four read series. The first one begins at offset 0, the second at offset 4 kB, the third at 8 kB, and the fourth series at 12 kB. Again, the test is executed for 1, 2, 4, . . . , and 32 processes by using the same files of the previous benchmarks. As the 4 *kB Strided Read* benchmark, it has been written in C using the POSIX `read` and `lseek` functions.

The CPU–bound benchmarks are:

- *Kernel Compilation for 4 Processes*. This test compiles the vanilla Linux kernel 2.6.17 with 4 processes (`make -j 4`) with the configuration for the 2.6.17-1.2142 Linux kernel found the Fedora Core 4 distribution. The "`-j 4`" option allows us to saturate the CPU, and to send to disk as many requests as possible.
- *TPCC-UVa*. This is a free, open–source implementation of the TPC–C Benchmark developed at the University of Valladolid (Spain) [59]. We have used 10 warehouses and 10 terminals. The benchmark is run with an initial 20 minutes warm–up stage and a subsequent measure time of 2 hours.

2.4.4. File systems

File systems determine the access pattern seen by the disk drive to a large extent. Therefore, to evaluate how a file system influences the behavior of REDCAP, five Linux file systems with different features have been considered: Ext2 [60]; Ext3 [41]; JFS [61]; ReiserFS [62]; and XFS [63, 64]. The first one, Ext2, has been the default file system in several Linux distributions for many years, whereas the other four are journaling file systems. All of them are integrated in Fedora Core 4.

Our first set of experiments (see Section 2.5.1) only uses the Ext3 file system, but, in the second one (see Section 2.5.2), REDCAP performance is evaluated for all of them. It is important to remark that we are interested in the REDCAP behavior with each file system, and not in the behavior of the file systems themselves.

Ext2 is an FFS–like file system [65] designed with the goal of expandability while maintaining compatibility. An Ext2 file system is split into block groups, that are essentially identical to the FFS's cylinder groups. Each block group contains a redundant copy of crucial file system control information and also a part of the file system. In order to reduce the disk head seeks made to read an i–node and its data blocks, related i–nodes and data are clustered together in block groups. Moreover, data blocks and i–nodes of all the regular files in a directory are put together in the same group assigned to the directory, or in nearby groups if the corresponding one is full. Ext2 has been included because it is a wide-used and well-understood file system.

Table 2.2: Features of the file systems used.

File system	Block groups	Type	File grouping	Extents
Ext2	Yes	FFS	Per directory	No
Ext3	Yes	Journal	Per directory	No
JFS	Yes	Journal	Per directory	Yes
ReiserFS	No	B+ tree and journal	Per key proximity	No
XFS	Yes	B+ tree and journal	Per directory	Yes

Ext3 is a journaling file system derived from Ext2. Ext3 provides different consistency levels through mount options. Ext2 and Ext3 have the same physical structure, so Ext3 also divides the file system into block groups, and always tries to allocate data blocks for a file in the same group of its i–node, and in the same group of its directory.

IBM's JFS originated on AIX, and from there was ported to Linux. JFS is also a journaling file system which supports metadata logging. Its technical features include extent–based storage allocation, dynamic disk i–node allocation, asynchronous write–ahead logging, and sparse and dense file support.

ReiserFS is a journaling file system which is specially intended to improve performance of small files, to use disk space more efficiently, and to speed up operations on directories with thousands of files. Like other journaling file systems, it only journals metadata. ReiserFS stores file metadata, directory entries, i–node block lists, and tails of files in a single, combined balanced tree (B+ tree) keyed by a universal object ID. In contrast to the other file systems, it does not split the file system into block or cylinder groups, and, therefore, allocation of directories and their files in the same block groups is not performed.

Finally, based on SGI's Irix XFS file system technology, XFS is a journaling file system which supports metadata journaling. It uses allocation groups and extent–based allocations to improve locality of data on disk. This results in a better performance, particularly for large sequential transfers. Performance features include asynchronous write–ahead logging (similar to that provided by Ext3 with `data=writeback` mount option), balanced binary trees for most file–system metadata, delayed allocation, and dynamic disk i–node allocation.

In our experiments, the default options are used for both formatting and mounting the file systems. Table 2.2 summarizes their main features from the point of view of our work.

2.5. Results

This section evaluates the proposed technique, which has been implemented in a Linux kernel 2.6.14 (called to short *REDCAP kernel*). The implementation has been carried out in the block device layer of the Linux kernel 2.6.14, under the page cache and just over the request queue of the I/O scheduler.

We have performed two different sets of experiments. The first one analyzes the behavior of REDCAP and the impact of its segment size. The second one considers two important aspect: the underlying file system and the REDCAP cache size, and it investigates the impact

of their variation. The results of the REDCAP kernel have been compared to those obtained with a vanilla Linux kernel 2.6.14 (called to short *original kernel*) without our proposal.

In order to trace disk I/O activity, we have instrumented both kernels to record when a request starts and finishes, and also when it arrives to the request queue. The REDCAP kernel also records information about the behavior of its cache, such as hits and misses, and the time needed to copy data on a cache hit.

We have performed five runs for every benchmark and system configuration, and the results presented here are the average of these five runs. The confidence intervals for the means, for a 95% confidence level, are also included as error bars. The computer is restarted after each run. Hence, all tests have been performed with a cold page cache and a cold REDCAP cache. The initial state of REDCAP is *active*.

Tests performed are described in detail in Section 2.4.3. For I/O–bound benchmarks, figures depict application time improvements achieved by REDCAP compared with the original kernel. For CPU–bound benchmarks, however, we provide I/O time improvements, where the I/O time is the total time required by all the disk I/O operations.

2.5.1. Evaluation of the REDCAP segment size

Our first experiments analyze how the segment size (i.e., the prefetching size), affects the REDCAP behavior, by considering different configurations of the REDCAP cache with different segment sizes, but with the same total size.

This investigation has been carried out with a REDCAP cache of 8 MB size, and four configurations:

- **256×32kB**, the cache is divided into 256 segments of 32 kB;
- **128×64kB**, in the case, there are 128 segments of 64 kB each;
- **64×128kB**, the cache is split into 64 segments of 128 kB;
- **32×256kB**, the segment size is 256 kB and there are 32 segments.

Since Complete Fair Queuing (CFQ) [42] is the most widely used I/O scheduler in Linux [52], and it is the default I/O scheduler in the “official” versions of the Linux kernel, we have used it in these experiments.

Here, the test disk uses a fresh Ext3 [41], with a logical block size of 4 kB. The file system contains nothing but the files for carrying out the benchmarks.

Note that, for this study, we use the first implementation of REDCAP, where on the temporary states, the algorithm does not perform any analysis (see Section 2.3.3). This was the implementation available when these experiments were conducted.

Linux Kernel Read

Let us start analysing the application time improvement achieved by REDCAP with respect to the original kernel in the *Linux Kernel Read* benchmark. Results are presented in Figure 2.6 for the four REDCAP configurations and different numbers of the processes.

The REDCAP results are always better than the original kernel ones, and the corresponding improvement increases with the number of processes. The 64×128kB configuration shows the best performance for 1, 16 and 32 processes, obtaining a 79% reduction for 32 processes,

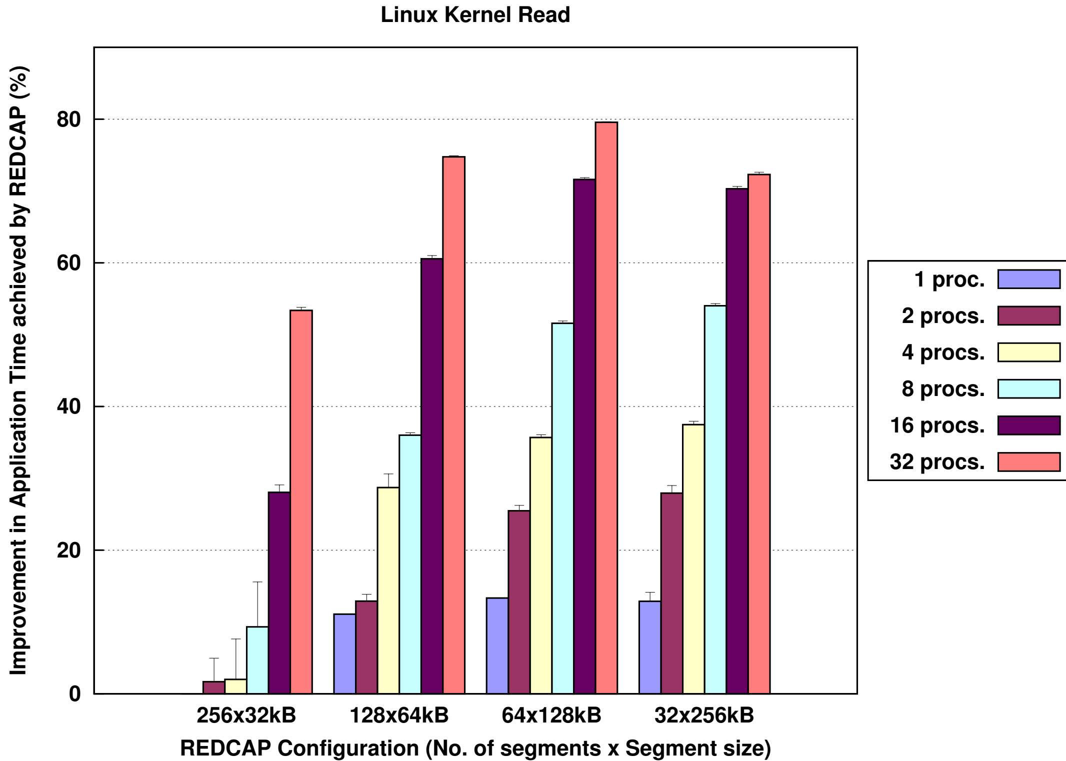


Figure 2.6: Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel for the *Linux Kernel Read* benchmark depending on the RECAP segment size.

whereas for 2, 4 and 8 processes it is achieved by 32×256kB. Although the 256×32kB configuration presents the smallest improvements, it is still clearly better than the original kernel for 8 or more processes, reducing the application time by 54% for 32 processes. With this configuration, for 1, 2, and 4 processes, both kernels statistically have the same throughput.

The REDCAP behavior strongly depends on the segment size, i.e., the prefetching size, and the time reduction increases with it. The exception can be found for 16 and 32 processes, because the higher time reduction is achieved with the 64×128kB configuration, and not with the 32×256kB one. The reason is that the number of segments, 32, of the 32×256kB configuration is not enough for 16 and 32 processes, and some prefetched segments are evicted before being reused. Therefore, the improvements obtained decrease slightly, although they reach 72%.

An explanation for these good results can be found in the way this benchmark reads the large amount of small files of a Linux kernel source tree. The reading process is performed directory by directory. In an Ext3 file system, regular files of the same directory are stored together in disk in the group assigned to the directory (or in nearby groups if the corresponding group is full) [41]. The operating system is not able to notice this pattern of close disk accesses nor does it perform a prefetching of files because they are small (it only prefetches disk blocks of a specific file when detects a sequential access). However, as already mentioned in Section 2.3.2, REDCAP exploits the spatial locality, so almost all the prefetched blocks are finally read by the processes, and, in fact, the REDCAP cache is almost always active.

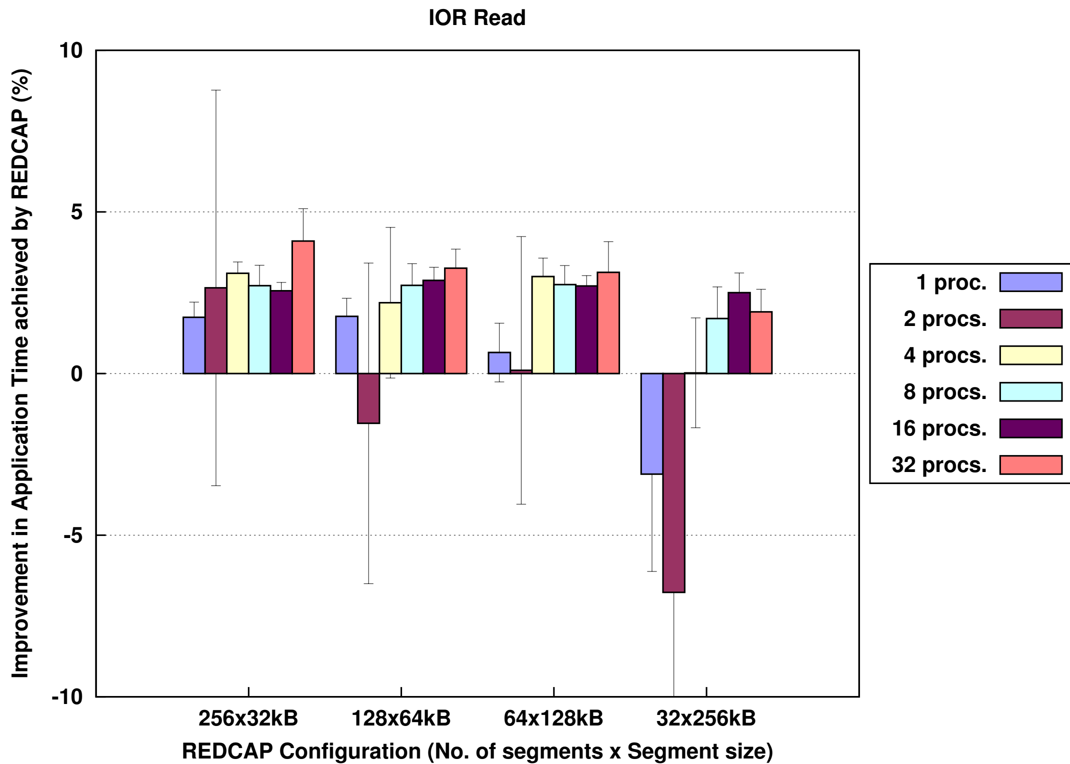


Figure 2.7: Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel for the *IOR Read* benchmark depending on the RECAP segment size.

Regarding the 256×32kB configuration, the problem is that the prefetching performed by REDCAP with this configuration is quite small, 32 kB, and does not contribute any benefit. For 1, 2 and 4 processes, the disk cache is large enough and being effective, and, hence, the original requests profit the prefetching performed by the disk itself. Thus, REDCAP is not being effective, although it is active up to 47%. However, for 8 or more processes, the disk cache is not large enough and some of its prefetched data are evicted before being read. Therefore, although the REDCAP prefetching is still small, it is effective, and the application time reduction reaches 54%.

IOR Read

Figure 2.7 depicts the application time improvement achieved by REDCAP with respect to the original kernel in the *IOR Read* benchmark, as a function of the REDCAP configuration and the number of processes.

The behavior of REDCAP is very similar to the original kernel's one, but our proposal achieves small improvements that are due to the prefetching performed by REDCAP at the beginning of the test (remember that REDCAP is initially active). Since all the read files for IOR are created in the same directory, their i–nodes, and first blocks of data and metadata are allocated in an interleaved way in the same block group assigned to the directory. This makes a prefetched REDCAP segment contain blocks of different files, what, in turns, benefits not

only the process that submits a request, but all the processes reading files from the same group. Therefore, the prefetching initially performed by REDCAP makes the first read operations much faster in REDCAP than in a vanilla kernel.

The exception is the 32×256kB configuration for 1 and 2 processes, where REDCAP performance is slightly worse than that of the vanilla kernel because the former reads more blocks than the latter. In these cases, the activation–deactivation algorithm turns the REDCAP cache on and off several times, when the best performance is achieved by an inactive cache. The problem is that sometimes, when REDCAP is inactive, it receives series of small requests (caused by meta–data reads) which turn the cache on before the algorithm realizes that the subsequent requests advice to keep it off.

The *IOR Read* test has a sequential access pattern, and the prefetching techniques of both the operating system and the disk cache are optimized for this kind of pattern. Indeed, the most part of the read requests issued has a size of 128 kB, which is the maximum disk request size allowed by the operating system. Therefore, the contribution of our method is rather small, and even a copy time is added in each cache hit, so the activation–deactivation algorithm detects this worsening, and the cache is turned off and is inactive most of the time.

TAC

The results for the application time achieved by REDCAP, as compared to the original kernel, for the *TAC* benchmark are presented in Figure 2.8, using all the considered configurations and number of processes.

The 64×128kB and 32×256kB configurations significantly improve the application time. Both show a qualitatively similar but quantitatively different behavior, that strongly depends on the corresponding configuration, i.e., the segment size.

The best performance is achieved by the 32×256kB configuration, with improvements of up to 40% for 4, 8 and 16 processes. In this case, the cache is always active and the activation–deactivation algorithm never turns it off.

Regarding the 64×128kB configuration, REDCAP decreases the application time with respect to the original kernel in all the cases. The greatest performance is a 16% reduction achieved with 4 and 16 processes. The cache is almost always active and is rarely turned off.

An explanation of the good results achieved for these two configurations is found in the Ext3 file system. Ext3 tries to allocate all the data blocks of a regular file together in disk in such a way that the sequential access is optimized [41]. As we have mentioned before, this spatial locality benefits the REDCAP prefetching. In contrast, the original kernel is not able to detect the backward access pattern, so it does not perform any prefetching.

On the other hand, for the 256×32kB and 128×64kB configurations, the results achieved by our proposal are similar to those obtained by the original kernel. Since the version used of the `tac` command reads files backward with requests of 64 kB, the prefetching (32 and 64 kB, respectively) performed by REDCAP with both configurations is very small and does not contribute any benefit. The algorithm detects that our cache is not being effective, and turns it off. Indeed, it is inactive all the time.

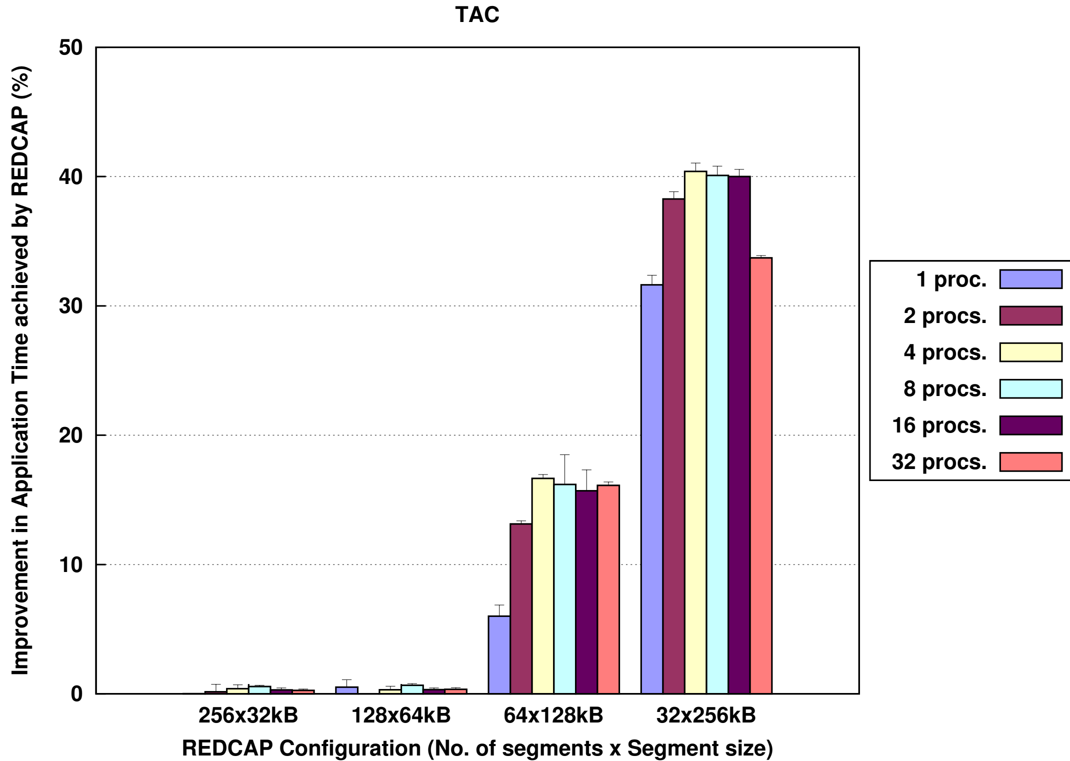


Figure 2.8: Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel for the *TAC* benchmark depending on the RECAP segment size.

4 kB Strided Read

Figure 2.9 shows the application time achieved by REDCAP compared to the original kernel in the *4 kB Strided Read* benchmark.

The REDCAP behavior strongly depends on both the segment size and the number of processes. For the 128×64kB, 64×128kB and 32×256kB configurations, it performs always better than the vanilla kernel, and the best results are achieved for 1 process with reductions of 45%, 40% and 33%, respectively. For the 64×128kB and 32×256kB configurations, the improvements obtained decrease when the number of processes increases. However, for 32 processes, a significant reduction of the time is still achieved with reductions of 13% and 4.7%, respectively. For the 128×64kB, the 8 processes case provides the smallest throughput, but still better than the original kernel with a 5% reduction.

On the other hand, for the 256×32kB configuration and the 1, 2 and 32 processes, our results improve the original kernel's ones, whereas for 4, 8 and 16 processes no further improvements are achieved, and the results are indistinguishable from those of the original kernel.

Since the activation–deactivation algorithm is not able to decide the proper state with this access pattern, the REDCAP cache is turned on/off many times. The problem, which only appears in this case, can be easily explained. When the REDCAP cache is active, the disk drive detects a sequential access pattern, and activates its read–ahead mechanism. The original requests profit the prefetching performed by the disk itself, and take a small time.

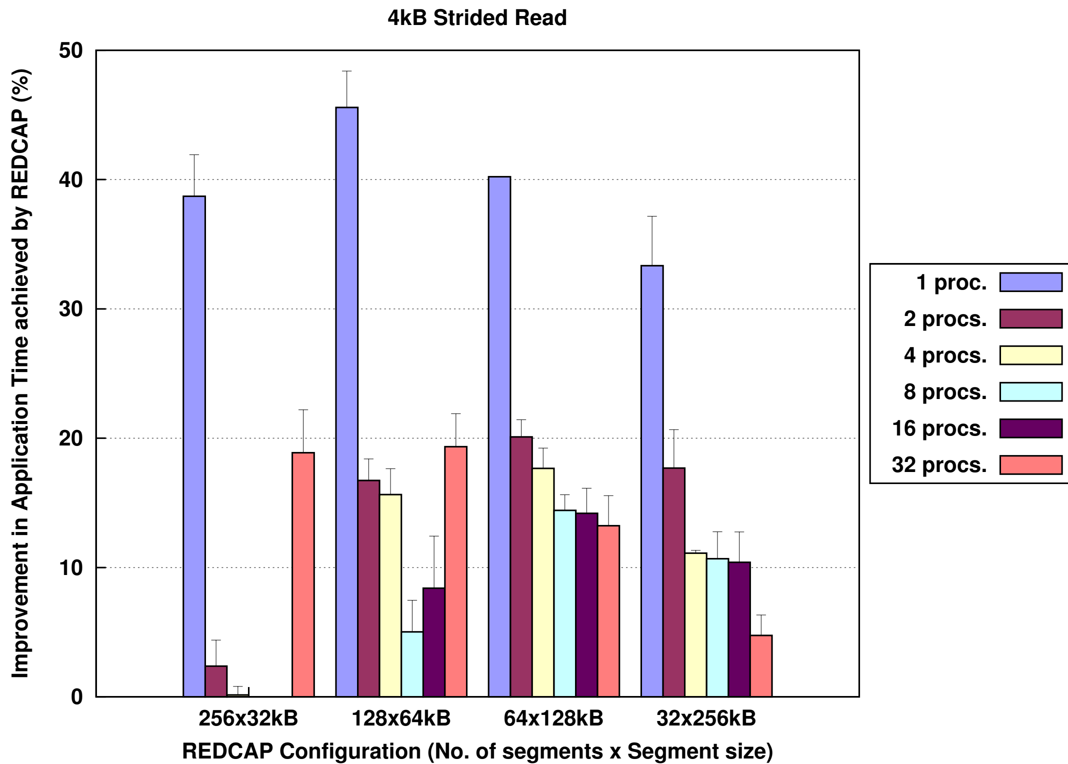


Figure 2.9: Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel for the *4 kB Strided Read* benchmark depending on the RECAP segment size.

Therefore, the algorithm decides that the cache cost exceeds the cost of directly reading data from disk, and it turns the cache off. However, when REDCAP is inactive, requests are not sequential, and the disk deactivates its read–ahead mechanism, making original requests take more time. Then, the algorithm decides that it is better to turn the REDCAP cache on again. Since this state switch is successively repeated, final results do not show a systematic behavior as in previous cases.

It is also interesting to note that the operating system does not detect this access pattern, nor does it implement any technique to enhance the performance under this type of workloads.

512 kB Strided Read

Let us continue discussing the REDCAP results in the *512 kB Strided Read* benchmark. The behavior of our technique is presented as a function of its configuration and the number of processes in Figure 2.10.

REDCAP does not perform better than the original kernel and shows a quantitative similar behavior for all the configurations. Indeed, the activation–deactivation algorithm detects that access time is not being improved and turns our cache off. The worst results appear for 1 and 2 processes, with an increase of 3.6% (for all the configurations) and 3.5% (for the 32×256kB configuration), respectively. An explanation for these results is found in the small application time for 1 and 2 processes. The REDCAP cache is turned off in the first check, and is kept

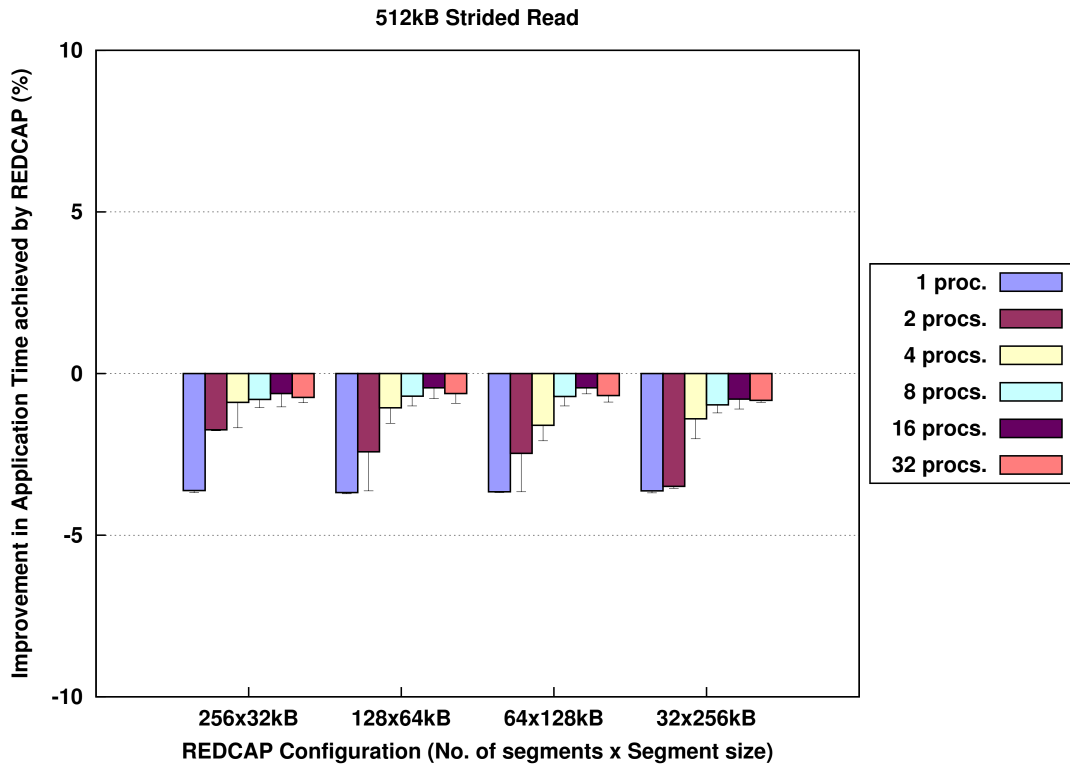


Figure 2.10: Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel for the *512 kB Strided Read* benchmark depending on the RECAP segment size.

inactive for the rest of the time, but the initial lost of time when it was still active is not recovered later on. For 4, 8, 16 and 32 processes the loss can be consider negligible, and it is mainly due to the time employed to simulate the behavior of the cache when it is inactive.

As in the *4 kB Strided Read* benchmark, the operating system does not implement any prefetching technique for this access pattern.

Kernel Compilation for 4 processes

As the *Kernel Compilation* benchmark is CPU–bound, here we focus our attention on the I/O time improvement achieved by REDCAP with respect to the original kernel. Figure 2.11 illustrates the results.

REDCAP is always are better than the vanilla kernel. The best performance, 4.4% of improvement, is provided by the 128×64kB configuration, while the lowest performance is 3.5% for the 32×256kB case. The number of requests served by the REDCAP cache when it is active ranges from 5% for the 256×32kB configuration to 43% for the 64×128kB and 32×256kB configurations. During the execution of the test, the cache is turned on and off several times. Nevertheless, we have checked that the best results in this benchmark would be achieved by having the cache active all the time, with an improvement of up to 6.9%

We remark that during the kernel compilation files from different directories are used at the same time, but not all the files of the same directory are read consecutively. As a consequence,

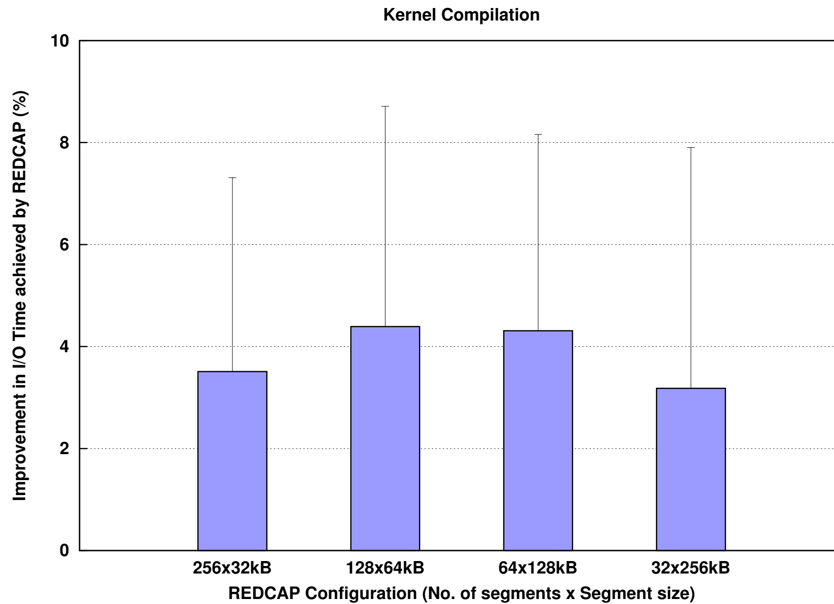


Figure 2.11: Improvement, in the I/O time, achieved by REDCAP over a vanilla Linux kernel for the *Kernel Compilation* benchmark depending on the RECAP segment size.

the spatial locality, existing in the *Linux Kernel Read* benchmark, is partly lost in this test. Thus, despite the fact that all the files of the kernel source are also read in this test, the corresponding results can not be compared to those obtained by the *Linux Kernel Read* one.

TPCC

Since *TPCC* is also CPU–bound, we show I/O time improvement achieved by REDCAP with respect to original kernel. The corresponding results are presented in Figure 2.12 for the four analyzed configurations.

As we can see, the behavior of REDCAP is very similar to the original kernel's one, and we can say that both kernels statistically get the same results. REDCAP only achieves an improvement of 1% in some cases.

The *TPCC* test has a random read pattern, so the REDCAP cache can not be effective, and it is almost impossible to take advantage of the prefetching performed. The activation–deactivation algorithm detects this and turns the cache off, which is inactive a long time in all the tests.

2.5.2. Impact of the file system and cache size

To get a better insight into the advantages and features of REDCAP, we analyze its behavior and evaluate its performance by using several file systems, different from Ext3, and a larger cache.

The access pattern seen by a disk drive depends, to a large extend, on the file system. In the present work, we evaluate its impact on the REDCAP behavior by considering five Linux file systems with different features: Ext2 [60]; Ext3 [41]; JFS [61]; ReiserFS [62]; and

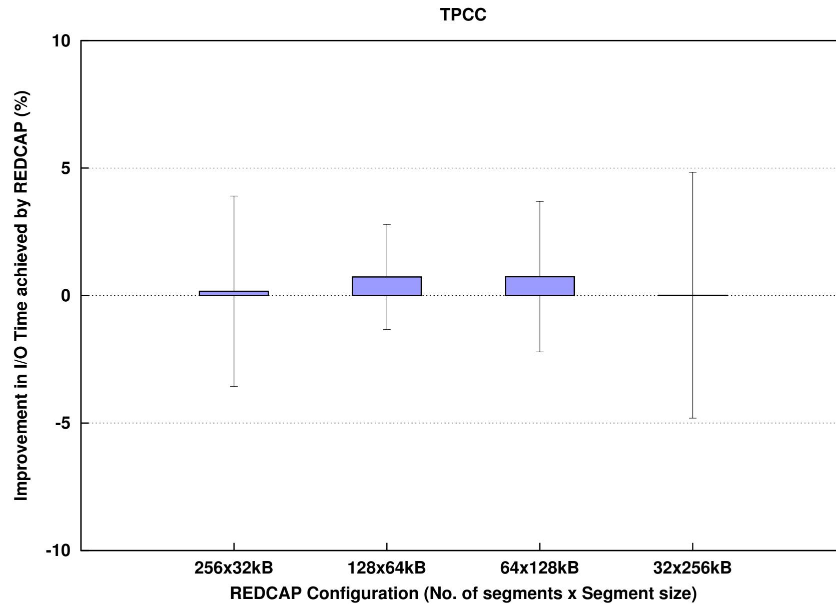


Figure 2.12: Improvement, in the I/O time, achieved by REDCAP over a vanilla Linux kernel for the *TPCC* benchmark depending on the RECAP segment size.

XFS [63, 64]. With this selection we have cover the main characteristics of the modern file systems: journaling; block or cylinder groups; extents to define data block ranges; and B+ trees to store i–nodes, directories or even all the file system. All these file systems are integrated in the Fedora Core 4 distribution. A brief description of their features, relevant for this study, is given in Section 2.4.4.

Remember that our goal is to investigate the throughput achieved by REDCAP with each file system, not to provide a detail comparison between the results obtained by all the file system. Indeed, if REDCAP achieves a very good improvement with one of them, it does not necessarily mean that this file system has the best behavior.

The performance achieved by a disk cache improves as its size is increased, although, if the size grows beyond a certain threshold, it only achieves a marginal contribution, that is not cost–effective [31, 32]. Therefore, in order to analyze how a larger cache size affects the performance of REDCAP, its cache has been configured with two different sizes: 8 and 16 MB. The aim is to determine how to achieve the best improvements with a minimum memory utilization.

By using a segment size of 128 kB, two configurations of the REDCAP kernel have been tested:

- **64×128kB**, there are 64 segments of 128 kB each.
- **128×128kB**, the cache is split into 128 segments of 128 kB.

Again, the CFQ scheduler [42], which is the default I/O scheduler in the latest "official" versions of the Linux kernel, has been used in all the experiments.

For this study, we have implemented the second version of REDCAP, where, on the temporary states, the performance analysis is done, and and a change back to the previous state is allowed (see Section 2.3.3).

Linux Kernel Read

Let us start discussing the application time improvement achieved by REDCAP with respect to the original kernel in the *Linux Kernel Read* benchmark. The results for both REDCAP configurations are presented in Figure 2.13 as a function of the file system. Note that, for the JFS file system, this test could not be executed for 32 processes because the computer ran out of memory with both kernels.

The REDCAP results are always better than the original kernel ones, and for almost all the cases, improvements become better as the number of processes increase. The relationship between the number of processes and the number of REDCAP segments can explain this fact. There are always more REDCAP segments than processes, whilst the disk cache does not have enough segments. At the same time, the percentage of cache hits is quite high, much higher than in the disk cache.

Except for the ReiserFS file system, results obtained by the two cache configurations are very similar, although the 128×128kB one achieves slightly better improvements. Two possible explanations could be given for the fact that an increase in the cache size does not imply a further reduction in the application: i) in the 64×128kB configuration, REDCAP is almost always active and it is already taking maximum advantage of the prefetched blocks; and ii) the number of segments is larger than the number of processes, and this prevents data prefetched into the REDCAP cache from being evicted by other read requests of other processes. We believe that with more than 64 processes, some performance differences between both configurations should be expected.

REDCAP shows a quite similar behavior for Ext2, Ext3 and JFS. With the first two, our proposal presents its best performance for 32 processes, reducing the application time by up to 83% in both cases. For JFS, it performs best with 16 processes, achieving a 78% improvement.

Special attention should be given to the behavior of REDCAP with ReiserFS. In this case, the improvements achieved do not increase as the number of processes increase. The best performance is obtained for 8 processes with a reduction of up to 57% for the 128×128kB configuration. For 16 and 32 processes, the performance decreases, and in particular, for 32 processes and the 64×128kB configuration, our technique provides no contribution.

The main reason for this behavior is the structure of ReiserFS, that produces apparent random accesses. In a normal system (without REDCAP), random requests do not benefit from the disk cache; it is not large enough and disk blocks, prefetched by the disk controller, are evicted from the disk cache before being read. However, in a REDCAP system, its cache is big enough to keep the prefetched segments for a long time, and many disk blocks are requested before the corresponding segments are evicted. This is specially true for 1, 2, 4, and 8 processes. In these cases, REDCAP produces significant reductions in the application time, and the performance achieved is quite good. Moreover, the best improvements are obtained for 2 and 4 processes when compared to the other file systems.

For 16 and 32 processes, the REDCAP cache is not large enough for ReiserFS either, and the improvement achieved decreases, because many prefetched segments are evicted before being reused. Nevertheless, the REDCAP kernels are still clearly better than the original kernel, except for 32 processes and the 64×128kB configuration, where they show a similar behavior to the original one. Another problem is that the randomness produced by ReiserFS and the increase in the number of requests make the activation–deactivation algorithm unable

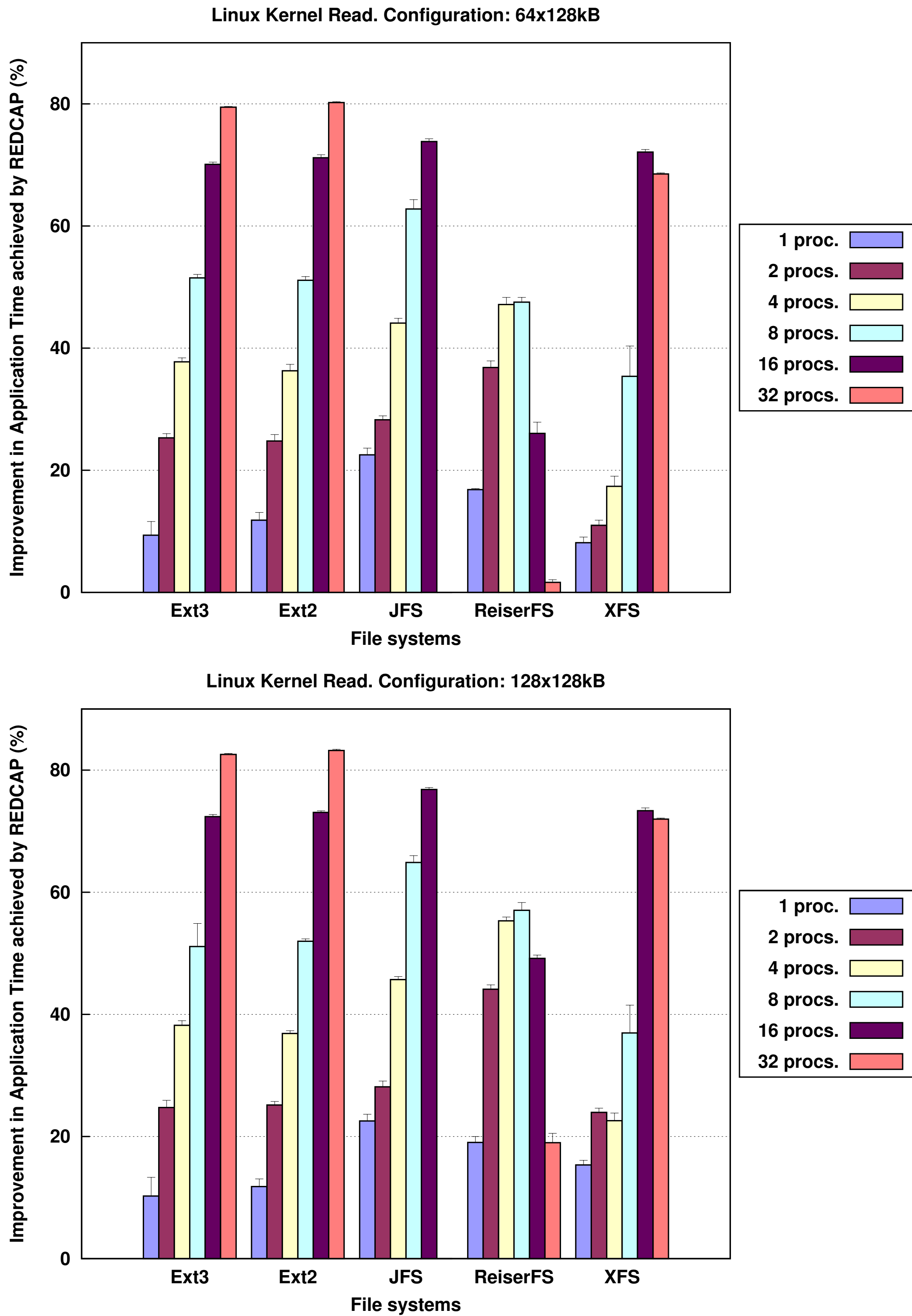


Figure 2.13: Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel for the *Linux Kernel Read* benchmark depending on the file system used.

to decide the proper state of the cache, turning it on and off several times. However, we have checked that the best results for ReiserFS in this benchmark are achieved by having the cache active all the time.

It is interesting to note that differences between the results obtained by the two REDCAP configurations are significant for ReiserFS (see Figure 2.13). These differences confirm the segment eviction problem and point out that, with the 64×128kB configuration, the number of segments is not large enough in some cases, and some of them are evicted from the cache even before REDCAP takes advantage of them completely.

Finally, XFS has its best performance for 16 processes, with a reduction by up to 73%. With this file system, REDCAP presents the smallest improvements for 1 to 8 processes, although they are still clearly better than those of the vanilla kernel. In these cases, the application time reduction achieved ranges from 8% to 34%. However, it is important to remark that the benefit achieved is as large as possible because our cache is always active with this file system.

IOR Read

Figure 2.14 depicts application time improvements achieved by REDCAP with respect to the original kernel for the *IOR Read* benchmark.

For all the file systems, the behavior of REDCAP is very similar to the original kernel one, but the confidence intervals are quite big. Hence, we can conclude that, statistically, both kernels have the same performance.

Remember that this benchmark has a sequential access pattern, and that the prefetching techniques used by the operating system and the disk cache are optimized for this kind of pattern. Due to the prefetching performed by the operating system, most of the read requests issued have a size of 128 kB, which is the maximum disk request size allowed by the file system. Thus, the contribution of our method is rather small, and a copy time is added on each cache hit, so the activation–deactivation algorithm detects this behavior, and the cache is turned off and is inactive almost all the time.

TAC

The REDCAP results for the application time compared to the original kernel ones are presented in Figure 2.15 as a function of the file system for the *TAC* benchmark, and both cache configurations.

The REDCAP kernels always perform better than the original kernel with all the file systems except JFS. In the JFS case, small improvements are achieved, and our approach basically behaves much like the vanilla kernel.

For all the file systems, the results are independent of the configurations, and in both cases are quite similar. Then, we can conclude that a cache size of 8 MB is enough for this benchmark.

For Ext2, Ext3 and XFS, the proposed technique shows a qualitatively and quantitatively similar behavior. For Ext2, its best result is got for 16 processes with a reduction of 16%. For Ext3, the best result is achieved for 8 processes with an improvement of 17%. And finally, for XFS, it is achieved for 4 and 16 processes with a 17% reduction in both cases. For Ext2 and Ext3, the cache is almost always active, whereas for XFS the number of requests served

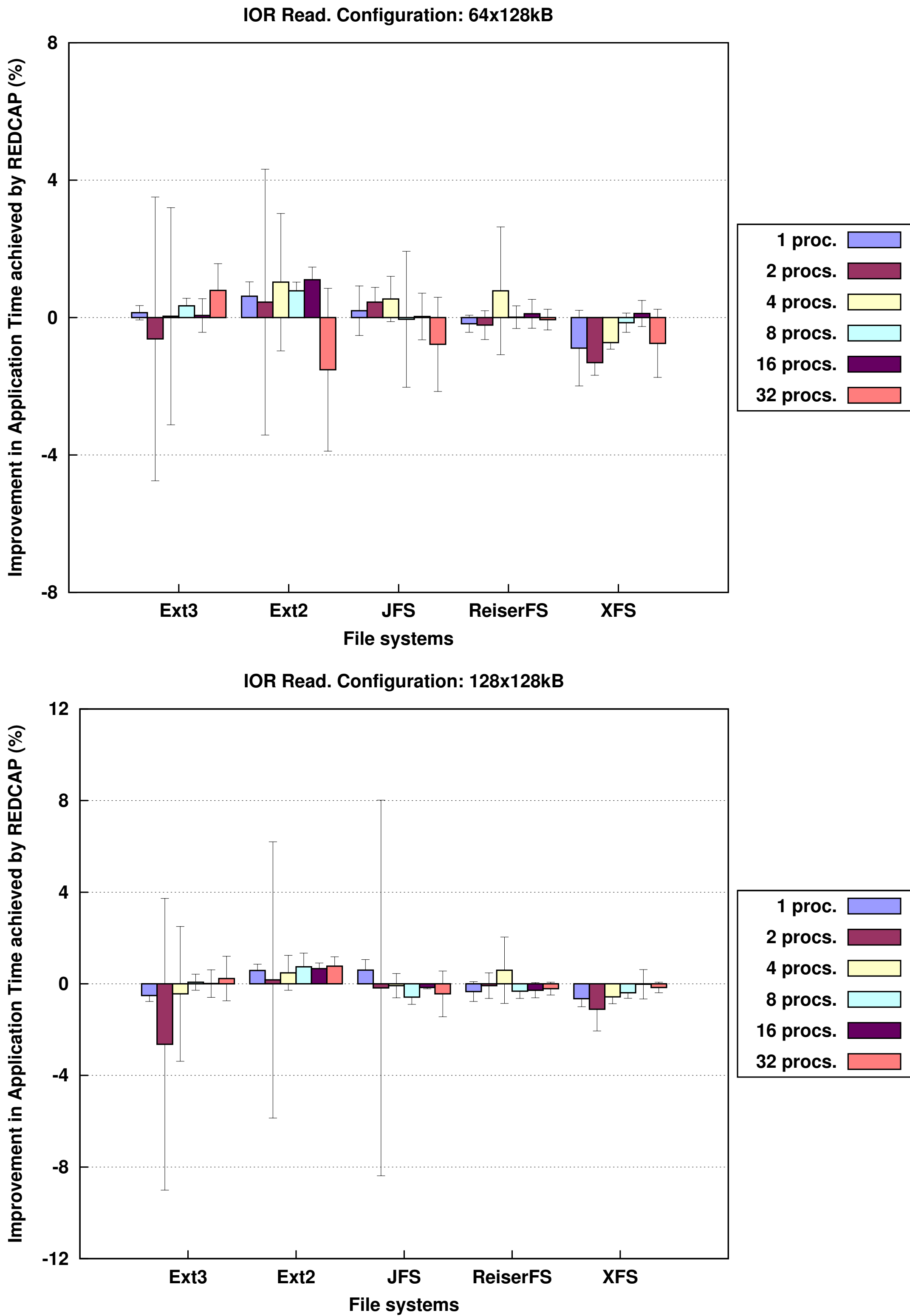


Figure 2.14: Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel for the *IOR Read* benchmark depending on the file system used. Note that the percentage values in the Y axis are small.

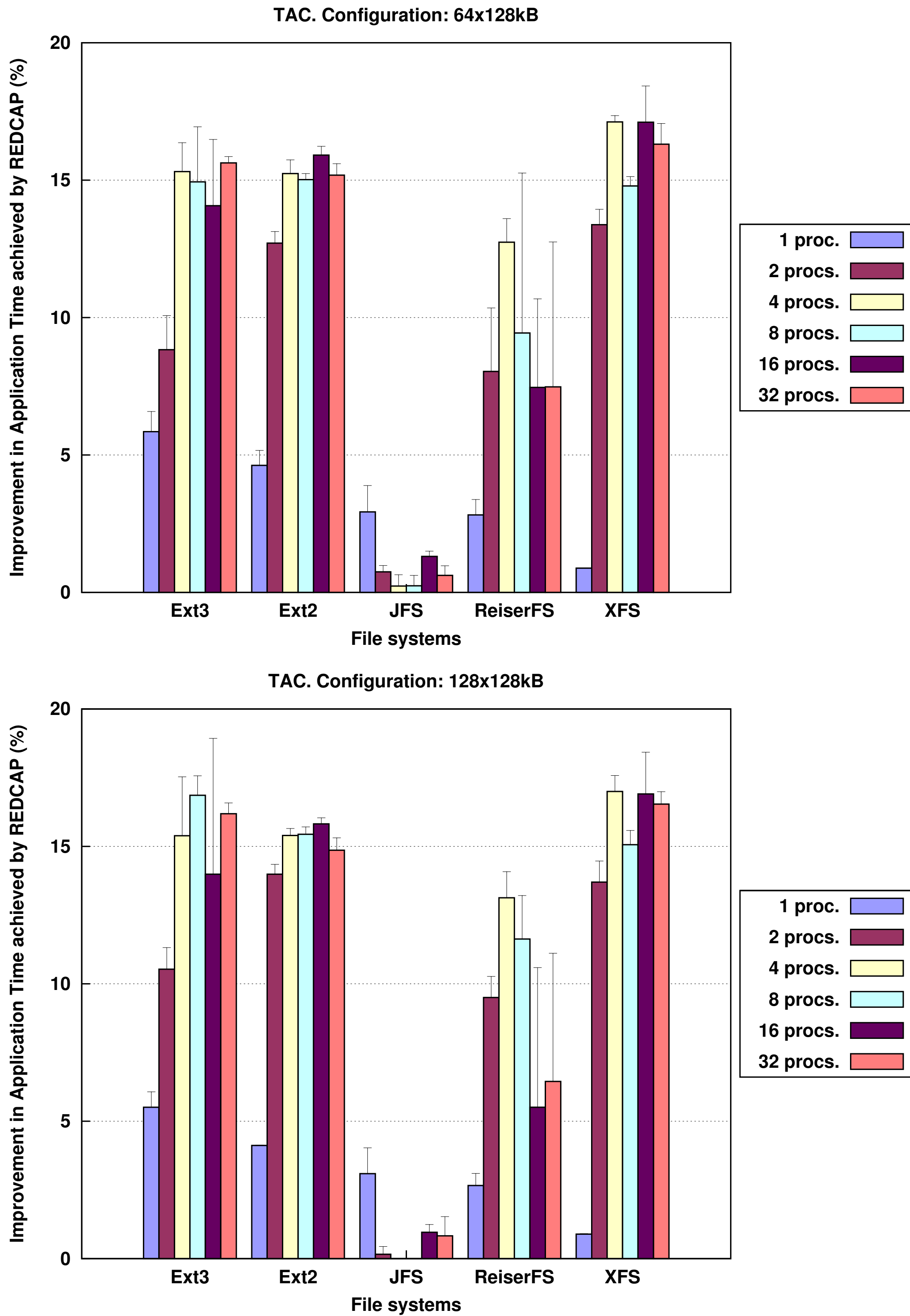


Figure 2.15: Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel for the *TAC* benchmark depending on the file system used.

by the REDCAP cache when it is active ranges from 74% to 99%. For all of them, REDCAP has its smallest improvements for 1 process, in spite of the fact that its cache is always active. Even then, for Ext3 and Ext2, REDCAP is still better than the original kernel by a 6% and 4.6%, respectively, but for XFS, our proposal does not improve the application time, and presents the same behavior as the original kernel.

Special attention should be given to the results for the JFS file system. In this case, the behavior of REDCAP is quite similar to that of the original kernel, and it does not provide any benefits. The reason can be found in the physical structure of JFS and the way data blocks are allocated. JFS divides the disk into *Allocation Groups* containing i–nodes and data blocks in such a way that i–nodes and their associated data can be stored in physical proximity. In our case, the allocation group size is 1 GB, and is equal to the size of the files read by `tac`. By using the `filefrag` command [66], we have found out that, although all the files are created in parallel, almost all the blocks of a file are stored in the same allocation group as only one extent. This block allocation produces sequential backward access patterns for `tac`. The operating system does not detect these access patterns and does not prefetch any block, REDCAP, however, is aware of these patterns, but it deactivates its prefetching mechanism because it is unable to detect the small improvements that it can achieve.

The problem is that modern disk drives use an optimistic approach for settling a disk head before a read operation, and they attempt a read as soon as the head is near the desired track. If the settle has not been completed and the head is on the wrong track, nothing has been lost. On the contrary, if the head is right positioned and data is correctly read, an entire revolution's delay has been saved [1]. As a consequence of this optimistic approach, blocks previous to the requested ones can be additionally read and stored in the disk cache. The effect is as if disk drives were performing backward prefetching, but they are not, and they are not able to detect such access pattern either. Through this thesis we refer to this behavior as *immediate read*.

Due to the read–ahead performed by the *immediate read*, and the sequential backward access pattern produces by tac on JFS, which fully exploits that read–ahead, the contribution of our approach is rather small in this case. If the REDCAP cache was always active, the average improvement would be 4.6%. The activation–deactivation algorithm is not able to detect this small benefit, and the cache is off all the time.

Figure 2.16 confirms our theory. It shows seeks performed to serve read requests during the execution of this test for 4 processes, the original kernel and the JFS file system. The seek pattern is determined by the CFQ I/O scheduler, that provides fairness at a per–process level, giving to each process exclusive access to the disk for a period of time [42]. CFQ consecutively serves a few requests of the same process by selecting them in the following way:

- A request of a new process, P_1, is selected, generating a large seek from one allocation group to the allocation group of the file that P_1 is reading.
- A few requests of the process P_1 are usually selected. Thus, small seeks of 128 kB (256 sectors) are performed. (Note that, due to the backward access, the jump is 128 kB).
- A request of a different process, P_2, is selected, and hence a large seek is performed from the current allocation group to the allocation group of the file that P_2 is reading.
- A few requests of P_2 are usually selected, and small seeks of 128 kB are again performed, and so on.

This sequence in the selection of requests is performed between the four processes and can be observed in Figure 2.16. The four lines at the top of the figure correspond to large seeks between requests of different processes, i.e., between allocation groups. These lines correspond with the requests issued when the scheduler selects a new process. The line at the bottom part corresponds to the 128 kB seeks to serve requests of a same process.

It is interesting to note that, for the other file systems, when creating files in parallel, blocks are allocated in a more interleaved way. Therefore, benefit is gained by the prefetching performed by REDCAP, that also benefits from the *immediate read* performed by the disk drive. The interleaved allocation of blocks implies that a REDCAP segment contains blocks that are read by different processes. The outcome of this fact is twofold. First, a segment, which is read because of a cache miss caused by a process, will probably produce cache hits for other processes in the benchmark. Second, a process reading blocks will partially use several segments. As a consequence, the REDCAP cache can have many partially read segments which can produce a series of cache hits for requests of several processes. Moreover, if a process produces a cache miss when the other processes are having cache hits, the disk controller will have enough time to prefetch all the blocks of the corresponding missed segment before serving new requests. Therefore, the I/O time of the prefetched segment will be hidden, and the overall system performance will improve. This can be seen with Ext3, where the REDCAP kernel reads more data than with JFS, but where the achieved improvement is significantly larger than with JFS. Figure 2.17 shows seeks performed to serve read requests during the execution of this test for 4 processes, the original kernel and the Ext3 file system. The seek pattern is also determined by CFQ, and again the line at the bottom part of the figure corresponds to the 128 kB seeks of requests of the same process. But with this file system, the large seeks among different processes are not as clearly represented as with JFS, and the points at the top part correspond to these seeks, and reflect the interleaved block allocation.

Finally, ReiserFS shows its best result for 4 processes with an improvement of up to 13% and a cache almost always active. The smallest improvement is obtained for 1 process, but the application time is still better than that provided by the original kernel, with a 3% reduction and an active cache all the time. For 2, 8, 16 and 32 processes, the algorithm is not able to decide the proper state and, although the improvements achieved are good, ranging from 7% to 9%, and from 5% to 11% with the 64×128kB and 128×128kB configurations, respectively, they could be a little better if the cache were always active.

4 kB Strided Read

Figure 2.18 depicts the REDCAP results for the application time compared to the original kernel results with both cache configurations, in the *4 kB Strided Read* benchmark as a function of the file system.

For this access pattern, REDCAP always performs better than the vanilla kernel, although its behavior strongly depends on the file system. Since both cache configurations obtain quite similar results, the size of the smallest cache (8 MB in the 64×128kB configuration) is enough to achieve the maximum improvements. The REDCAP cache is almost always active in this test.

The best results are achieved for 1 process with reductions of up to 36%, 42%, 49%, 30% and 48% for Ext2, Ext3, JFS, ReiserFS, and XFS, respectively. For the rest of the processes,

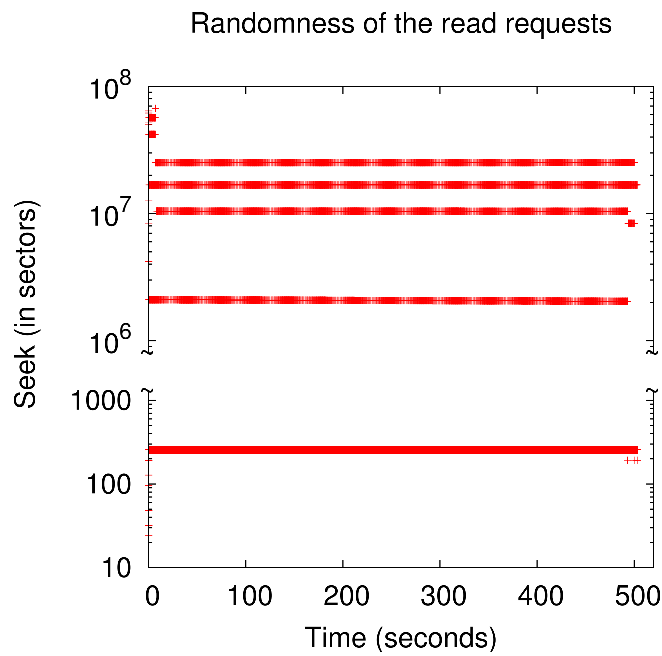


Figure 2.16: Seeks of the read requests performed during the execution of the *TAC* benchmark run with 4 processes, the original kernel and the JFS file system. Note the log scale in the Y axis.

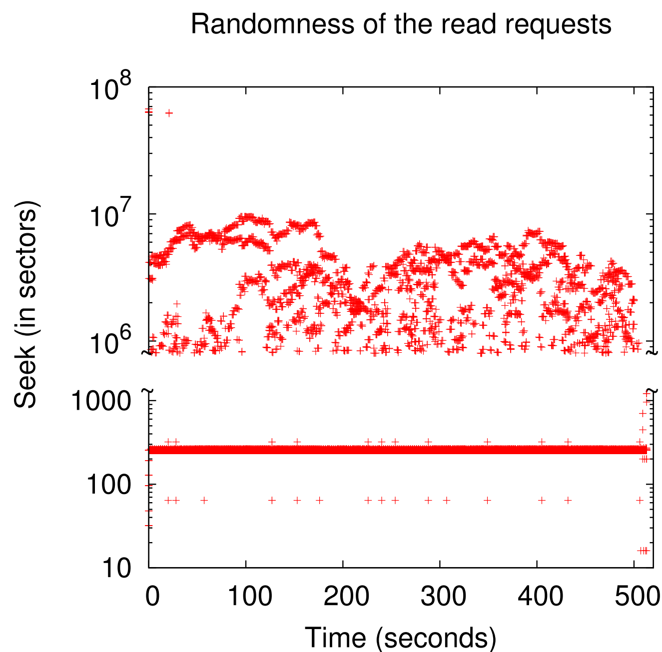


Figure 2.17: Seeks of the read requests performed during the execution of the *TAC* benchmark run with 4 processes, the original kernel and the Ext3 file system. Note the log scale in the Y axis.

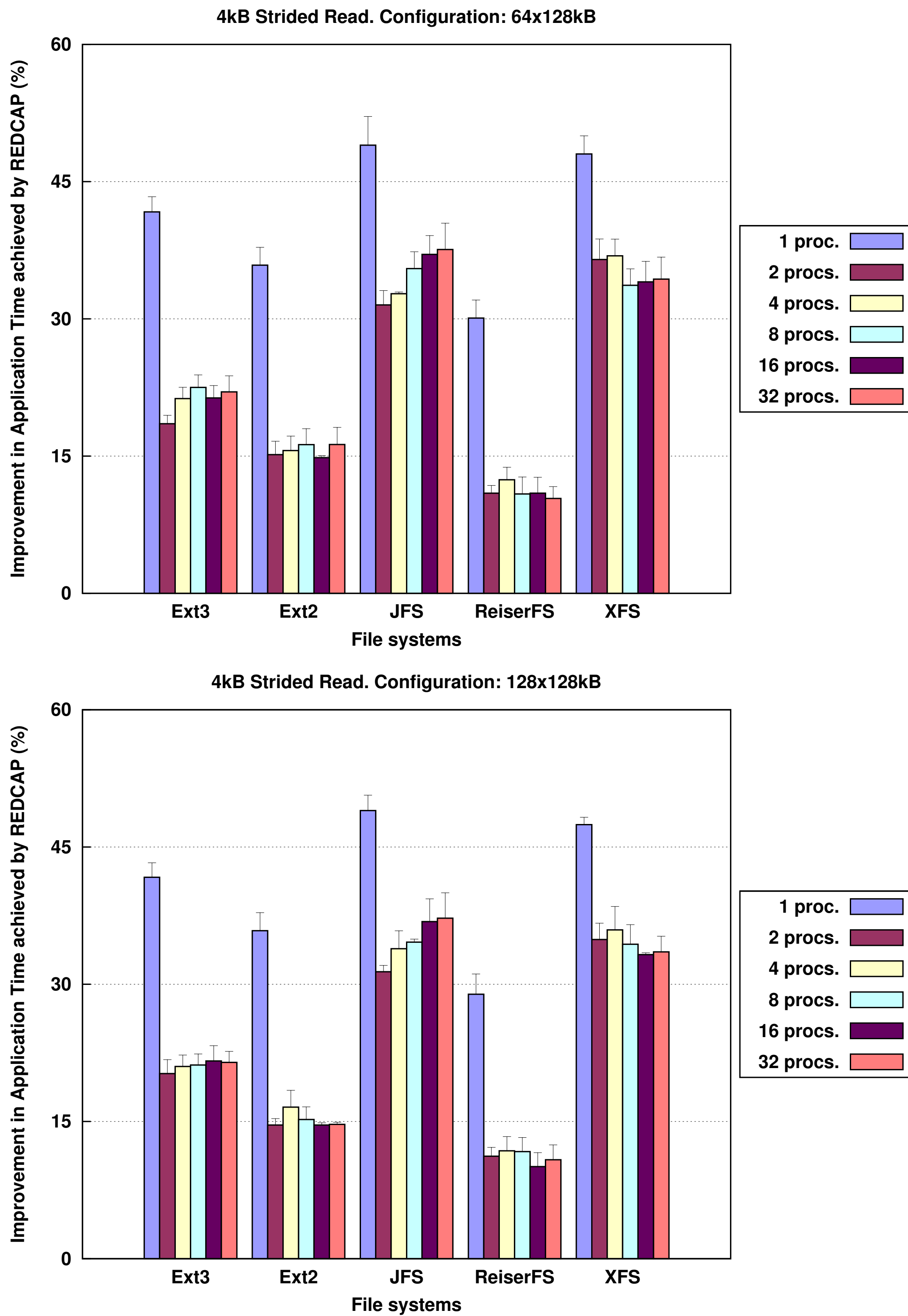


Figure 2.18: Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel for the *4 kB Strided Read* benchmark depending on the file system used.

the improvements are smaller, and range from 14% to 16% for the Ext2 file system, from 19% to 23% for Ext3, from 31% to 37% for JFS, from 10% to 12% for ReiserFS, and from 33% to 37% for XFS.

As we mentioned in Section 2.3.3, the problem of the first version with the intermediate states has been already solved. The results of our current implementation are better than those obtained in our first analysis [33] (see Section 2.5.1). Although our cache is still made active–inactive many times, because the algorithm is not able to decide the proper state, REDCAP achieves the maximum possible reductions. As we have explained in Section 2.5.1, the problem, appearing only in this benchmark, is that, when the REDCAP cache is active, the disk drive detects a sequential access pattern, and activates its read–ahead mechanism. Then, the original requests take a small time so the algorithm decides that the cache cost is larger than directly reading data from disk, and the cache is deactivated. However, since in that state the requests are not sequential, the disk does not activate its read–ahead mechanism, and the original requests take more time, what makes the algorithm activate the REDCAP cache again.

It is also interesting to remember that the operating system does not detect this access pattern, nor does it implement any technique to enhance the performance under this type of access.

512 kB Strided Read

Let us continue discussing the REDCAP results obtained in the *512 kB Strided Read* benchmark. The results appear in Figure 2.19.

As we can observe, the results are analogous to those obtained for the same benchmark in the previous study of segment size. REDCAP has an identical behavior in both cache configurations and for all the file systems, but it does not perform better than the vanilla kernel. The worst results are usually achieved for 1 and 2 processes, and JFS is the file system where the degradation is more noticeable, but still pretty small, less than 5%. The problem is that the application time for 1 and 2 processes is rather small, and although the REDCAP cache is inactive all the time, the time initially lost when it is still active cannot be recovered later. For 4, 8, 16 and 32 processes, the loss can be considered negligible, and it is mainly due to the time employed to simulate the behavior of the cache when it is inactive. In all the cases, the algorithm turns the REDCAP cache off on the first chance and never turns it on again.

Remember that, as in the *4 kB Strided Read* benchmark, the operating system does not implement any prefetching technique for this access pattern.

Kernel Compilation for 4 processes

Since the *Kernel Compilation* benchmark is CPU–bound, here we focus our attention on the I/O time improvement achieved by REDCAP with respect to the original kernel. Figure 2.20 illustrates the improvement achieved for both cache configurations as a function of the file system.

In this test, our proposal significantly improves the I/O time, and the time reduction strongly depends on the file system. The REDCAP cache achieves improvements that range from 17% for Ext2 and the 128×128kB configuration to 2% for XFS and the 64×128kB one.

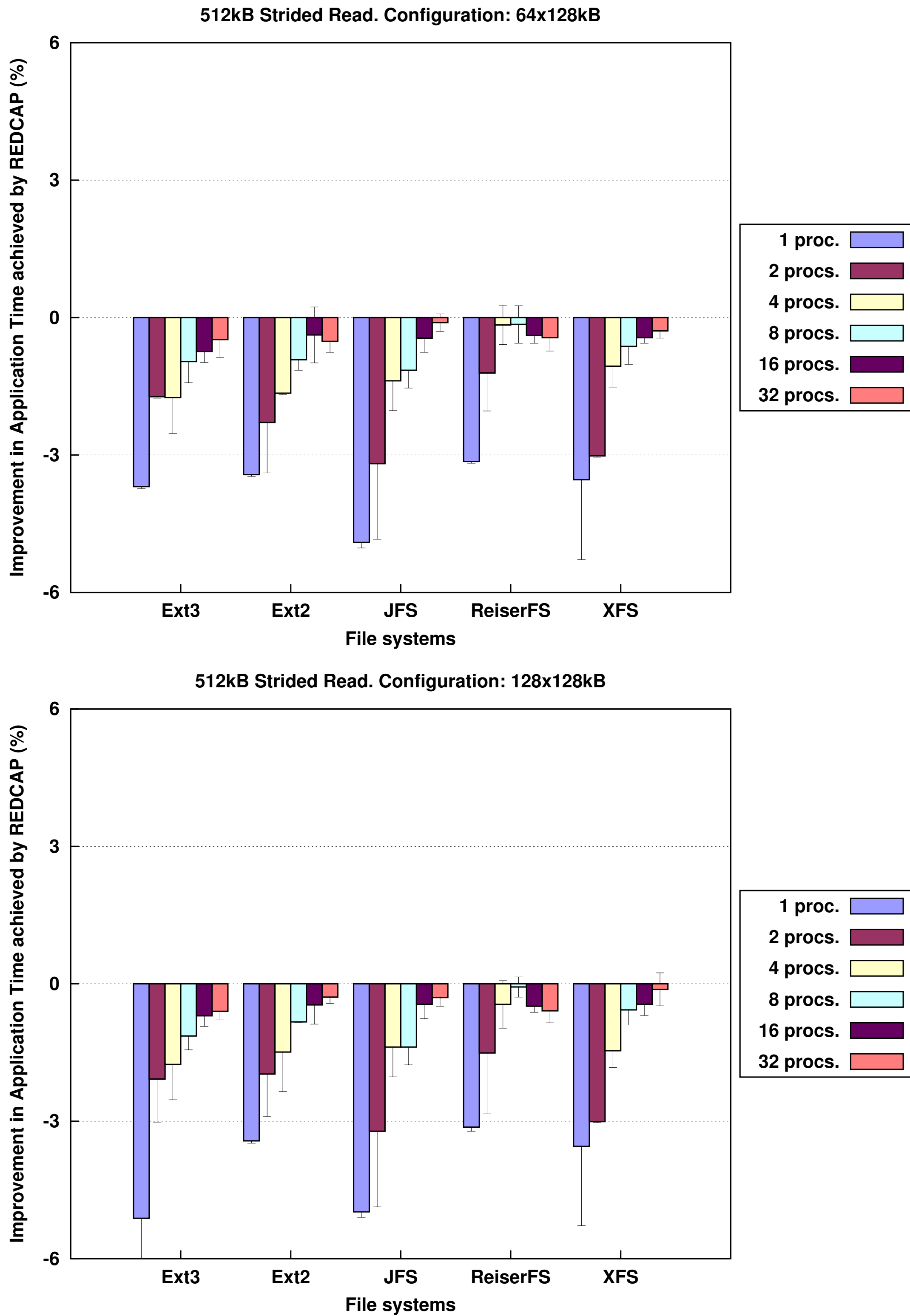


Figure 2.19: Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel for the *512 kB Strided Read* benchmark depending on the file system used. Note that the percentage values in the Y axis are small.

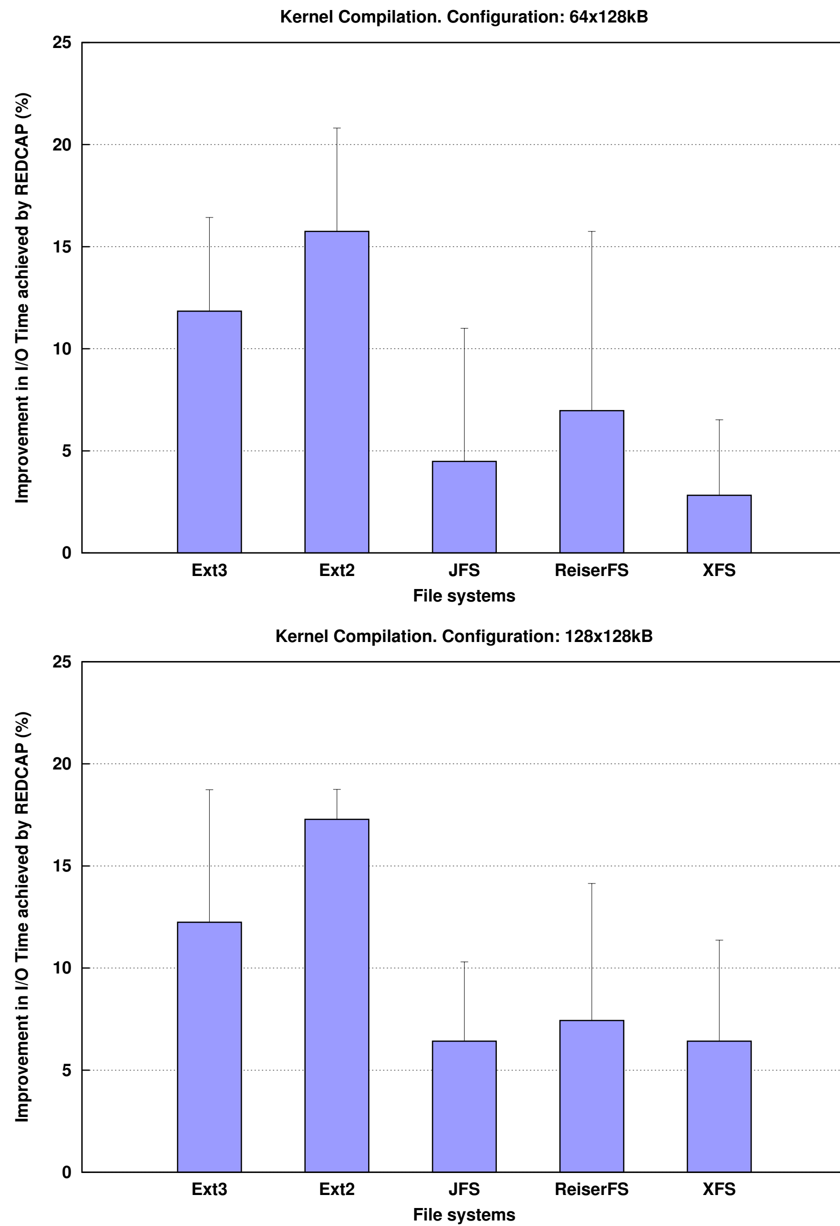


Figure 2.20: Improvement, in the I/O time, achieved by REDCAP over a vanilla Linux kernel for the *Kernel Compilation* benchmark depending on the file system used.

For Ext2, Ext3 and ReiserFS, both cache sizes show a similar behavior, however, for JFS and XFS, the results obtained with the 64×128kB configuration are slightly smaller than with the 128×128kB one. During the test, the cache is turned on and off several times. The percentage of requests served by REDCAP when its cache is active ranges from 31% (with ReiserFS) to 63% (with Ext3).

When comparing the results for Ext3 and the 64×128kB configuration with the results obtained by the first implementation for the same configuration (see Section 2.5.1), we can observe that a higher improvement is achieved now. This larger improvement is not due to REDCAP but to a new layout of the files on disk that, by chance, increases the REDCAP effectiveness (note that the file system was formatted and the tarballs containing the kernel files extracted again since the first set of experiments described in Section 2.5.1).

Again, we remark that the spatial locality, existing in the *Linux Kernel Read* benchmark, is partly lost in this test because during the kernel compilation files from different directories are used at the same time, and not all the files of a certain directory are read consecutively. Hence, despite the fact that all the files of the kernel source are also read in the kernel compilation, the corresponding results cannot be compared to those obtained by the *Linux Kernel Read* benchmark.

TPCC

Since *TPCC* is also a CPU–bound benchmark, we restrict our analysis to the I/O time improvement achieved by REDCAP compared to the original kernel. Histograms illustrating the results for the considered configurations and the five file system are plotted in Figure 2.21.

Again, the behaviors of the REDCAP and original kernels are very similar, and only an improvement of 2.8% is achieved for the Ext3 file system. For the other file systems, it does not perform better than the normal system, although the loss is rather small. JFS is the file system where the degradation is more noticeable, achieving the worst results with an increase in the application time of 6% for the 128×128kB configuration.

The *TPCC* benchmark has a random read pattern, that causes our cache to be ineffective, and it is almost impossible to take advantage of the prefetching performed. The activation–deactivation algorithm detects that and turns the REDCAP cache off, which is inactive a long time in all the tests.

2.6. Related Work

Since Maurice Wilkes proposed the cache concept in 1965 [67], there has been extensive and fruitful research for improving computing systems' performance by using several types of cache hierarchies. In this section we will only consider work related to disk caches, or techniques that take advantage of them.

There are several studies which describe the disk drives' operation and, analyze the impact of different caches and mechanisms on the I/O performance. Ruemmler and Wilkes [1] present a good summary of disk characteristics, and a description of how a disk drive works. They also describe in detail the read–ahead and write caching mechanisms of a disk cache. Shriver [49] gives a thorough description of disk caches, their parameters, their behavior, and the proposed read–ahead strategies. Furthermore, she even discusses the features of several disks, such as the replacement algorithm used or the read–ahead policy implemented.

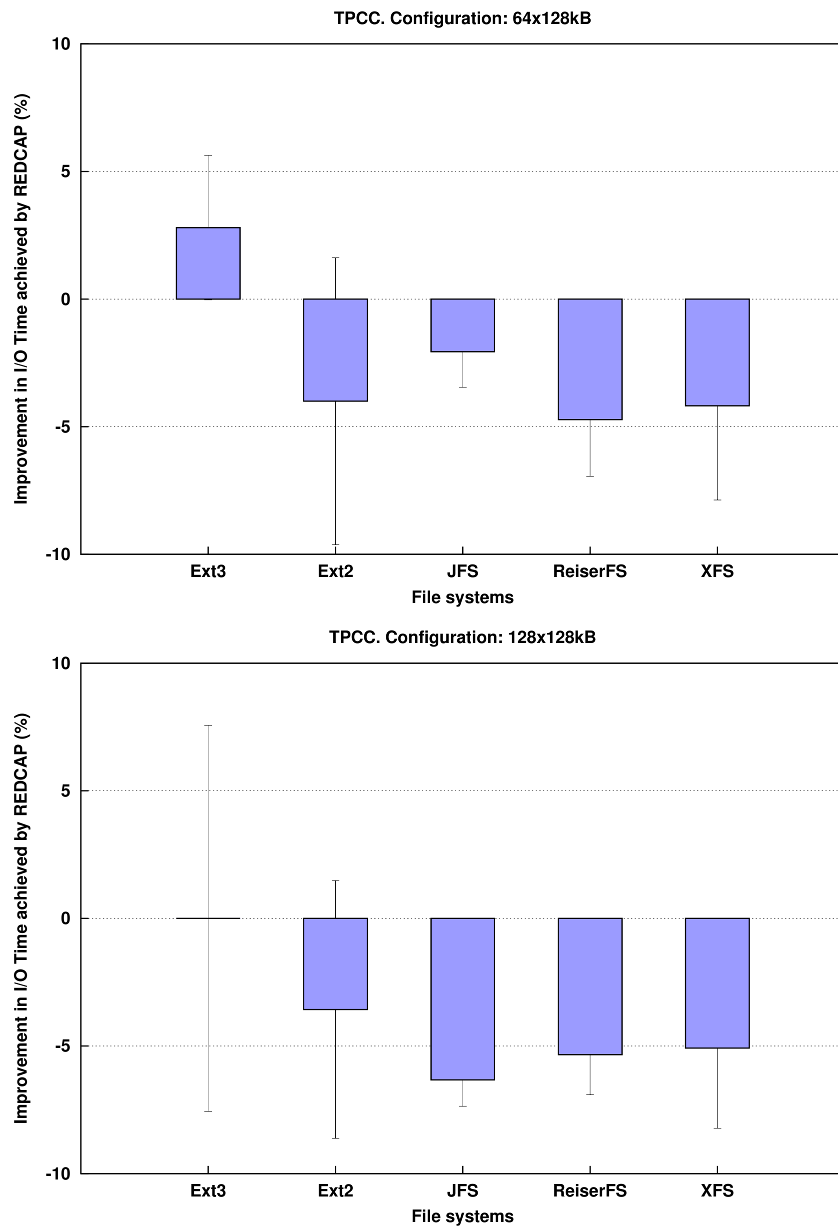


Figure 2.21: Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel for the *4 kB Strided Read* benchmark depending on the file system used. Note that the percentage values in the Y axis are small.

Karedla *et al.* [31] examine the use of caching as a means of decreasing the system response time and improving the data throughput of the disk system. They describe cache design parameters, caching strategies and cache replacement algorithms. They also investigate the performance of three algorithms: random replacement (RR); least recently used (LRU); and a frequency–based variation of LRU known as Segmented LRU.

Hsu and Smith [32] analyze the performance impact of various I/O optimization techniques. Among them, the most interesting ones for this study are: read caching; sequential prefetching; and opportunistic prefetching. They find that the most effective approach to improving I/O performance is to reduce the number of physical I/Os that have to be performed. They also suggest that a reliable means of improving performance is to use larger caches up to and even beyond 1% of the storage used. Our proposal implements these two ideas: we try to reduce the number of I/O operations by converting workloads with thousands of small requests into workloads with hundreds of large sequential read requests, and we effectively enlarge the disk cache by means of the REDCAP cache.

Hu and Yang [68, 69] present a *Disk Caching Disk* (*DCD*), a disk storage architecture which is aimed at optimizing write performance by using a small log disk as a secondary disk cache. The *DCD* is a hierarchical architecture consisting in three levels: a RAM buffer, a *cache–disk* that stores data in a log format, and a data disk that stores data in the same way as a traditional disk. The *cache–disk* is implemented either using a separate physical drive or a logical disk that is a partition of the data disk. The small RAM buffer collects small write requests to form a log which is transferred onto the *cache–disk* whenever it is idle. These data will be sent to the data disk afterward when the system is idle. It is important to remark that *DCD* manages read requests in the usual way, therefore its read performance is similar to a traditional disk.

Another disk cache architecture inspired in *DCD* and called *Redundant, Asymmetrically Parallel, Inexpensive Disk Cache* (*RAPID–Cache*) is presented by Hu *et al.* [70]. It consists of two redundant write buffers on top of a disk system. One of them is a primary cache (made of RAM o NVRAM) and the other one is a *backup cache* that consists of a small NVRAM buffer on top of a log disk (*cache–disk*). In the *backup cache*, the small and random writes are first buffered in the small NVRAM buffer to form large logs that are written into the *cache–disk* later in large transfers. As the *backup cache* is not involved in read operations, the read performance provided by RAPID–Cache is similar to that provided by a traditional disk.

Although REDCAP also introduces a new cache, just over the disk cache, its purpose is completely different to that of DCD [68, 69] and of RAPID–Cache [70]. Our new cache is implemented in RAM memory, and not in a log disk or NVRAM buffer. It is used for prefetching adjacent blocks from disk, and not to buffer modified blocks that will be transfer to disk later. Therefore, our proposal optimizes read requests, and not write operations. Moreover, unlike the experimental results presented for DCD and RAPID–Cache which are obtained by using a trace–driven simulation program, our proposal has really been implemented and evaluated inside the Linux kernel.

There are also several proposals that take into account the operation of the disk controller cache in order to improve the I/O performance. Worthington *et al.* [71] prove that algorithms that effectively utilize prefetching disk caches, like C–LOOK, provide significant improvements in workload with read sequentiality. They also propose two algorithms that take into

account the contents of a disk cache. One of them is *Shortest Positioning (w/Cache) Time First (SPCTF)*, which is a modification of *Shortest Positioning Time First (SPTF)* [72]. The other one is *Aged Shortest Positioning (w/Cache) Time First (ASPCTF)*, which is based on *Aged Shortest Positioning Time First (ASPTF)* [73]. In both algorithms, a positioning time of zero is assumed for any request that can be satisfied (at least partially) from the cache. However, the results obtained are not conclusive [74, 75], because the behavior of these two algorithms depends on workloads and on the type of system model used.

Chang *et al.* [76] present *CARDS*, a *cache–aware real–time disk scheduling* algorithm that also takes the on–disk cache into consideration during scheduling, helping to minimize the cache miss ratio. A simulation–based evaluation shows that *CARDS* is highly successful compared to classical real–time disk scheduling algorithms.

Two management techniques for the disk controller cache have also been proposed by Carrera and Bianchini [77]. The first one, *File–Oriented Read–ahead* (*FOR*), adjusts the number of read–ahead blocks brought into the disk cache according to the file system information. This technique determines that, on each disk access, a block should only be read–ahead if it belongs to the same file as the block that was actually accessed. Therefore, the disk controller requires information about the file layouts on disk. The second technique, *Host–guided Device Caching* (*HDC*), gives the host direct control over part of the disk controller cache. HDC is based on the observations that servers normally use disk arrays, and each disk has its own cache. The host processor integrates the management of the host and disk controller caching. As an example, they propose that a set of disk controller permanently caches the files that cause most misses. Their results show that both techniques can significantly increase disk throughput.

The main drawback of the above proposals is that the experimental results are based on simulations. This is because the disk manufacturers do not currently provide a means of externally managing the cache of the disk controller. Our proposal, however, emulates the disk cache by using a small amount of main memory which allows us to present a real implementation. Furthermore, they need to modify the disk controller to implement their proposals, whereas REDCAP does not need to perform any modification to them. Finally, the FOR technique only prefetches blocks that belong to the same file, but our prefetching technique does not take into account the layout of files on disk.

An adaptive prefetching mechanism for disk caches is proposed by Grimsrud *et al.* [78]. They maintain information about the order of past disk accesses in an adaptive table that stores the most probable successors for each disk block. Each successor is tagged with a weight which indicates the likelihood that it will be referenced given that its parent is referenced. The table and the associated weight are used for predicting future access sequences and to control the prefetch mechanism. Their results show that its prefetching technique can reduce the average time to serve a disk request significantly. It is interesting to note that to implement the algorithm they need to know several parameters, such as branch factor, look–ahead level, weight ceiling, weighting function, fetch threshold, etc. A similar approach is presented by Zhu *et al.* [79]. They caching algorithm that uses an adaptive prefetching scheme to optimize the system performance in disk controllers. Again, a table structure is needed to store information about the next most probable disk access for each disk block. They use on–line measurements of disk transfer times and of inter–page fault rates to adjust the level of prefetching dynamically. Their results show the effectiveness and efficiency of their proposal.

As before, the problem of these two proposal is that the experiments have been performed in a simulation model, and the proposed algorithms have not been implemented on any machine, whereas REDCAP has been tested and implemented inside the Linux kernel. Furthermore, the size of the tables is significantly large, specially given the capacity of modern disks. Note that REDCAP takes advantage of the prefetching performed by the disk cache, and does not try to modify its behavior.

Operating systems also incorporate some kind of caching (usually, a buffer cache) and prefetching in their file systems. Prefetching during sequential access patterns is the most common technique implemented. For example, Linux implements a simple mechanism that prefetches file blocks when it detects a sequential access to a file [77, 51]. It adapts the number of disk blocks to be prefetched according to the sequentiality of the accesses to a specific file. Other more sophisticated mechanisms are also possible. This is the case of the prefetching method proposed by Lei and Duchamp [53], that prefetches entire files by taking into account file access patterns. Its technique builds semantic structures that capture the interrelationships between file accesses, in such a way that it can make predictions of future file accesses. Another example is the file system proposed by Cao *et al.*, called ACFS (Application–Controlled File System), that integrates application–controlled caching, prefetching and disk scheduling [80]. They use a two–level cache management strategy: a policy allocates blocks to processes, and each process integrates application–specific caching and prefetching. Note that our technique is independent of the prefetching already implemented by the operating system and/or application.

The hardware and software caches create a memory hierarchy which can cause some inefficiencies, and most of the times they will have duplicate blocks among them. Wong and Wilkes [54] propose that storage caches should be made exclusive, i.e., a data block should be only cached at one level of the cache hierarchy, helping to create the effect of a single and large unified cache. They introduce a *DEMOTE* operation to transfer data ejected from an upper level cache to a lower level one. Their results show that they can obtain useful speedups. However, the implementation of their proposal in a real system requires modification to source code of client software to explicitly use the new operation.

There are after studies and proposals about exclusive caching in the literature, but, for the sake of simplicity, we only highlight two of them. Chen *et al.* [81] generalize the idea of an exclusive cache and present an eviction–based cache placement policy for storage caches by obtaining eviction information from client buffer caches without modifying client applications. Their approach also introduces great improvement in cache hit ratios, but they only use two type of workloads in their studies. Karma [82] is also a management policy for multiple levels of cache. It leverages application hints to make informed allocation and replacement decisions in all cache levels, preserving exclusive caching. It also introduces significant improvements, but applications should be modified in order to provide this information. In our proposal, the idea of *exclusive caching* is used whenever it is possible, although we do not implement a *demote* operation or a control of evicted blocks from an upper level cache.

Soloviev studies the performance of a multi–disk storage system equipped with a segmented disk cache processing a workload of multiple relational scans, and she observes problems of load imbalance [83]. She investigates several approaches to ensure load disk balancing, such as partial declustering of files across disks, a load control of the maximum number of jobs that can be concurrently executed in the system, or an adaptive cache segmentation policy.

However, her most interesting proposal is the use of main memory to prefetch disk blocks, supplementing or replacing disk cache prefetching. She proposes an algorithm that limits the total number of scans in the system that prefetch in disk cache. This limit is fixed to the expected total number of disk–prefetching scans that can be handled by all disks of the system without overloading. Up to this limit, scans are permitted to prefetch in disk and memory at the same time, and the remaining scans are served by prefetching in main memory. This method transfers five pages at a time: one is the requested page and four are the prefetched ones. Although the idea of prefetching in main memory is quite similar to the main idea of REDCAP, this proposal has important differences with our approach and even several drawbacks. The proposed mechanism needs to control and manage the disk cache to be able to decide which blocks have to be cached and which not, but, currently, disk caches work as a black box and it is not possible to impose any control or management from outside. To test its mechanism, she only uses workloads made up of sequential scans, although with different inter–arrival intervals, whereas our proposal has been tested with different kinds of workloads covering several access patterns. Her best results are obtained when prefetching in main memory is performed for each scan. However, as our experimental results show, blind prefetching can downgrade the throughput of the system since some prefetched blocks could be never used. Furthermore, she does not implement any control of the improvement achieved, but REDCAP has its own activation–deactivation algorithm that makes its prefetching dynamic by controlling its performance and the improvement achieved. She uses a simulation model to make this study, but REDCAP has been implemented and tested inside the Linux kernel. Finally, she claims that memory prefetching makes disk–level prefetching practically useless. However, we disagree with this statement, since disk cache prefetching provides significant performance improvements for workloads with read sequentiality and, therefore, it speeds up the requests that perform the prefetching.

Zhu *et al.* investigate the impact of a disk cache on file system response times when the file system buffer cache becomes larger [84]. By using an analytic model, they study the performance of the disk cache by using different sizes of both the disk cache and the file system buffer cache. The disk cache size varies from 512 kB to 16 MB, and the file system buffer cache size from 16 MB to 128 MB. They conclude that large disk caches will not significantly improve the overall system performance, because the operating system already use large I/O buffer caches to cache reads and writes. However, from our point of view, this comparison between the disk cache and the buffer cache is not fair, because, although both caches have the same goal (decreasing I/O time), they work at different levels. Disk cache improves I/O performance whenever it can directly satisfy I/O requests without accessing the disk. A study varying features of the disk cache (number of disk cache segments or the read–ahead size) will prove its efficiency and performance. Indeed, they also study the impact of segment numbers and obtain a decrease in the response time of 65% when the request size takes 1 kB. Moreover, their claim will have more sense if the operating system would always issue large requests, but operating systems like Linux does not behave this way. Actually, many small requests from different applications are dispatched to disk, and disk drives with small caches and few segments are not able to efficiently manage these workloads, because read–ahead blocks will be evicted before being served. In these cases, large disk caches with hundreds of segments can treat more I/O streams at the same time and provide a much better throughput.

HPCT–IO is an application–based caching and prefetching technique proposed by Seelam *et al.* [85, 86, 87]. They maintain a file–IO cache in application address space, which has to be configured by the user by specifying the cache size as well as the page size. The application file–I/O calls are re–directed to libraries that analyze the traces for access patterns, and initiate the prefetching of subsequent blocks of a file into the cache. When the application submits an I/O request, HPCT–IO processes this request: if it is a cache hit, the data is returned immediately; otherwise, it will issue new requests of cache page size to read the requested data. After the data is returned to the application, HPCT–IO tries to perform forward or backward sequential prefetchings. They have implemented their solution on a Blue Gene/P System, and their results show that HPCT–IO improves the I/O bandwidth utilization and reduces the file I/O latency. Although their cache plays a role similar to that carried out by the REDCAP cache, this proposal has important differences with REDCAP and even several drawbacks. HPCT–IO is application centric, and has been implemented as a user–space library that has to be linked with the application. Since their cache is associated with a file, they only perform file prefetching, but no metadata prefetching. Furthermore, the user has to provide information about the cache size and the page cache size. On the other hand, our approach is part of the operating system. REDCAP neither requires the modification of the applications nor the user's configuration to perform the prefetching. The REDCAP cache is implemented inside the Linux kernel, and prefetchs both data and metadata blocks, but only on cache miss.

Finally, the idea of a disk cache that turns itself on and off dynamically depending on the performance achieved was suggested by Smith [88]. According to Smith, both the cache miss ratio and write ratio would be used for deciding on the state of the disk cache. It is important to note that the idea was not tested, and no algorithm was developed to specify when the cache should be enabled or disabled. REDCAP, however, implements a real mechanism to turn the cache on and off dynamically, and it uses a different metric to control the change. Moreover, some of our initial tests showed that the cache miss ratio, as proposed by Smith, is not always a good metric, especially for sequential reads. Smith also suggests the possibility of using the main memory for the disk cache, *eliminating* the controller cache, and using a mechanism that is similar to the mapped files in memory. Nevertheless, our cache does not replace the disk cache of the drive but instead *takes advantage* of it.

2.7. Conclusions

In this chapter we have introduced REDCAP, a RAM–based disk cache which is able to greatly reduce the I/O time of disk read requests by using a small portion of the main memory. REDCAP consists of three parts: a cache, a prefetching technique, and an activation–deactivation algorithm. The cache works as an extension of the built–in cache of the disk drive. The prefetching technique reads in advance consecutive disk blocks in such a way that it takes advantage of the read–ahead performed by the disk cache. The algorithm controls the performance achieved and makes the prefetching dynamic.

REDCAP has several features which make it unique. First, it is I/O–time efficient, since takes advantage of the disk read requests issued by the application in order to prefetch adjacent disk blocks. Second, it converts workloads with thousands of small requests into workloads with disk–optimal large sequential requests. Third, it implements an activation–

deactivation algorithm which makes it dynamic. The algorithm is quite simple, although has proved to be very effective for a wide range of workloads. And fourth, REDCAP is independent of the underlying device. The activation–deactivation algorithm does not take into account any of the physical characteristics of the disk, it only uses the I/O times obtained during the normal system operation.

We have performed two different sets of experiments: one to analyze the behavior of RED-CAP and the importance of its segment size, and another one to evaluate the impact of both the file system and the cache size.

Our first set of experiments studies the REDCAP behavior by using four different configurations of its cache, all of them with the same total size, and several workloads. The results prove that the REDCAP behavior strongly depends on the segment size, i.e., the prefetching size, and that the application time decreases with large prefetching sizes.

Although with segment sizes of 32 kB and 64 kB, our technique presents the smallest improvements, they can reach up to 54% and 73%, respectively. However, for some access patterns, REDCAP does not achieve a better improvement because the prefetching size is small, and its results are equivalent to those obtained by a vanilla Linux kernel.

On the other hand, with segment sizes of 128 kB and 256 kB, REDCAP presents the highest improvements. Indeed, it can improve the performance up to 80% for some workloads, while achieves identical results to that obtained by a normal system for workloads where an improvement in the I/O time is hard to obtain. Both segment sizes get a quite similar performance. Nevertheless, the 128 kB size presents, in general, the best behavior. In fact, the 256 kB segment size only gets the best result for a backward access pattern.

The obtained results show that, with a small portion of the main memory, our approach is able to considerably reduce the I/O time of the disk read requests.

The second set of experiments analyzes our proposal under five different Linux file systems (Ext2; Ext3; JFS; ReiserFS; and XFS), two cache sizes, and different workloads. The experimental results show that, although REDCAP is able to obtain its maximum performance with all of them, the improvements achieved depend, to some extent, on the file system used.

The best results are achieved for those file systems which divide the disk into several groups, such as Ext2; Ext3; JFS; and XFS. These groups produce data locality which is exploited by the prefetching mechanism of REDCAP. For these file systems, even more than an 80% reduction is achieved.

The block allocation policy of the file system also affects the improvements achieved. This is the case with JFS, that creates single–extent files even when the files are created in parallel in the same directory. These single–extent files are ideal for the disk controller read–ahead and Linux kernel block prefetching mechanisms in some workloads, where the benefit provided by REDCAP is unavoidably small.

We also observe that ReiserFS, which does not split the disk into groups, produces requests that are apparently random for REDCAP. The large disk print caused by these requests makes many cache segments be evicted before being re–used, limiting the effectiveness of the REDCAP cache for a large number of processes. Despite these problems, an improvement of more than 55% is achieved.

The results obtained for different REDCAP cache sizes are very similar, because the number of processes performing I/O operations is never larger than the number of REDCAP segments in our benchmarks. This study suggests that the REDCAP cache size should be dynamic

and dependent on the number of concurrent read streams in the workload.

Finally, to conclude, the experimental results have proven that our proposal significantly improves the I/O time of disk read requests, for many workloads and any file system. It also achieves similar results to those obtained by a vanilla Linux kernel for those workloads which have a random access pattern, or perform large sequential reads.

Chapter 3

In–Kernel Disk Simulator

This chapter presents a framework for simulating the performance obtained by different I/O system mechanisms and algorithms at the same time, and for dynamically turning them on and off, or selecting between different options or policies, in order to improve the overall system performance. A key element of this framework is the design and implementation of a disk simulator inside the Linux kernel, called *virtual disk*. Our virtual disk creates a virtual block device that is able to simulate any disk drive with a negligible overhead, taking into account possible dependencies among requests, and without interfering with regular I/O requests.

In order to implement the virtual disk, the storage device is modeled by using a dynamic table that takes parameters features of a given request as input and returns the I/O time needed to serve it. The table is dynamically updated with the I/O times calculated while the real disk serves requests.

We also describe the potential utility of the framework in REDCAP. Depending on the REDCAP state, the virtual disk simulates the behavior of the real disk in a "normal system", or in a "REDCAP system". Thus, the activation–deactivation algorithm (described in Section 2.3.3) has been improved, and now controls the performance by comparing the service times of the real and virtual disks.

The organization of this chapter is as follows. The first section motivates the problem and presents the aim of our proposal. The in–kernel disk simulator is presented in Section 3.2 by introducing its design and implementation. Section 3.3 describes how the virtual disk is used in REDCAP. The hardware platform, benchmarks and I/O schedulers used in the experiments are described in Section 3.4. Section 3.5 evaluates the in–kernel virtual disk by analyzing the accuracy of the disk model and by studying the behavior of REDCAP with our simulator when the storage device is a traditional hard disk. In Section 3.6, we discuss how the simulator can be used with SSD disks, by analyzing the accuracy of the model and the REDCAP performance for these devices. Section 3.7 contains a brief description of previous work related to the proposed technique, specially related to disk simulators and disk models. Finally, Section 3.8 concludes the chapter.

3.1. Motivation

Over the years, advances in disk technology have been very important, and vast improvements in disk drives have been made. However, the disk I/O subsystem is still identified as the mayor bottleneck for system performance in many computer systems, because the mechanical

operations of the disk considerably reduces its speed as compared to other components, such CPU and main memory [3]. Solid–State Drives (SSDs) do not suffer the mechanical problem, and obtain a performance much better than that provided by hard drives. However, they are still expensive and small in capacity as compared to traditional disks, and their speed is still far away from the bandwidth achieved by other system components, such as CPU, main memory, and GPU. Hence, the efficiency and throughput of I/O intensive applications extremely depend on the response time of the secondary storage.

There are several mechanisms that play an important role in the performance achieved by the I/O subsystem: page and buffer caches of the operating system; built–in cache (*disk cache*) of the hard disk drives; prefetching carried out by the operating system and the disk drive itself; I/O schedulers; etc. All these mechanisms can significantly reduce I/O time, although they are not optimal in the sense that improvement that they provide depends on the current workload, disk drive, file system, and so on. Moreover, all of them usually have a worst–case scenario that could downgrade I/O performance when the mechanism is active. For instance, for workloads where data is accessed only once, a buffer cache does not achieve any benefit from keeping the data in main memory, and it can even downgrade the system performance because of the eviction of pages of running processes to disk. Another example is the prefetching performed by an operating system or disk drive, that can improve the performance of a sequential access pattern, but can also downgrade the performance of a random one.

Therefore, it could be a good idea to be able to activate/deactivate a mechanism, or to change from one mechanism or algorithm to another depending on the workload and the expected performance. In order to achieve this dynamic behavior, we need a means to evaluate several I/O strategies at the same time. Obviously, only one of them could be active at a specific moment, whereas the rest should be simulated.

As a first step to implement such a general system–wide simulation, we present the design and implementation of an in–kernel disk simulator which fulfills the above requirement. Our simulator is implemented inside the Linux kernel by creating a *virtual disk* that it is able to capture the behavior of a disk drive. The virtual disk simulates part of the I/O system by working as both a block device driver and a disk drive. It has its own I/O scheduler to decide the order of the incoming requests in the queue and when each request is dispatched to the "virtual" device.

Disk drives are quite complex systems that have intelligent caching and prefetching algorithms, complex data layout with zone division, bad sector management, and so on. These complex features and mechanisms are hidden to the operating system and its components by the disk controller, which provides a quite simple interface: read and write operations to blocks organized in a linear array. This simplified interface allows operating system components, such as file systems, to focus on high–level aspects, such as, metadata and block layout, extent–based allocations, consistency management, etc. Due to the complexity of modern drives, disk modelling becomes a challenge, and, although Ruemmler and Wilkes introduces a disk model which takes into account almost all the main features [1], it is not an easy task to build one able to capture all these complexities. For this reason, we have designed a simple disk model that gives reliable estimations of I/O times, without performing an exact simulation.

The simulation of I/O strategies introduces a problem: the order in which I/O requests

are submitted depends on the active I/O mechanisms because, when a request is served, the application that issued the request is waken up and can submit new requests, whereas other applications are still blocked on their own I/O operations. Therefore, to properly simulate a different I/O strategy, the disk simulator can not use the request order produced by the active I/O mechanisms. However, at the same time, requests of a process should be served in the same order as they were submitted by the process, and the disk simulator has to control this order. Thus, the virtual disk must generate a new simulated arrival order of the I/O operations by taking into account dependencies among them and controlling the processes that have submitted them.

3.2. In–Kernel Disk Simulator

In an operating system, a block device driver and a disk drive work together to attend disk I/O operations. The driver provides access to the drive by acting as an interface between the system and the device. In the simplest sense, the block device driver receives I/O requests issued by applications, and dispatches them to the disk device. Once the device gets the requests, it performs the corresponding I/O operations.

In a Linux system, the above works as follows. Firstly, during the startup process, the block device driver creates and initializes its data structures and scheduler queue. Once this phase is finished, the block device driver is ready to receive requests and to dispatch them to the disk drive, effectively starting the processing of I/O requests.

The processing of a request can be summarized in three steps: *request* function; disk operation; and *request completion* functions [89]. The *request* function, which is the core of every block driver, gets the next request of the I/O scheduler queue, and issues it to the disk device. The disk device processes the request and performs the corresponding operation. When the operation is finished, the drive raises an interrupt to inform the block device driver, and, as a consequence, the *request completion* functions are called to manage the completion of the I/O operation. After that, the *request* function is invoked again and the same process starts over. When there are no more pending requests to serve, the disk device is *plugged*, waiting the arrival of new I/O operations.

We have implemented a disk simulator inside the Linux kernel to reproduce part of the I/O subsystem [35]. The new in–kernel disk simulator is created by a piece of code that works as both a block device driver and a disk device. It has its own request queue and its own I/O scheduler to sort incoming requests before being served, and simulates the behavior of a disk drive. The simulator is called *virtual disk*, to short. Figure 3.1 depicts the whole I/O process followed by the virtual disk.

Our virtual disk receives *virtual requests* to serve. These requests are a copy of those submitted to the real disk: before inserting a request into the I/O scheduler queue of the real disk, the system creates a new virtual request with the same basic parameters (LBA sector, size, and type) of the real one. The virtual request is then inserted into the disk simulator, from where it is processed. Real requests, without any modification, are queued in the scheduler of the real disk to be served.

The disk simulator uses different auxiliary queues to deal with the virtual requests before inserting them into the scheduler queue of the virtual disk. Once in the scheduler queue, virtual requests are handled similarly to the way real requests are handled. Moreover, the

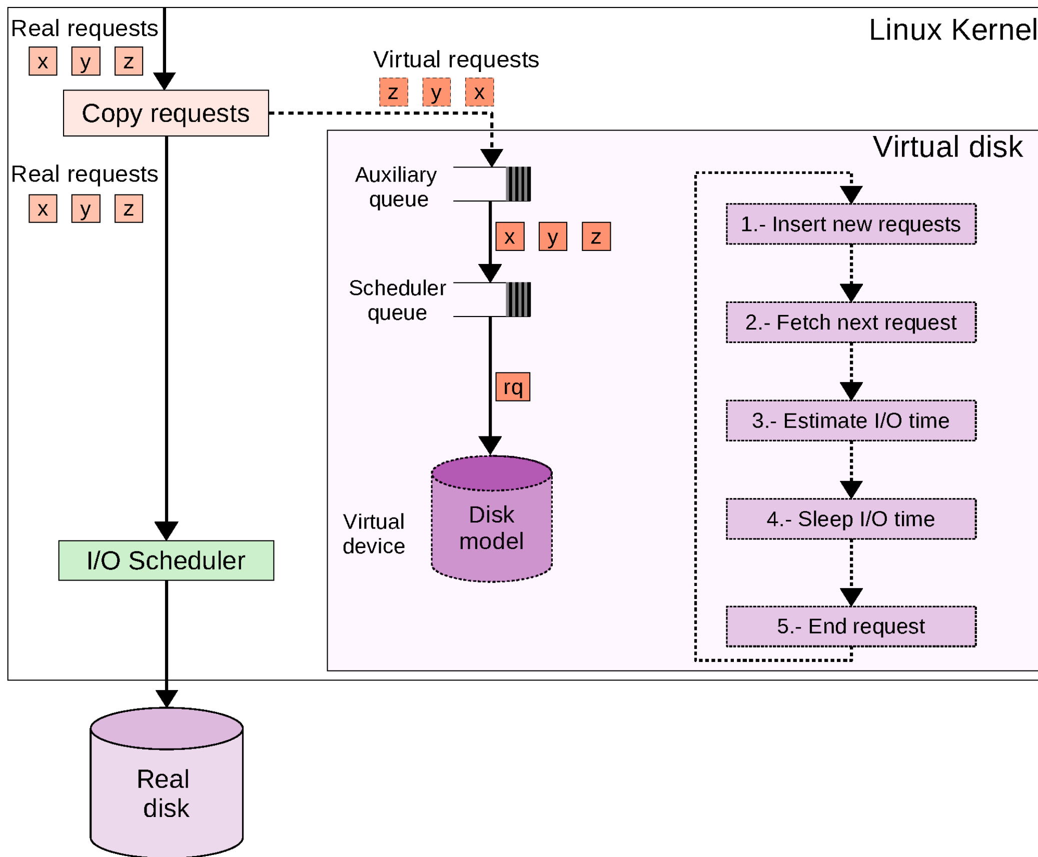


Figure 3.1: Virtual disk routine.

simulator controls dependencies between requests (such as dependencies between request arrivals and previous request completions), because the virtual requests of a process have to be served in the same order as they were issued.

Our simulator creates a kernel thread that entirely performs the above described process. Its first task is the initialization phase: the virtual disk itself is created and initialized, and its data structures, auxiliary queues and scheduler queue are allocated and initialized.

Once this phase has finished, processing of requests begins. The kernel thread runs a routine that implements both a disk driver and a disk unit by executing the three mentioned steps, and by also modeling the arrival of requests. The routine performs the following actions:

1. Move the virtual requests from the auxiliary queue to the scheduler queue by simulating their arrival to the block device driver. This step corresponds to the insertion process performed by processes that issued I/O requests.
2. Fetch the next request from the scheduler queue. The I/O scheduling policy assigned to the virtual disk will pick up the next request to dispatch. This step simulates the *request* function.
3. Get the estimated I/O time needed to attend the request from a table–based model of the real disk.
4. Sleep the estimated time to simulate the disk operation is being performed.

This step and the previous one “execute” the disk operation.
5. Finally, after waking up, complete the request, and delete it from the scheduler queue. The *request completion* functions are represented by this last step.

The kernel thread runs continuously these steps until there are no more pending requests to serve. Then, as a regular disk, the virtual disk is plugged until new requests arrive. When a new request is copied by the system, the virtual disk is unplugged, and the routine starts again.

It is worth emphasizing that the sleeping performed in step 4 to simulate the disk operation is needed to make a right simulation. Although another option would be not to wait this time and just to count it and then serve the next request without waiting, that would prevent the virtual disk from simulating the effect of the scheduler in the arrival order of the requests. Furthermore, those schedulers, such as CFQ and Deadline, that take into account the elapsed time in order to decide which requests from which process have to be dispatched, should be modified to properly work with the in–kernel disk simulator.

In addition, it makes the computation of the service time of each request easier: if the waiting were not done, I/O time of a request would have to be added to all the pending requests in the scheduler queue (see Section 3.2.4).

The next sections describe different aspects of our in–kernel disk simulator in detail. The disk model to simulate the behavior of a real disk is described in Section 3.2.1. Section 3.2.2 presents the I/O schedulers that can be used in the virtual disk to sort incoming virtual requests. The request management and the control of dependencies are described in Section 3.2.3. Section 3.2.4 discusses the method implemented to compute the service time of virtual requests, time that is needed to compare the throughput of the real and virtual disks. How the tables are trained is explained in Section 3.2.5. The method to avoid the congestion of the I/O scheduler appears in Section 3.2.6. Finally, Section 3.2.7 summarizes the operation of the virtual disk.

3.2.1. Disk model

A disk model is needed to capture and simulate the behavior of a storage device. The disk model has to determine, given an I/O request, its disk access time, i.e., the I/O time to serve the request.

An accurate disk model is essential for performing a good and precise simulation, but, to achieve this goal, too many parameters have to be considered. Ruemmler and Wilkes claim that aspects such as the seek time curve, settle time, head and track switches, transfer rate, disk cache, and data layout should be taken into account, although other aspects such as soft–error retries, or effect of spared sectors or tracks could be omitted [1]. The problem is that not all the parameters can be obtained from the technical specification of the disk. In fact, some of these parameters are even considered trade secrets, and details about them are not provided by the disk manufacturers. It is interesting to note that several researchers have proposed methods to directly extract information from hard disk drives (for example, data layout or seek curve [45, 90, 50, 91, 92]), and even to extract performance parameters of SSD drives [93]. However, to the best of our knowledge, the source code of these mechanisms is not publicly available, and to construct one of these extraction tools is not an easy task [94, 95].

We should also consider that the more detailed the model, the costlier to run it. And, although a complete and accurate disk model can give reliable estimates of a disk performance,

its computation time can be too large to be useful in an on–line simulation. Nevertheless, since our disk model is going to be run inside the Linux kernel, it cannot be too complicated to provide an on–line decision support for performance predictions. Hence, our goal is to build simple disk model with a high degree of accuracy to allow us to perform on–line evaluations.

We propose a dynamic table to model storage devices [35]. Given a request, the resulting table–based model takes a series of parameters as input, and returns an estimation of the I/O time needed to attend the request. As we will see in Section 3.2.5, the table of our disk model is trained by means of requests served by the real disk, without taking into account any disk–specific features. Therefore, our approach will be able to model the behavior of any disk drive that could be used in practice, including SSD drives.

In the following sections we describe, in detail, the input parameters of the disk model, the structure of the table, and its dynamic behavior.

Input parameters

The time taken by an I/O operation may depend on several factors: the type of the request, its size, the state of the disk cache, the distance from the previous operation, the data layout (due to the zone division of disks), the immediately previous request's characteristics, and so on [1, 45]. However, for simplicity reasons, our table–based model only uses three of these parameters as input to predict the I/O time for the current request: the type of the request (read or write), its size, and the *inter–request distance* from the previous request. Section 3.5.1 shows that these three parameters, along with the dynamic behavior of the table, are enough to accurately model the real disk.

Regarding the request type, read and write operations take different times to execute [1, 45]. Disk drive attempts a read as soon as the disk head is supposed to be near the right track. If the settle has not been complete and the read is wrongly done, it has to be repeated. Obviously, write operations will not be carried out if the disk head is not correctly settled. Moreover, as we mentioned in Section 2.1, the disk cache also affects the I/O time required to perform an I/O operation. Each time that a read request can be "directly" satisfied from the disk cache, without seeking data and reading it off the disk, a shorter I/O time is needed. However, not all read operations are cache hits. On the other hand, since a write–back policy and an immediate reporting are usually used in disk caches [49], write requests are considered "done" as soon as they are in cache. For these reasons, the proposed model manages two tables:

- a read table that predicts the I/O time of read operations;
- a write table that predicts the I/O time of write operations.

The type of the last operation dispatched by the disk also influences the I/O time of the current one. We have not, however, considered this parameter. The reason is because, on a Linux system, write operations are deferred in the page cache and then periodically flushed to disk by the `pdflush` thread [51, 42, 52]. `pdflush` flushes dirty pages to disk when the amount of free memory in the system shrinks below a specified level, trying to relieve low–memory conditions. In addition, it periodically wakes up and writes out dirty pages by ensuring that no dirty pages remain in memory indefinitely. Some I/O schedulers also give higher priority to read operations. They usually delay writes operations for a small time interval, so they can merge write requests together, and dispatch large requests to disk. Thus, we assume that

I/O operations occur in bursts, alternating long read bursts with long write bursts, and that the type of the last and current operations will be the same, with a high probability.

We have considered the request size as a parameter for a couple of reasons. The first one is that transfer time is proportional to the length of a request [1], especially for small inter–request distances for which transfer time is the dominant factor in I/O time. For large distances, the dominant factor is usually the seek time, whereas the importance of transfer time is reduced. The second reason is that our model indirectly takes into account the disk cache, and I/O time greatly depends on the request length in a cache hit. To illustrate the importance of the request size, we have analyzed its impact by comparing the predictions of the proposed model with the predictions of the same model but without taking the size of the requests into account. The results show that the size is fundamental for the accuracy of our model (see Section 3.5.1).

Worthington *et al.* [45] define the inter–request distance as the rotational distance between the physical starting locations of two requests. When both requests start on the same track, they give the value in number of sectors. Otherwise, they measure it as the angle between the first sectors of the request pair, or as the time necessary for the disk to rotate through this angle. However, since our disk model does not take into account the data layout of the disk, we define the inter–request distance as the logical distance (i.e., number of blocks) from the last block of the previous request to the first block of the current one. Note that, for an I/O operation, seek time is a function of the distance in cylinders with the previous request [1, 45]. Furthermore, Popovici *et al.* [46] claim that the logical distance between two requests and the request type are sufficient for predicting positioning time. Thus, we can consider that seek time is a function of the inter–request distance from the previous request.

To conclude, the three input parameters of the model are devoted to select the either the read or write table of I/O times (request type) and address the table (request size, and inter–request distance).

It is interesting to realize that the I/O time of a request is measured as the sum of the seek time, settle time, head and track switch times, rotational delay and transfer time [1]. However, the proposed disk model does not independently simulate each physical component of the disk drive, and it is not possible to determine what fraction of an estimated time comes from each one, although we believe that the model is indirectly taking into account all of them. The seek time (that we also consider that includes the settle and switch times) is represented by the inter–request distances which appear in the columns of the table. The transfer time is considered by means of the request sizes in the rows. Finally, an average rotational delay is included in the estimated I/O times, because each cell of the table keeps the I/O time average of several samples, and each request served by the real disk has a different rotational latency.

Table structure

Both tables have the same structure, the only difference is the kind of values they hold: I/O read times, or I/O write times.

Rows of the tables represent request sizes. Each table has thirty two rows that mean request sizes from one block (4 kB) to 32 blocks (128 kB). These sizes have been selected because they are the minimum and maximum request sizes allowed by the file system. Note that, in a real system, the scheduler can merge several requests into one request which can be larger than 128 kB. However, since these large requests are unlikely, we have not considered these

cases. When a request larger than 128 kB is submitted, the disk model uses the last row to estimate its I/O time.

Columns represent inter–request distances. However, given the large capacity of a modern disk, the number of all the possible inter–request distances is unmanageable. For this reason, we have assigned ranges of distances to each column, so the total number of column is determined by the disk capacity. The distribution of the inter–request distances among the different columns has been the following:

- The first column corresponds to an inter–request distance of 0 kB.
- From the 2nd to the 19th column, each cell stores values for small inter–request distances in such a way that the behavior of the disk is simulated with a higher precision, and even the effect of its cache and read–ahead mechanism is simulated. The $n-th$ column, with n from 2 to 19, stores values for requests with inter–request distances from $4 \cdot 2^{n-2}$ kB to less than $4 \cdot 2^{n-1}$ kB.
- The rest of the columns corresponds to the largest inter–request distances. The 20th column corresponds to inter–request distances from 1 GB to less than 2 GB, the 21st from 2 GB to less than 3 GB, and so on.

We would like to remark that the first column, that means contiguous operations, represents some disk cache hits. The first columns after the first one (the number depends on the size of the disk cache), that imply small seeks among requests, also represent cache hits.

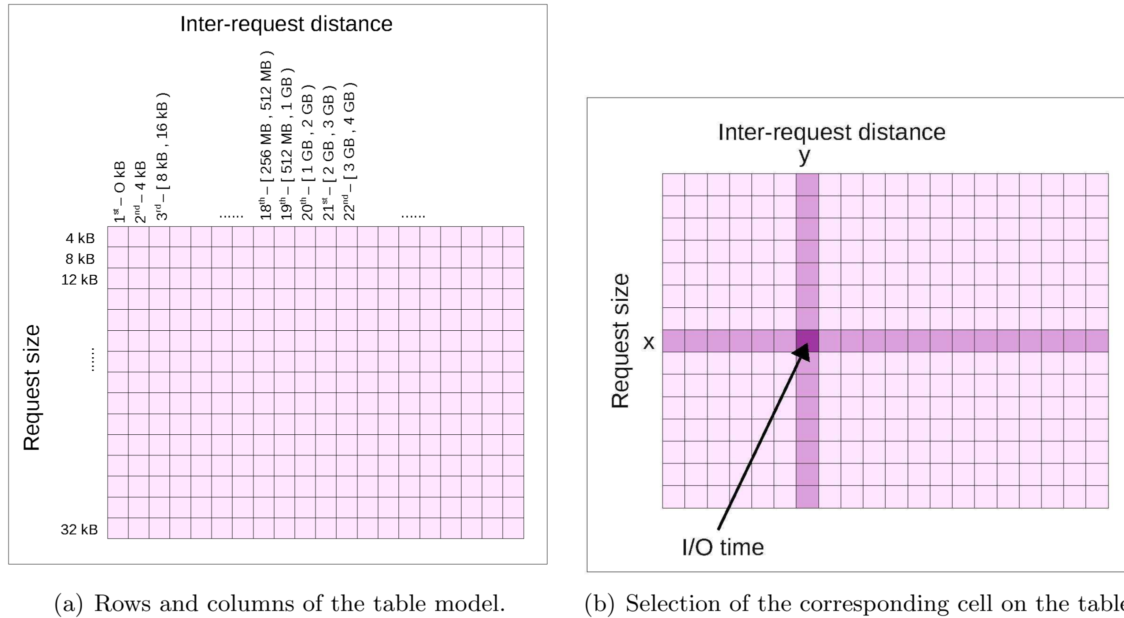
To sum up, given a request, its type selects the table to use either the read or write table, its size the row of the table, and the inter–request distance from the previous request the column. The value of the selected cell is the I/O time needed to serve a request with these parameters. Figure 3.2 presents the table–based disk model. The figure on the left shows the possible values for rows and columns, while the figure on the right summarizes the selection process of the corresponding cell, given a request of size x and inter–request distance y.

Dynamic behavior

In order to model the disk behavior in a precise way, *to adapt the model to the current workload*, and *to catch the effects of the disk cache*, a dynamic method has been implemented. The table–model always keeps updating cells as requests are served by the real disk drive.

During regular system operation, each time a new I/O request is dispatched by the real disk, the system calculates its I/O time, and with its type, size and inter–request distance from the previous request, it updates the corresponding cell of the table.

Each cell stores the *last sixty four* measured I/O times for the cell, and the I/O time returned is computed by averaging all these values. When, for a cell, a new sample is received, the oldest sample is forgotten, and the new one added. In this way, our model forgets past I/O times that depend on past workloads, and keeps values that depends on the current workloads. The number of sixty four has been chosen after performing an analysis of the sensitivity of the disk model to the number of averaged values per cell (see Section 3.5.1). Specifically, we have evaluated the accuracy of the model when the number of averaged values is eight, sixteen, thirty two or sixty four. The study shows that, when each cell stores the average of the last sixty four values, our model gives fairly precise estimations, and, thanks to this dynamic approach, the virtual disk's behavior significantly matches the behavior of the real drive.



(a) Rows and columns of the table model.

(b) Selection of the corresponding cell on the table.

Figure 3.2: Table disk model.

Let us emphasize that, since the proposed model is always learning the behavior of the disk by updating dynamically the tables as requests are served, and the request sequence depends on the current workload, the state of the disk cache is taking into account in an indirect way.

We are aware that the model does not explicitly consider several modern disk features, such as zoned recording, track/cylinder skew, and bad sector remapping. The impact of these features, however, is indirectly modelled through the I/O times obtained from the real disk during the dynamic update of the tables.

3.2.2. I/O schedulers for the virtual disk

As we mentioned before, the virtual disk has an I/O scheduler to manage its request queue and to decide the order in which requests are dispatched to the disk drive. In Linux 2.6.23 four disk I/O schedulers are provided [42, 51, 52]:

- Anticipatory [1] (AS) [42], that minimizes the number of seek operations in the presence of synchronous read requests: if a read request has just been serviced, it stalls the disk and waits (a small interval of time) for adjacent requests. Furthermore, AS sorts requests by LBA sector on disk. Note that the first version of Anticipatory, proposed by Iyer and Druschel [47], waits for additional requests to arrive from the process that issued the last serviced request, and not for adjacent requests of any process.
- Deadline [42], that manages two request queues, one for read operations and another for writes, sorted by physical location on disk. To avoid request starvation, it assigns an expiration time to each request, giving a higher preference to reads. Requests with overdue deadlines are served first.

[1] Note that the Anticipatory scheduler was removed from the Linux kernel as of version 2.6.33 because, supposedly, its behavior is mostly a subset of CFQ [96].

- Complete Fair Queuing (CFQ) [42], that assigns incoming I/O requests to specific queues based on the processes that issued them. Requests in each queue are sorted based on their physical locations. It gives to each process exclusive access to the disk for a quantum of time by serving a few requests of the same process in a row, and even sleeping a time interval to wait for the next request of the process. The process queues are selected by using a round robin policy. CFQ is provided as the default I/O scheduler in the Linux kernel.
- No–operation (Noop) [42], that imposes a FIFO (First In First Out) order in the request queue, without performing any sorting. It is not truly a FCFS (First Come First Served) scheduler as requests may be merged with other requests ahead in the queue, if they are physically consecutive.

CFQ and AS are process–aware schedulers that use information about processes that issue I/O operations to perform the scheduling. The former maintains a request queue for each process submitting I/O requests, and the latter keeps track of per–process statistics, pertaining to block I/O habits, to make its decisions. Both use the `current` kernel variable to get the `io_context` structure of the process that issues a request. This structure is then used for sorting requests in the scheduler queue or for maintaining statistics. The problem is that, in the virtual disk, requests are queued in the scheduler by the virtual disk itself. Therefore, all virtual requests belong to the kernel thread of the disk simulator, and have the same `io_context`. In order to fix this problem, the schedulers have been adapted to retrieve a process's `io_context` structure in a different way. The new schedulers, called *CFQ–VD* and *AS–VD*, behaves exactly as the corresponding original ones, but work with the virtual disk.

The solution works as follows. For each request copied to the disk simulator, the system records information about the process that issued the request: its PID (Process Identifier); its creation time; its PPID (Parent Process Identifier); and its parent creation time. The virtual disk uses this information to index requests. When the first virtual request of a process is received, the virtual disk creates an `io_context` structure for this process, and associated this structure to any subsequent virtual request of the same process. In this way, the new schedulers CFQ–VD and AS–VD get the `io_context` information from the virtual requests themselves, and not from the process that submitted them.

To properly identify the requests for a process, for each process, we need not only its PID, but also its PPID, its creation time, and its parent creation time. The problem is that the Linux operating system reuses process ID values [97], and it does not guarantee uniqueness of process IDs over a long period.

Since Deadline and Noop do not take into account any process information to schedule requests, the virtual disk can use them without any modification.

Finally, it is worth remarking that, because the virtual disk appears as a block device, it can have a different I/O scheduler to the one assigned to the real disk, and it is even possible to change the I/O scheduler of the virtual disk on the fly, without rebooting the system.

3.2.3. Request management

For each real request submitted to the real disk, the in–kernel disk simulator receives the corresponding virtual request that is copied to an auxiliary queue by the system. The virtual

disk extracts virtual requests from the auxiliary queue and inserts them into its I/O scheduler. If the I/O mechanism that the virtual disk is reproducing is less efficient than the mechanism active on the real disk, the virtual disk could serve requests slower than the real one. So, it is possible that, when the virtual disk serves a request of a process, there are several requests of the same process waiting to be queued. To perform a right scheduling, the virtual disk should know the order in which these requests were issued (the first one, the second one, etc.), and also whether a request was submitted when the previous one was ended or not. That is, requests of a process have to be scheduled in the virtual disk in the same order as the system had scheduled them. In order to fulfill these requirements, the virtual disk controls the requests' arrival order and the dependencies between them.

Since read and write operations have different requirements, our implementation also manages them in different ways. Read operations are usually synchronous with respect to their corresponding applications: when a read request is submitted, its application is blocked and it can not issue a new operation until the request is served. Read operations, hence, introduce dependencies among requests of the same process: a new request can not be issued until the previous one is not finished. For instance, if a process submits two synchronous read requests, RQ_1 and then RQ_2, the virtual disk has to queue them in the same order: first RQ_1, and when this one is finished, and only then, RQ_2.

This order control should also be done among requests of related processes, i.e., a child process and its parent, because there are dependencies between them too. A simple example is a parent process that executes the following piece of code:

1. Issue a synchronous read request (P_RQ_1).
2. Create a child process and wait for it.
3. Issue a second synchronous read request (P_RQ_2).

If the child process issues a synchronous read request (C_RQ) during its execution, the following dependencies will arise among the three requests:

a) C_RQ has to be issued once P_RQ_1 has finished, and
b) P_RQ_2 can not be issued until C_RQ has been completed, and the child process has finished.

Note that if the parent process does not wait for its child, dependencies among C_RQ and the requests of the parent do not appear, although the dependency between P_RQ_1 and P_RQ_2 does.

Nevertheless, there also exist asynchronous read requests. For example, the Linux operating system supports sequential file prefetching in a generic read–ahead framework. When it detects a sequential access to a file, it actively intercepts file read requests, and transforms small requests into large asynchronous read–ahead requests [51, 42, 98]. Subsequent reads submitted by an application to that file could be served directly from the page cache, and the application will not be blocked on I/O operations, since the requested data has been prefetched. An interesting point is that, although these read–ahead requests are "composed" by the operating system, they belong to the process that submitted the "original" one, because they are queued in the I/O scheduler by the process itself.

Thus, the virtual disk has to distinguish between two kinds of read requests: synchronous, submitted by the application (*synchronous reads*), and asynchronous, submitted by the operating system (*read–aheads*).

On the other hand, write operations are usually asynchronous with respect to the application, and do not have dependencies. The operating system defers the write of dirty blocks to disk until they grow older than a threshold, or when free memory shrinks below a level. In Linux, the `pdflush` thread is responsible for writing dirty pages to disk [42, 52].

The virtual disk maintains the arrival order of the requests and dependencies between them, by managing three queues in addition to the scheduler queue. It also implements a heuristic to decide the insertion of requests into the scheduler queue. The next sections describe the queues and heuristic of the disk simulator.

Auxiliary queues

The three queues, that controls the dependencies and the arrival order, are:

- The *shared queue*, that makes the communication between the disk simulator and the operating system. When an I/O request is submitted, just before queueing it in the scheduler of the real disk, the system copies its main parameters to the shared queue. These parameters are: the LBA sector number; the size; the type (read or write); the priority of the request; the PID of the process that has issued the request; the PID of its parent process; and the creation time of both processes. For read operations, it also records whether the operation is a read–ahead request or not.

  Note that, to not introduce overhead in the regular I/O path of a request, the system does not create the corresponding virtual request, it just copies the main parameters needed to create it. By using these parameters, the virtual disk itself will create the new virtual request.
- The *waiting queue*, that stores requests which can not be inserted into the scheduler queue because they have dependencies to meet. This queue also maintains the arrival order of the submitted requests. After serving a request and before dispatching the next one, the virtual disk creates a virtual request for any pending entry in the shared queue, and inserts the new requests into the waiting queue. Each virtual request is inserted into this queue, even if it does not have dependencies, and from there to the scheduler queue. To facilitate the control of dependencies, this queue is indexed by processes, and each process keeps its own pending requests.
- The *process queue*, that is used for exactly knowing the requests that are pending in the scheduler queue of the virtual disk. It is needed because the Linux operating system manages I/O scheduler queues as black boxes that can not be scanned. The process queue traces these pending requests and, hence, allows the virtual disk to control dependencies with read operations dispatched but not finished. For instance, the disk simulator can know whether a new virtual request has a dependency with one that is already in the scheduler, or even that has already been sent to disk but not ended yet. The process and the scheduler queues have the same requests, and when a request is finished, it is deleted from both. The process queue is indexed by processes too.

It is important to emphasize that the shared and waiting queues have quite different goals. The main goal of the former is to prevent the virtual disk from interfering with the regular processing of requests. Once the operating system records the parameters of a real request in the shared queue, the request is inserted into the scheduler queue of the real disk, as usual. Any subsequent processing is done by the virtual disk itself, without delaying the processing

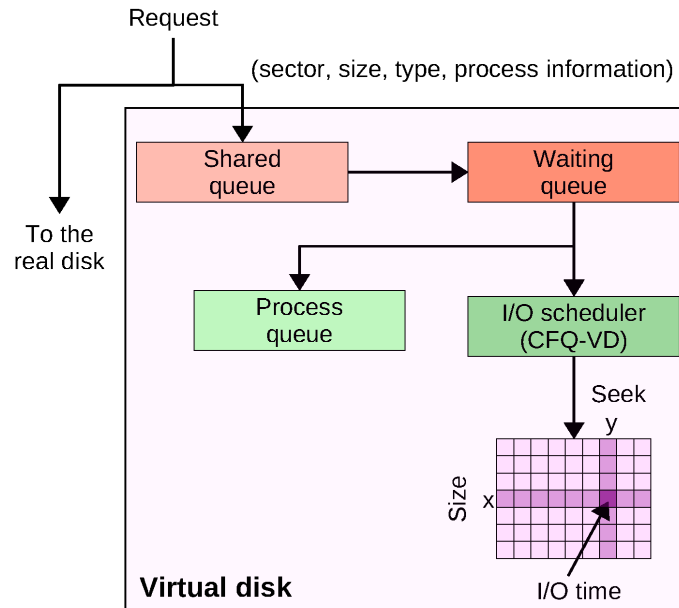


Figure 3.3: The auxiliary and I/O scheduler queues, and table model of the virtual disk.

of the regular request. This delay could downgrade the I/O performance of the system and affect the order in which the scheduler of the real disk serves requests. However, the main goal of the latter is to control request dependencies, and to store requests that cannot be queued in the scheduler yet.

Figure 3.3 presents an scheme of the auxiliary and scheduler queues, and the flow of requests throughout these queues.

Heuristic control

A request in the waiting queue is moved to the scheduler if, and only if, it has solved its dependencies. To control these dependencies, we have implemented the following heuristic:

- Write requests are inserted immediately into the scheduler queue. They are usually asynchronous, and do not have any dependency.
- A *synchronous read* request will be inserted into the scheduler queue if there is not another *synchronous read* request of the same process, either ahead in the waiting queue, or in the scheduler queue. Note that a *synchronous read* request can be inserted into the scheduler queue even when there already exists a *read–ahead* request of the same process in this queue.
- A *read–ahead* request is inserted into the scheduler queue if there is no *synchronous read* request of the same process ahead in the waiting queue.
- The first request of a new process is inserted into the scheduler queue when the last request of its parent process, issued before child creation, has been served.
- If a parent process waits for the completion of its child, none of the new requests of the parent will be inserted into the scheduler queue until the child exits.

- If the scheduler queue is congested, no requests are queued. In the Linux operating system, a queue is considered congested if the number of pending requests exceeds a certain threshold. In that case, it is marked as full and new requests are not admitted, blocking the process that issued them. The queue will become uncongested when the number of pending requests falls below this value. Section 3.2.6 explains the management of the congested queue performed by the virtual disk.

We are aware that the above heuristic does not control all the dependencies between requests. For instance, dependencies among requests of two independent processes that communicate with each other through a pipe. However, we think that most of the dependencies are captured by our heuristic, and the virtual disk can process requests in the order it had used if being the real one.

It is worth remarking that this heuristic tries to copy the default Linux behavior, and its goal is to imitate how requests arrive to the I/O scheduler queue of a regular disk.

3.2.4. Time control

We have implemented a time control method to analyze the performance achieved by the virtual disk. This control allows us to compare the performance of the virtual disk with the performance of the real one.

The time control method is based on the arrival and completion times of virtual requests. When a virtual request is queued in the I/O scheduler, after solving its dependencies, the virtual disk records its arrival time. When the virtual request finishes, its completion time is also recorded. Service time is calculated by subtracting the arrival time from the completion time, and it includes the waiting time in the scheduler queue, and the I/O time provided by the corresponding table. Therefore, although, in the virtual disk, a request has to be moved through different queues in order to be attended, its service time is computed in the same way as in a real disk, i.e., it is the elapsed time since the request is inserted into the scheduler queue until the completion of the request.

An interesting point is that the system also uses the table model to compute the I/O time of serving a real request, and not its I/O time given by the real disk. As our accuracy study shows (see Section 3.5.1), the virtual disk's behavior is usually very similar to that of the real disk, so it could use service times of requests submitted to the real disk. However, since behaviors are similar but not identical, it is more coherent to compare I/O times obtained from the same source.

For each served request, by using its size and its distance from the previous one, the system obtains, from the table model, the estimated I/O time to serve it. If $T_{ServiceDisk}$ denotes the time needed to serve a request from the real disk, T_{Wait} denotes the time that this request spends on the scheduler queue waiting to be dispatched, and $T_{Table-I/O}$ specifies the estimation given by the table model to serve a request of this features, our time control estimates the service time as

$$T_{ServiceDisk} = T_{Wait} + T_{Table-I/O}. \tag{3.1}$$

3.2.5. Training the table

The read and write tables can be initialized on–line or off–line. For the off–line initialization, we have implemented a training program. This program has to be executed only once for every disk model, and, due to write operations, before actually using the real disk. The training is usually “fast”. In our system, it took 80 minutes for a 400 GB hard disk drive, and 2 minutes for a 64 GB SSD drive. It is important to realize that, during its execution, there are no other programs performing I/O operations on the disk, and operations are submitted in a synchronous way by a single process, so the disk scheduler has no influence on the obtained results.

The program receives the following input parameters: the operation type (read or write), the number of samples per cell, and the disk capacity. The disk capacity sets the maximum inter–request distance. The off–line training program issues requests in a random way and uniformly distributed over all cells of a table. It picks up a cell randomly and then generates a read or write request (depending on the operation type) with the proper size given by the cell’s row, and a random inter–request distance which falls into the range given by the cell’s column. The program performs as many operations as needed to obtain the requested number of samples per cell.

Although the training program is executed in user space, tables are built inside the Linux kernel. The kernel has been instrumented to record, for each request, its size, its type, the inter–request distance from the previous request, and the disk I/O time to serve the request. Tables are calculated as requests issued by the training program are served, and the value of each cell is the mean of the samples obtained for the cell.

Once obtained, the tables are copied from the kernel through the */proc* virtual file system, and saved for later use. Note that this copy does not include the 64 samples obtained per cell but just their average.

/proc is also used for providing the computed tables to the virtual disk after an off–line initialization. In this case, every cell initially stores only one value; new I/O times are added to a cell as the real disk produces them, up to a maximum of 64 samples per cell. Once the maximum is reached, old samples are discarded to make room for new ones. Since a cell can stores less that 64 samples, the I/O time that the disk model returns for a cell is always the average of the available samples in the cell.

The training program produces a random disk access pattern which does not appear in many workloads. Thus, the tables obtained by the training program could fail to reflect the actual value of many cells because those values (and the hard disk’s performance as well) depend on the active workload. For instance, a workload with random requests (as produced by the training program) achieves a poor performance since produce large seeks, and does not leverage either the disk cache or the prefetching mechanism provided by the drive. A workload with large seeks, but with some disk locality and sequential accesses, gets a much better output because profits different mechanisms provided by the hard disk. The result is that requests with the same size, type and inter–request distances in different workloads could have very different service times. Therefore, the dynamic update of the cells, through I/O times provided by the real disk, is crucial because adapts the values to the current workload characteristics, and the disk behavior is modeled in a more precise way.

With the on–line configuration, the read and write tables will be first zeroed, and then our model will learn the behavior of the disk as requests are served. Of course, there is no

initial overhead at installation time to train the tables. But, for the initial predictions, the tables will not have enough information. In this case, for a not–yet–updated cell, the model will return the average of the corresponding column as I/O time, if this value is not zero; otherwise, it will return the average of the nearest column with non–zero cells.

3.2.6. Avoiding the scheduler's queue congestion

In the Linux operating system, each scheduler queue has a maximum number of allowed pending requests. When this threshold is reached, the queue is marked as full, and blockable processes trying to add new requests to the queue are put to sleep waiting for the completion of some I/O transfers [51].

Since, in our simulator, requests are inserted into the scheduler queue by the virtual disk driver itself, we can not allow the queue congestion. Otherwise, the virtual disk would be blocked, and no simulation will be performed. Thus, we have implemented a mechanism to control and avoid the congestion in the scheduler and waiting queues.

The congestion control in the I/O scheduler is done as in a normal system. The number of pending requests in the scheduler queue is controlled, and, when the congestion threshold is close to being reached, new requests are not queued, and the virtual disk is marked as congested.

The congestion control in the waiting queue avoids the scheduler congestion. If the number of pending requests in the waiting queue exceeds the congestion threshold, new requests are neither inserted into this queue nor into the scheduler queue, and the virtual disk is marked as congested.

Note that, since asynchronous read and write requests do not have any dependencies, the I/O scheduler queue can be congested, while the waiting queue is not.

When the virtual disk is congested, three measures are taken:

- The waiting queue is cleaned by removing all its requests.
- No requests are inserted into the scheduler queue. The insertion is blocked until the virtual disk is marked as uncongested.
- No comparison between the performance of the real and virtual disks is done.

When the number of requests in the scheduler queue is again below the congestion threshold, the virtual disk is marked as uncongested, and all the performance counters are set to zero to start over with the comparison.

It is important not to drain the scheduler queue during congestion. Otherwise, new requests would be sent to disk at once, without any delay. This would make their service time small, whereas the service time of the same requests in the real disk (probably, also congested or almost congested) would be higher due to the waiting time in the scheduler queue, making the real disk wrongly slow.

3.2.7. Operation of the disk simulator

Let us close this section by summarizing the operation of the virtual disk explained above, and detailing how the simulation process is run. Figure 3.4 presents a scheme of this simulation and of the interaction between the real and virtual disks.

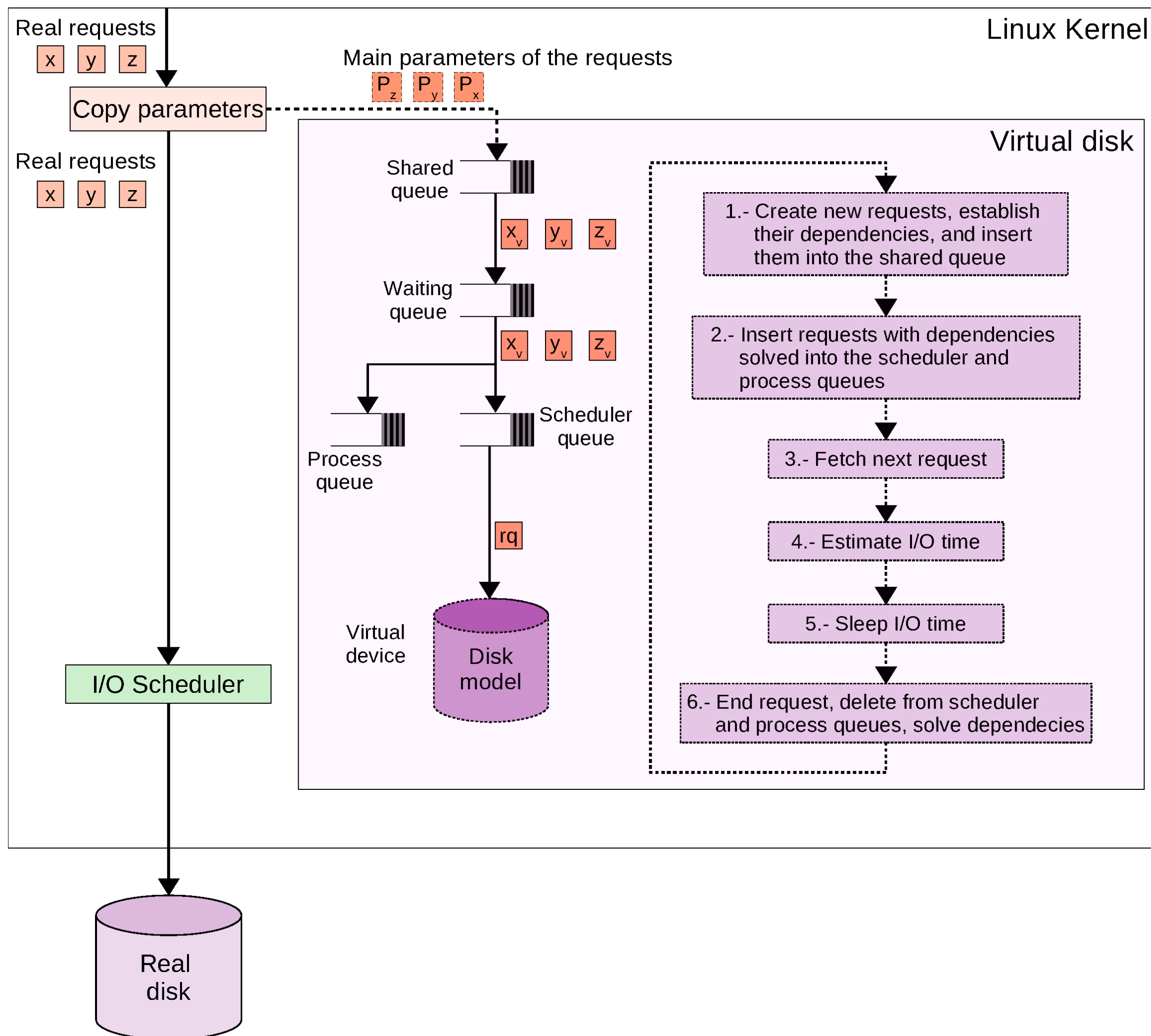


Figure 3.4: Virtual disk routine and the auxiliary queues.

Operating system

- For each request submitted to the real disk, the system copies its main parameters to the shared queue of the virtual disk. The parameters given are:
  - LBA sector number;
  - request size;
  - operation type;
  - priority of the request;
  - PID of the process that has issued the request;
  - PID of the parent process;
  - creation time of the process;
  - creation time of the parent process;

  - for read operations, whether the operation is a read–ahead request or not.

  By using these parameters, the virtual disk can create a virtual request with the same features as the real one.

  The system also checks if the virtual disk is plugged (because there were no pending virtual requests). If this is the case, it wakes up the virtual disk.

  Then, the real request is queued in the scheduler of the real disk without any modification.

- For each request served by the real disk, the system updates the corresponding table with the I/O time of the request. The service time of the request is computed and stored for a latter comparison.

Disk simulator

The kernel thread of the virtual disk continuously performs the following tasks:

1. For each new request copied to the shared queue:
   a) Create a virtual request by using the parameters:
      - LBA sector number;
      - request size;
      - operation type;
      - priority of the request.

   b) Create the `io_context` structure with the process information:
      - PID;
      - PPID;
      - creation time of the process;
      - creation time of the parent process;

      and associate the structure with the virtual request. Note that if the request is the first one issued by this process, the structure is created; otherwise, the virtual disk looks for it in the shared or process queues.

   c) Establish the dependencies of the new request by using its process and read–ahead information. Section 3.2.3 details the possible dependencies among requests.

   d) Insert the new virtual request into the waiting queue.
2. For each virtual request in the waiting queue with its dependencies solved:
   a) Insert it into the scheduler and process queues.
   b) Delete it from the waiting queue.
3. Fetch the next request from the scheduler queue. The scheduling policy assigned to the virtual disk will decide the next request to serve.
4. Get the estimated I/O time needed to attend the request from the table–based model of the real disk. The table is selected with the operation type, and the corresponding cell, that contains the I/O time, is selected with the size of the request and the inter–request distance from the previous request.
5. Sleep the estimated time to simulate that the disk operation is being performed. An important aspect of our disk simulator is that the time that the kernel thread sleeps to simulate an I/O operation is the time directly obtained from the tables minus an

adjustment time. This adjustment is mainly made up of the overhead introduced by the call to the sleep function. Without the adjustment, the virtual disk would always take more time to serve a request than the real disk.

6. Finally, after waking up:
   a) Complete the request.
   b) Compute and store the service time of the request for a latter comparison.
   c) Delete it from the scheduler and process queues.
   d) Solve possible dependencies that other virtual requests can have with the current one.

The virtual disk continuously runs this routine until there are no more pending requests to serve. Then, as a regular disk, it is plugged until new operations arrive. When a new request is submitted, the system copies its parameters to the shared queue and unplugs the virtual disk, and the process starts over.

3.3. A use case: REDCAP

Our first proposal, explained in detail in Chapter 2, has been REDCAP, a new cache of disk blocks, between the page and disk caches, that can significantly reduce I/O time of read requests [33, 34]. The REDCAP cache extends, in main memory, the cache of a disk drive. When a read cache miss occurs, it prefetchs some consecutive disk blocks, in such a way that it takes advantage of the read–ahead mechanism performed by the disk drive itself.

A dynamic activation–deactivation algorithm controls the performance achieved by REDCAP. The algorithm compares the time that the REDCAP cache needs to process requests with the estimated time to process them without that cache, and turns the cache on or off accordingly. There are two main states. In the *active* state, the REDCAP cache dispatches read requests. If the algorithm detects that access time is getting worse with than without the cache, it moves REDCAP to the *inactive* state. In the inactive state, REDCAP does not process requests, and no prefetching is performed. The algorithm, however, keeps studying the possible success of REDCAP: it simulates that its cache is still working, and records the hits and misses on each read request. When the algorithm detects that REDCAP could be efficient, moves it to the active state again.

Our first implementation estimates I/O times with the I/O times of the read requests sent to disk. On the active state, it uses the total times taken by the original read requests on a cache miss to estimate the total times of the read requests that are cache hit. On the inactive state, the cache time is estimated with the values stored during the active state. The algorithm computes a "seconds per kilobyte" average with the total I/O times and total size of the last 100 disk requests, and, with this average, it determines the proper state of REDCAP. Section 2.3.3 details how estimations are done, and how the state change is decided.

The use of an I/O time average actually provides a coarse model of the disk drive that works reasonably well, although it has problems in some workloads [33, 34]. For example, for a strided access pattern, with small strides, the algorithm is unable to set the proper state of REDCAP, which is turned on and off many times (see Section 2.5). Another problem is that this simple model hinders the implementation of the algorithm because it does not model other details, such as the impact of the I/O scheduler, the arrival order of the requests, and

so on. These problems has been managed by using several heuristics which has not always provided satisfactory results for all workloads.

In order to solve these problems, we have modified REDCAP to use the in–kernel virtual disk [35]. The new simulator implements a more accurate disk model (see Section 3.2.1), and provides better estimated I/O times that can improve and simplify the activation–deactivation algorithm. Depending on the state, the virtual disk simulates the behavior of the real disk in a normal system (active state) or in a REDCAP system (inactive state).

Now, the modified algorithm calculates the mean time required to serve one block of 4 kB by a REDCAP system and by a normal system, and compares these times. If the time needed by REDCAP is less than the time needed by the normal system, our cache is been effective, and it has to be active. Otherwise, it should be turned off because the system does not take advantage of the prefetching performed.

To perform the computation, the algorithm stores the following information:

- B_{Redcap}, that denotes the number of file system blocks (4 kB) served by REDCAP. In case of a cache hit, it only includes the number of requested blocks, but in a miss, it includes the requested and prefetched blocks.
- T_{Redcap}, that is the time that REDCAP needs to serve B_{Redcap} blocks. For each request, its service time is computed as the amount of time passed since the request arrives to REDCAP until it is ended. This time could be the time of a cache hit (to copy the requested blocks), or the time of a cache miss (to read from disk the requested and prefetched blocks), or even both times. When the blocks are read from disk, this time includes the time waiting in the scheduler queue (the waiting time) and the I/O time.
- B_{Normal_System}, that denotes the number of blocks requested by applications and the operating system's prefetching, also of 4 kB. This value is different from B_{Redcap} which also includes the blocks prefetched by REDCAP.
- T_{Normal_System}, that specifies the time that a normal system needs to read B_{Normal_System} blocks from disk. This time includes the waiting time of the request in the scheduler queue and the I/O time.

By using expressions, the new algorithm says that if condition

$$\frac{T_{Redcap}}{B_{Redcap}} < \frac{T_{Normal_System}}{B_{Normal_System}} \tag{3.2}$$

is true, REDCAP is improving the access time and it has to be active, otherwise, it has to be inactive.

In addition, the check interval has been also modified by fixing its value to 1000 requests, and never changes. In our previous implementation, the check is usually made after 100 requests [33, 34]. With the new disk model, each served request can update a different cell of the tables. Therefore, it is possible that, after 100 requests, the I/O times stored do not reflect the current workload, specially if a workload change has occurred. However, if the check is made after 1000 requests, the cells' averages are more representative, and estimations performed by our disk model can be more precise. Moreover, with 1000 requests, it is more likely that the tables have been adapted to changes in the workload.

Since the verification is made every 1000 requests, the condition actually checked by the algorithm is

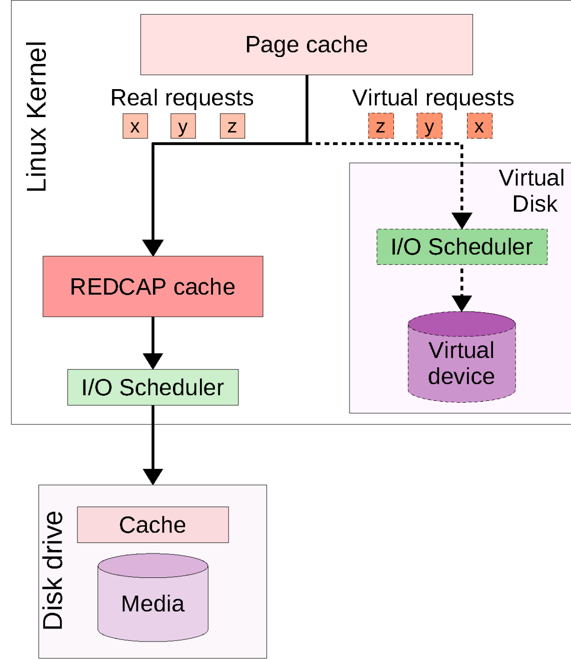


Figure 3.5: The virtual disk simulating a normal system when REDCAP is active.

$$\sum_{i=1}^{1000} \frac{T_{Redcap_i}}{B_{Redcap_i}} < \sum_{i=1}^{1000} \frac{T_{Normal_System_i}}{B_{Normal_System_i}}. \tag{3.3}$$

The next sections first describe the operation of the algorithm in the active state, then in the inactive case, and finally how the cache misses are now managed when REDCAP is active.

3.3.1. Active State

When REDCAP is in the active or pending–active states, requests are served as we have explained in Section 2.3.2, and the virtual disk simulates the behavior of the real disk in a normal system. Each original I/O request, without any modification, is inserted into the disk simulator before being managed by REDCAP. The virtual disk serves these original requests by simulating the real disk (see Section 3.2). Figure 3.5 shows an scheme of this adaptation. For the sake of clarity, we have omitted the auxiliary queues of the virtual disk.

The time that the virtual disk needs to serve a request corresponds to the time to serve the request in a normal system. For each request, the virtual disk provides B_{Normal_System} as the number of disk blocks served, and T_{Normal_System} as the time needed to attend B_{Normal_System} blocks. The time control method implemented in the virtual disk is used for calculating this time. It is computed as the time elapsed since the request arrives to the scheduler queue of the virtual disk until it is ended (see Section 3.2.4 for more details).

On the other hand, for each request, REDCAP calculates and stores the B_{Redcap} and T_{Redcap} that we have described above. Remember that T_{Redcap} can be the time for a cache

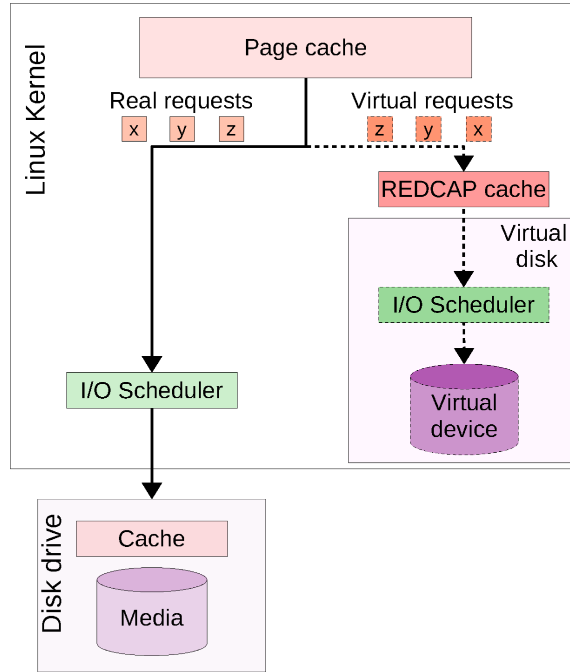


Figure 3.6: The virtual disk simulating a REDCAP system when it is inactive.

hit, a cache miss, or both. In the case of a cache miss, the service time of the REDCAP request is calculated by using the table model and not the I/O time given by the real disk (see Section 3.2.4 and Expression 3.1 for more details).

3.3.2. Inactive State

When REDCAP is in the inactive or pending–inactive states, the submitted requests are queued in the I/O scheduler of the real disk without modification, and the virtual disk simulates the behavior of the real disk if REDCAP would be active. Indeed, the system simulates that REDCAP is still working: it records cache hits and misses. For each cache miss, the corresponding REDCAP request is issued to the virtual disk which serves these requests by simulating the real disk. Figure 3.6 depicts this implementation. Again, to simplify the figure, we have omitted the auxiliary queues of the virtual disk.

Now, since the cache is being simulated, B_{Redcap} is exactly known, but the time needed by its management, T_{Redcap}, is calculated as:

$$T_{Redcap} = T_{Virtual_Disk} + T_{CHit}, \tag{3.4}$$

where $T_{Virtual_Disk}$ is the service time of the virtual disk, and corresponds to the time that REDCAP needs to read from disk the requested and prefetched blocks. While, T_{CHit} is the time of the cache hits, and is estimated by using values stored for cache hits during the active state (see Equation (2.10) in Section 2.3.3).

At the same time, the system calculates the block served, B_{Normal_System}, and the time needed to serve them, T_{Normal_System}. This total service time is again calculated by using the

estimated I/O times provided by the table model, and not the I/O times given by the real disk (see Section 3.2.4 and Expression 3.1).

3.3.3. Management of the cache misses

In addition to the activation–deactivation algorithm, an important aspect has been modified in this new version: the management of the cache misses. Now, when there is a cache miss, a single read operation is submitted to disk. All the blocks of the corresponding disk segment are read from disk in a single request (called REDCAP request), which is composed of the requested and prefetched blocks. But, again, to partially achieve *exclusive caching* [54], the data requested by the original read operation is not stored in the REDCAP cache. The corresponding REDCAP request has blocks from the cache and from the original request. When this request finishes, our mechanism manages the completion of the original operation. Note that the service time of a REDCAP request is the time that REDCAP needs to serve a cache miss. The decision of sending a single request has been taken because, after several tests, we have realized that, in the Linux kernel 2.6.23 and with the current experimental environment, it is more efficient to send an only one request than several independent requests to read from disk the requested and prefetched blocks.

3.4. Experiments and methodology

Several experiments have been performed to analyze the accuracy of the disk model and also the performance of REDCAP when it uses the in–kernel virtual disk. This section describes the hardware platform, benchmarks and Linux I/O schedulers used for carrying out the analyses.

3.4.1. Hardware platform

Our experiments are conducted on a 2.67 GHz Intel dual–core Xeon system with 1 GB of RAM and three disks, that is called **Hera**. The first one is the system disk. It is a Seagate ST3500630AS disk [6] that contains a Fedora Core 8 operating system. The system disk collects traces for a subsequent study.

The other two are the test drives and their main features are presented in Table 3.1. One is a Seagate ST3400620AS disk [6], that has a capacity of 400 GB and a built–in cache of 16 MB. It has a clean *Ext3* file system, containing nothing but files used for the tests. It was formatted and then files were created. During the explanation of the results, we refer to this test disk as “clean file system” (or “Clean FS”).

The second test drive is a Samsung HD322HJ disk [48]. It has a capacity of 320 GB and a 16 MB built–in cache. It contains several aged Ext3 file systems in different partitions, obtained by copying sector by sector the disk of our department server. The file system containing the users’ home directories has been selected to perform the tests; it is 270 GB in size, was in use for several years, and, at the time of the copy, was 84% full. Files for carrying out the benchmarks have also been created in this file system. We refer to it as “aged file system” (or “Aged FS”).

Table 3.1: Specifications of the test hard disks.

Features	Values	Values
Disk	Seagate ST3400620AS	Samsung HD322HJ
Capacity	400 GB	320 GB
Cache	16 MB	16 MB
Read	Adaptive	Adaptive
Write	Yes	Yes
Average latency	4.16 ms	4.17 ms
Rotational Speed	7200 RPM	7200 RPM
Seek time		
Read	< 8.5 ms (average)	8.9 ms (average)
Track–to–track, read	0.8 ms (typical)	0.8 ms (typical)
Nick name	“Clean file system” or “Clean FS”	“Aged file system” or “Aged FS”

3.4.2. Benchmarks

In order to study the behavior of the in–kernel disk simulator, we have used several benchmarks. Since we want to only evaluate the virtual disk and how it can be used for running simulations inside the kernel, we have selected I/O–bound tests.

Some of the benchmarks are the same as those used in the previous evaluation of REDCAP (see Section 2.4.3), and cover several access patterns: traversal of a directory tree with small files; sequential read; backward read; 8 kB strided read; and 512 kB strided read. We have also included two new tests that are quite interesting: a workload that covers all these access patterns by running their corresponding benchmarks in a row, and another one that creates a blend of access patterns by running all the tests at the same time. For the aged file system, a traversal of a directory tree with files of different sizes is also tested, being user’s home directories the selected directories.

Section 2.4.3 gives the description of the tests *Linux Kernel Read*, *IOR Read*, *TAC*, and *512 kB Strided Read*. Here, we only describes the new ones:

- 8 *kB Strided Read (8k–SR)*. This test reads a file with a strided access pattern with small strides. The benchmark reads a first block of 4 kB at offset 0, skips two blocks (8 kB), reads the next 4 kB block, skips another two blocks, and so on. Again, it is executed for 1, 2, 4, . . . , and 32 processes, and each process reads its own file. Files are the same reads by *IOR Read* and *TAC*. It has been written in C, and uses the POSIX `read` and `lseek` functions.
  Note that this benchmark is different from the one used for analyzing the behavior of REDCAP in Chapter 2 (see Section 2.4.3). In our first study, we used the 4 *kB Strided Read* test, in which the stride was of one block (4 kB). However, Linux kernel 2.6.23 is able to detect this access pattern, and it performs prefetching. Since we want an access

pattern with small strides that the Linux kernel does not detect, we have changed the stride from 4 kB to 8 kB.

- *Directory Read (DR)*. This benchmark is new, it reads files from selected directories in the aged file system by using:

```
find -type f -exec cat {} > /dev/null \;
```

We have chosen up to 32 user's home directories, whose sizes, including its files and subdirectories, range from 1 to 3 GB. This test performs a traversal of a directory tree with files of different sizes. It is only carried out in the aged file system, because the clean file system does not contain these directories. This benchmark is also executed for 1, 2, 4, ..., and 32 processes, each one running on a user's home directory.

- *All the benchmarks in a row*. In this test, the previous benchmarks are run one after another, without restarting the computer until the last is done. Since some of the benchmarks use the same files, the execution order tries to reduce the effect of the buffer cache. On the clean file system, the order is: *TAC*; *512k–SR*; *8k–SR*; *LKR*; and *IOR*. On the aged one, it is: *TAC*; *DR*; *512k–SR*; *8k–SR*; *LKR*; and *IOR*. Note that this test covers all the access pattern of the previous benchmarks. Thus, it shows how the virtual disk and the table–based model adapt to changes in the workload.
- *All the benchmarks at the same time*. This test runs all the previous benchmarks in parallel, and finishes when each one has been run at least once. If a benchmark ends when others are still in their first run, the benchmark is launched again. Unlike the previous tests, this one is only executed for 1, 2 and 4 processes, i.e., each benchmark is run for that number of processes, and each one reading its own files. The aim is to analyze the behavior of our proposal when the workload is a blend of different access patterns. Let us remark that, in this test, only "individual" benchmarks are run, and *All the benchmarks in a row* is not run.

3.4.3. I/O schedulers of the experiments

Since CFQ and AS have been the most widely used I/O schedulers in Linux [42, 51], we have selected them to evaluate the behavior of the virtual disk. It is interesting to note that CFQ has become the default I/O scheduler in the latest "official" versions of the Linux operating system, at the expense of AS. Therefore, all the benchmarks has been run for CFQ and also for AS.

To carry out the experiments, the real and virtual disks use the same scheduling policy. That is, when the real disk has the CFQ scheduler, the virtual disk has CFQ–VD, and when the real disk has AS, the virtual disk has AS–VD.

3.5. Results

This section evaluates the in–kernel disk simulator which has been implemented in a Linux kernel 2.6.23. REDCAP has been updated to this kernel, and modified to use the virtual disk. To short, this kernel is named *REDCAP–VD kernel*.

Two different sets of experiments have been carried out. The first one analyzes the accuracy of the virtual disk by comparing the I/O time it produces with the I/O time obtained by the real disk. The second set of experiments studies the utility of the virtual disk by analyzing

the performance achieved by the new implementation of REDCAP. In this second study, the results of the REDCAP–VD kernel have been compared to those obtained by a vanilla Linux kernel 2.6.23 (called to short *original kernel*). Neither the virtual disk nor REDCAP have been implemented in the original kernel.

Activity of the test disks has been traced by instrumenting both kernels to record request information: when a request starts and finishes, and when it arrives to the scheduler queue. The REDCAP–VD kernel also records information about the behavior of its cache, such as hits and misses, state changes, and virtual requests.

3.5.1. Accuracy of the virtual disk model

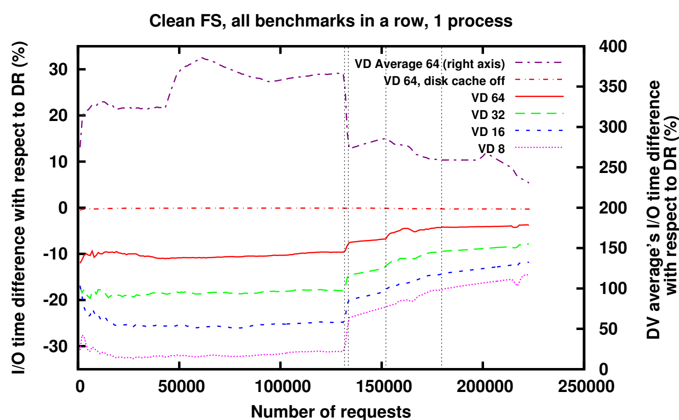
In order to evaluate the accuracy of the disk model, the *All the benchmarks in a row* test has been run by making the real and virtual disks serve the same requests and having both the same scheduling policy. The results for I/O times provide the comparison between the two disks to perform the study. Let us emphasize that I/O times compared are the times directly given by the real and the virtual disks. In this analysis, we do not use the table model to estimate I/O times for the real disk. REDCAP is neither active nor simulated. The virtual disk simulates the behavior of the real one: each submitted request is copied to the shared queue of the virtual disk just before being queued in the scheduler of the real disk. However, although both disks receive the same requests in the same order, they could dispatch them in a different order because each one has its own I/O scheduler with its own request queue. That is, the real and virtual disks have the same scheduling policy (CFQ and CFQ–VD, respectively), but they can sort requests differently, specially when there are more than one process issuing requests.

The test *All the benchmarks in a row* has been selected because it shows how the virtual disk adapts to changes in the workload, and, indirectly, its accuracy in all the benchmarks. This test also shows how the dynamic update of the tables allows the virtual disk to follow the behavior of the real drive.

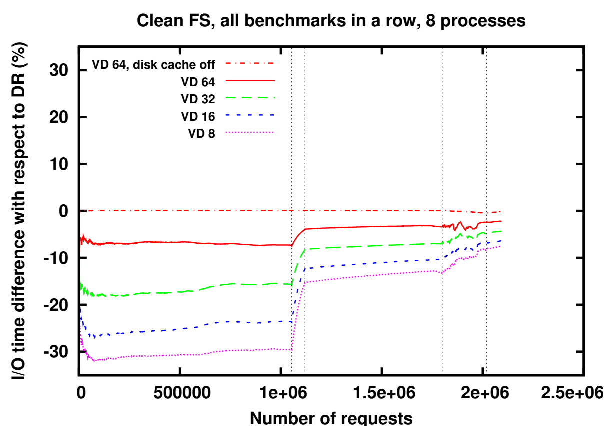
Figure 3.7 shows the evaluation of different configurations of the virtual disk based on the number of values averaged per cell in the tables. Specifically, we have analyzed the results obtained by averaging eight ("VD 8" in Figure 3.7), sixteen ("VD 16"), thirty two ("VD 32"), and sixty four ("VD 64") values per cell. It presents the difference, in percentage of I/O time, of the virtual disk with respect to the real disk for 1, 8 and 32 processes, the clean file system, and the CFQ scheduler (the virtual disk has CFQ–VD). Each point of the figure represents the difference between the last 1000 requests served by each disk. In the figure, to facilitate the comparison and explanation, vertical black dashed lines mark the end of a benchmark and the beginning of the next one. Note that a negative difference means that the virtual disk is "faster" than the real disk.

The average I/O time per request has also been measured during the execution of the test and presented in Table 3.2 for 1, 8, and 32 processes. The row "Real disk" shows the times achieved by the real disk, and the rows "VD 8", "VD 16", "VD 32", and "VD 64" show the times of the configurations analyzed. Times are presented independently for each individual benchmark.

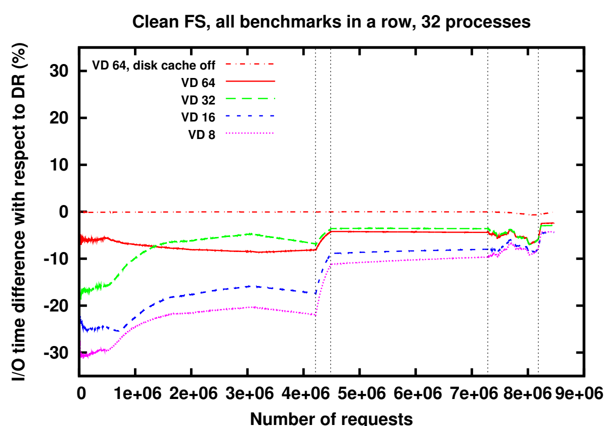
As expected, when each cell stores the average of the last sixty four values, our model presents its best behavior by matching the real disk in a closer way. By having more values per cell, the estimation performed is more precise.



(a) 1 process.



(b) 8 processes.



(c) 32 processes.

Figure 3.7: Difference, in percentage of I/O time, of the virtual disk with respect to the real disk in the *All the benchmarks in a row* test, when the clean file system and CFQ are used, for 1 (a), 8 (b) and 32 (c) processes. Vertical dashed lines mark the end of a benchmark and the beginning of the next one. Benchmarks conforming the *All the benchmarks in a row* test are executed in the order TAC, 512k–SR, 8k–SR, LKR, and IOR.

The major differences are observed at the beginning of the test execution, when the TAC benchmark is run. These differences are due to the disk cache. Like a sequential access pattern takes advantage of the prefetching performed by the disk drive, a backward access pattern takes advantage of the *immediate read* done by the disk drive. However, at a cache miss, the time needed to read the requested blocks is larger than in the forward access due to the backward seeks. Although our read table is updated with all these times, the disk cache effect has a noticeable impact on the stored times, making the virtual disk faster than the real one. As we can observe in Table 3.2, when TAC is run, the I/O time per request for all the configurations of the virtual disk is smaller than the one of the real disk. The "VD 64" configuration presents the smallest difference, although for 32 processes, the "VD 64" and "VD 32" configurations have a similar behavior.

After the TAC execution, the difference between both disks decreases quickly. In fact, when each cell stores the average of the last sixty four values, this difference is, on average, less than 5%. The average I/O time per request during the execution of the 512k–SR, 8k–SR, LKR and IOR benchmarks, can also be observed in Table 3.2. Now, differences between the real disk and the virtual disk configurations is very small. Only for LKR and 32 processes, the difference is a bit larger due to the disk cache again.

The reason of this fast adaptation is that, although our tables have many cells, just a small set of cells is used and updated, even when the test is executed for 32 processes. This can be observed in Figure 3.8, which shows the cells in the read table that have been modified once the execution of the test has finished. The x–axis stands for the inter–request distance (table columns), and the request sizes (table rows) are represented in the y–axis.

By observing all these results, we can claim that the "VD 64" configuration of the virtual disk has a good behavior, and that *the virtual disk closely matches the real one.*

One reason why the virtual disk behavior is not the same as the real disk's one is the difficulty to simulate the disk cache, as we haven seen in the TAC case. Performance of disk caches is determined by a large set of possible policies and parameters [49]. Most of these design parameters are set by manufacturers, and sometimes their configuration and behavior change a lot from one model to another. Hence, it is not easy to deduce the real operation of disk caches [1, 49], specially for those that dynamically change their own configuration depending on the current access pattern. To analyze the cache influence, the same test has been run with the disk cache off, and only for the "VD 64" configuration. The results appear as line "VD 64, disk cache off" in Figure 3.7. Without cache, the virtual disk matches the real one in a very accurate way, with differences less than 0.2%. Now, when TAC benchmark is executed, the difference between both disks is negligible. Table 3.3 also presents the average I/O time per request calculated during the execution of the test when the disk cache is off, and for 1, 8 and 32 processes. The average I/O times per request predicted by our table model are almost identical to those provided by the real disk, except for the IOR benchmark. The reason of the difference in I/O time of the IOR test is due to the fact real disk receives requests larger than 128 kB, and the virtual disk does not have information for those requests (in these cases, the virtual disk interprets the requests as ones of 128 kB). As we have mentioned in Section 3.2.1, these large requests are the result of merging several smaller requests.

Finally, in order to show that the request size is important in the proposed disk model, we have run the same test, but this time taking into account only the type and inter–request distance of every request, and leaving out request sizes. The disk has been modeled by using,

Table 3.2: For the real disk and the four configurations of the virtual disk, average I/O time per request measured during the execution of the *All the benchmark in a row* test, for 1 (a), 8 (b) and 32 (c) processes.

(a) 1 process.

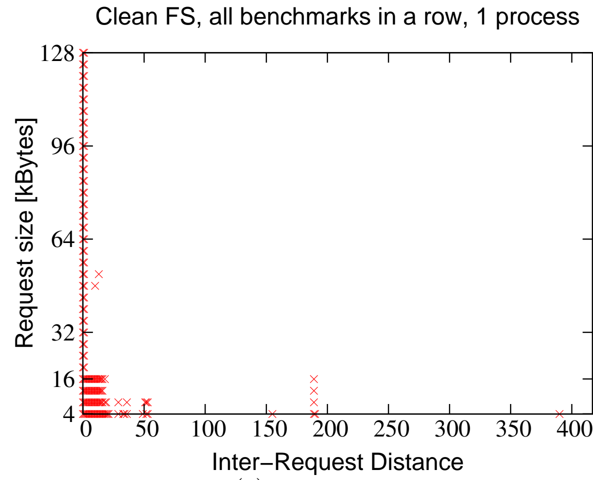
	Average I/O time per request				
	TAC	**512k–SR**	**8k–SR**	**LKR**	**IOR**
Real disk	351 μs	6695 μs	342 μs	562 μs	420 μs
VD 8	241 μs	6655 μs	347 μs	535 μs	420 μs
VD 16	263 μs	6693 μs	341 μs	551 μs	419 μs
VD 32	286 μs	6671 μs	346 μs	573 μs	420 μs
VD 64	316 μs	6670 μs	346 μs	580 μs	413 μs

(b) 8 processes.

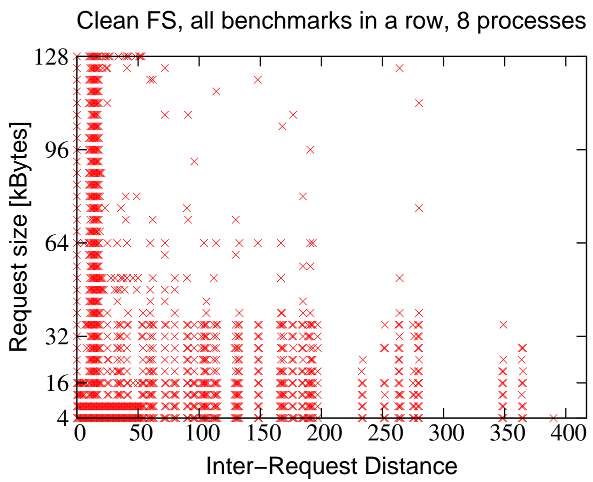
	Average I/O time per request				
	TAC	**512k–SR**	**8k–SR**	**LKR**	**IOR**
Real disk	435 μs	6604 μs	363 μs	3383 μs	1983 μs
VD 8	301 μs	6587 μs	342 μs	3263 μs	1980 μs
VD 16	327 μs	6573 μs	354 μs	3221 μs	1969 μs
VD 32	361 μs	6577 μs	355 μs	3238 μs	1986 μs
VD 64	397 μs	6581 μs	358 μs	3247 μs	2002 μs

(c) 32 processes.

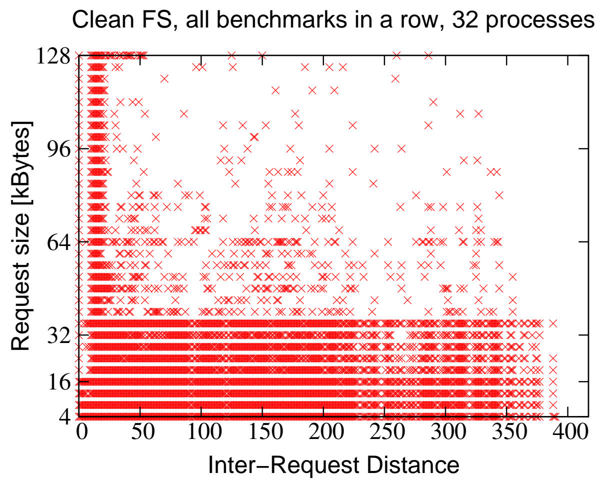
	Average I/O time per request				
	TAC	**512k–SR**	**8k–SR**	**LKR**	**IOR**
Real disk	470 μs	6661 μs	389 μs	10635 μs	1991 μs
VD 8	335 μs	6633 μs	372 μs	9902 μs	1941 μs
VD 16	354 μs	6634 μs	370 μs	9793 μs	1961 μs
VD 32	400 μs	6635 μs	376 μs	9893 μs	1974 μs
VD 64	394 μs	6638 μs	371 μs	9916 μs	1988 μs



(a) 1 process.



(b) 8 processes.



(c) 32 processes.

Figure 3.8: Modified cells in the read table once all the benchmarks in a row are executed for the clean file system, CFQ, and 1 (a), 8 (b) and 32 (c) processes.

Table 3.3: Average I/O time per request measured during the execution of the *All the benchmark in a row test* when the disk cache is off, for 1 (a), 8 (b) and 32 (c) processes.

(a) 1 process.

	Average I/O time per request				
	TAC	**512k–SR**	**8k–SR**	**LKR**	**IOR**
Real disk, disk cache off	8153 μs	6805 μs	8383 μs	6597 μs	8524 μs
VD 64, disk cache off	8141 μs	6779 μs	8358 μs	6517 μs	8479 μs

(b) 8 processes.

	Average I/O time per request				
	TAC	**512k–SR**	**8k–SR**	**LKR**	**IOR**
Real disk, disk cache off	8131 μs	6605 μs	8339 μs	12374 μs	9957 μs
VD 64, disk cache off	8132 μs	6593 μs	8321 μs	11975 μs	10514 μs

(c) 32 processes.

	Average I/O time per request				
	TAC	**512k–SR**	**8k–SR**	**LKR**	**IOR**
Real disk, disk cache off	8177 μs	6678 μs	8359 μs	13097 μs	9941 μs
VD 64, disk cache off	8167 μs	6663 μs	8345 μs	12550 μs	11242 μs

for a given inter–request distance, the average of the values in its corresponding column, i.e., the average I/O time obtained for all the sizes. This study has been performed only for 1 process because the virtual disk becomes very slow and the execution takes a long time. The result appears as "VD 64 average" in Figure 3.7(a). This single execution, however, is enough to see that, when request sizes are not taken into account, differences between our disk model and the real disk are very significant, and the behavior of the virtual disk does not match the behavior of the real one.

Therefore, for the remainder of our experiments, our disk model will store the average of the last sixty four values per cell.

3.5.2. Performance of REDCAP with the virtual disk

To get a better insight into the advantages and features of the virtual disk, we have analyzed its potential utility in REDCAP. This study shows how REDCAP uses the virtual disk to improve its activation–deactivation algorithm by making a more precise simulation. To carry out the evaluation, the REDCAP cache size has been fixed to 64 MB, four times as large as the cache of the test disks. The main memory utilization is less than 6.25%. The cache has been configured with 512 segments of 128 kB each. This segment size has been selected

because it showed the best behavior in our early tests [33] (see Section 2.5.1).

We have used the benchmarks described in Section 3.4.2 to evaluate our proposal. Since these benchmarks cover several access patterns, the results show how REDCAP and the virtual disk behave with different workloads, and even how the disk model and REDCAP adapt to changes in workloads. We have carried out five runs for every benchmark and kernel, and the results showed are the average of this five runs. The confidence intervals for the means, for a 95% confidence level, are also included as error bars. The computer is restarted after every run, hence all tests have been performed with a cold page cache and a cold REDCAP cache. In all the executions, the initial state of REDCAP is *active*.

The read and write tables, obtained from the off–line training, are given to the virtual disk each time the system is booted, so the tables are initially the same in all the tests. Each cell of the table averages the last sixty four values computed. We have decided to use the off–line training to have good predictions from the moment the experiments start; otherwise (with an on–line training) the tables would not have enough information for the first predictions, and an initial time interval would be required to achieve good ones.

Benchmarks executed independently

We first analyze the results for the benchmarks run in an independent way. Figures 3.9 and 3.10 show improvements in application time achieved by REDCAP with respect to the original kernel for the clean file system and the CFQ and AS schedulers, respectively. Analogously, the results for the aged file system are presented in Figures 3.11 and 3.12. For simplicity reasons and to facilitate the comparison, all the tests are ordered in the figures in the same order as they are run in *All the benchmarks in a row* test. Moreover, to explain the results of the AS scheduler, improvements in I/O time achieved for the clean and aged file systems are depicted in Figures 3.13 and 3.14, respectively.

TAC. With this test, REDCAP always performs better than the original kernel. For the clean file system, it obtains improvements of up to 24.7% and 29.1% for CFQ and AS, respectively, and, for the aged file system, of up to 28.4% and 32.5% for CFQ and AS, respectively. The regular operating system is unable to detect the backward access pattern, and it does not perform any prefetching. However, REDCAP takes advantage of the *immediate read* performed by the disk drives, and of the prefetching performed by its own cache, and the algorithm keeps its cache active almost all the time. Now, the `tac` command reads the files of the test by issuing requests of 8 kB. Therefore, almost fifteen out of every sixteen application requests are cache hits. With this cache hit rate, large improvements should be expected. The problem is that the time needed to serve a cache miss in a backward access is larger than in a forward access, due to the backward seeks. For instance, for the aged file system, the read of 256 sectors, that is the size of the REDCAP requests, in a sequential backward access takes around 3500 μs, and 1200 μs in a sequential forward access. Obviously, when requests are served from a disk cache, access direction is not relevant. For this reason, REDCAP does not achieve larger reductions, although it almost gets its maximum possible improvements for this workload.

Directory Read. In this benchmark, only executed in the aged file system, our method reduces the application time up to 9.7% and 13.6% for CFQ and AS, respectively, although

there are two remarkable exceptions.

The first one is with the CFQ scheduler and for 2 and 4 processes, for which, if we take into account the confidence intervals, both kernels statistically have the same performance. Improvements depend on the scheduler used, and, except for 1 process, reductions obtained for the AS scheduler are larger than those for CFQ. However, REDCAP behaves almost the same with both schedulers. The mean percentage of activation, i.e. the number of requests served when REDCAP is on, is rather similar, 81% and 78% for CFQ and AS, respectively. The improvement differences are due to the access pattern of this test and the behavior of the schedulers.

In this benchmark, each process reads files of one users' home directory tree. Since home directories belong to different users, they have files of rather different characteristics. For instance, while one of the directories has 7 files that almost use all the directory space (2.3 GB out of 2.35 GB), another directory contains a large amount of small files, with an average size of 72 kB. Therefore, home directories have a quite different distribution on disk, with many files (specially large files) that could be fragmented due to the use of the file system over several years. The selection of home directories (and subdirectories within them) read by each process is made according to their overall size (including subdirectories and files), without taking into account their number of files, or their files' sizes. The idea is to make every process roughly read the same amount of data.

The `find` command scans a given directory looking for matching files, and recursively processes subdirectories as they are appearing. `find` also creates a new process to execute the command given through the `-exec` option (`cat` in our case) on every matching file. As a consequence, the auxiliary processes created by `find` usually read nearby blocks, because, in an Ext3 file system, regular files in the same directory are usually stored together in disk, in the group assigned to the directory [41].

Regarding the I/O schedulers, when two or more processes are run in this test, AS exploits the spatial locality in a better way than CFQ. The former usually performs less disk seeks, because it sorts requests by physical location on disk, and waits a small interval of time for requests close to the last one dispatched [42]. The latter uses a round robin policy to choose the processes to attend [42, 99]. CFQ's behavior implies more disk seeks, and, can mean an increase in application and I/O times. Therefore, although REDCAP presents a quite similar behavior with both schedulers, AS provides higher improvements.

An additional point is that, with the original kernel, the request size is, on average, 82 sectors, which means a sequential access to several files. The contribution of REDCAP for sequential workloads is rather small since the prefetching technique of both the operating system and the disk cache is optimized for this access pattern.

The second exception appears with the AS scheduler and 1 process. In that case, REDCAP performs worst than a normal system and the application time is increased by 11%, despite it achieves an improvement of 20% in the I/O time, as we can see in Figure 3.14.

In order to understand the results obtained with AS, it is necessary to explain two aspects of the Linux operating system and the AS scheduler. First, when a new asynchronous operation is issued, if there is no pending requests (the scheduler queue is empty), the Linux kernel plugs the block device delaying I/O operations, that will be performed later [51, 52]. The goal of this action is to increase the chances of clustering requests for adjacent blocks, and, if a new I/O operation is issued to an adjacent block, the two requests will be merged into a

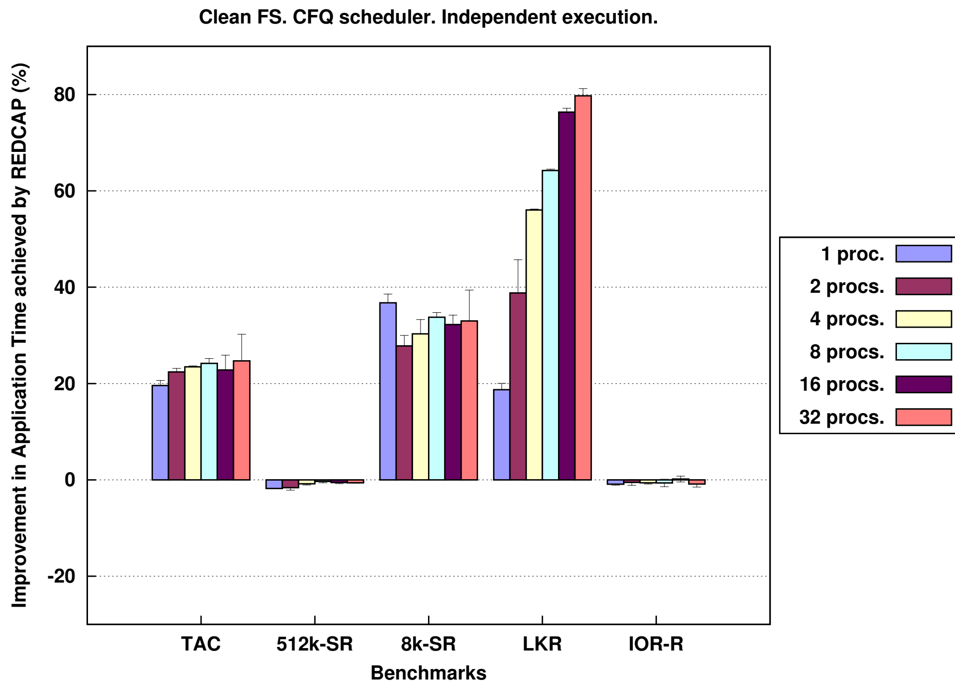


Figure 3.9: Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed independently, on the clean file system and with the CFQ scheduler.

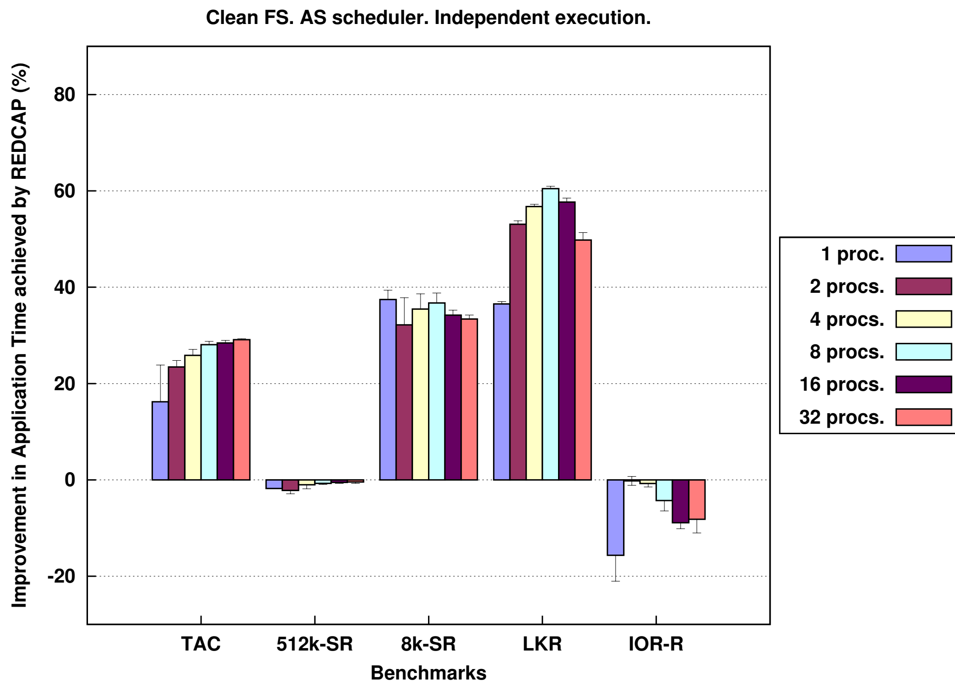


Figure 3.10: Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed independently, on the clean file system and with the AS scheduler.

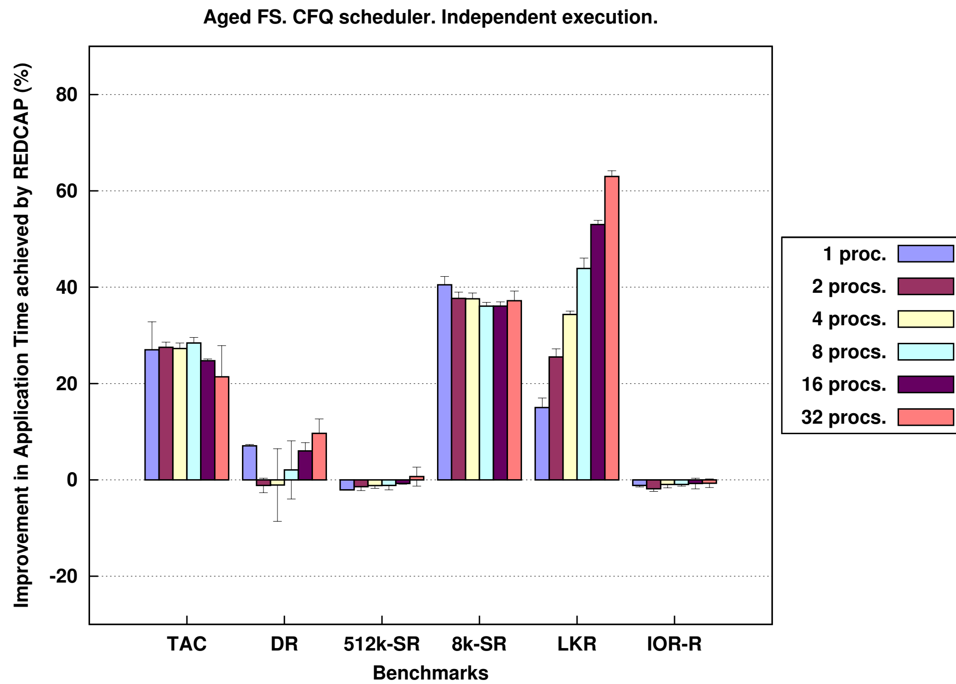


Figure 3.11: Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel, when benchmarks are executed independently, on the aged file system and with the CFQ scheduler.

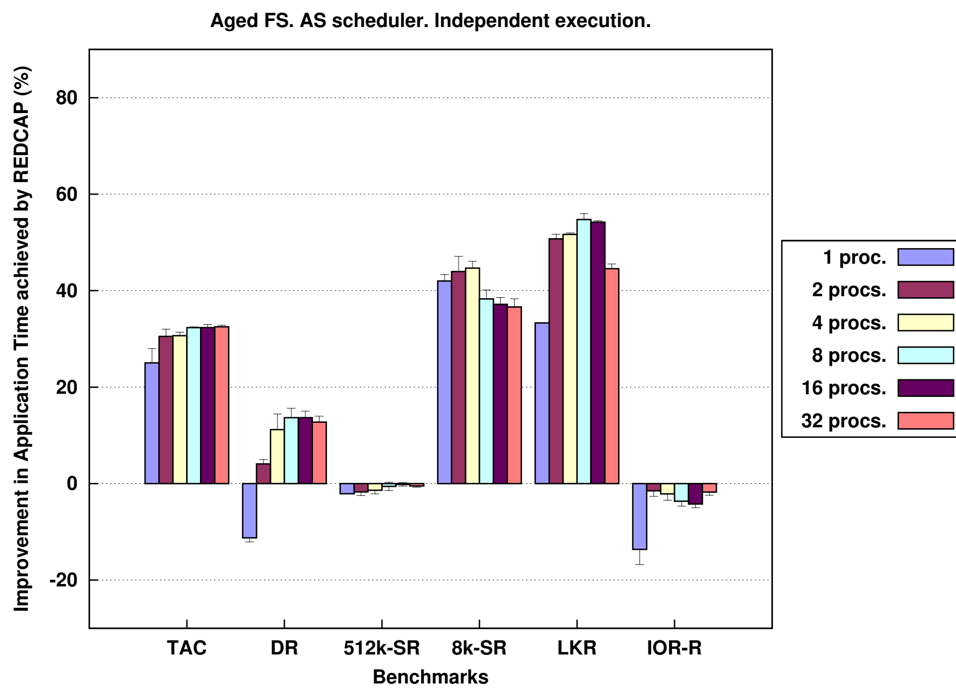


Figure 3.12: Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed independently, on the aged file system and with the AS scheduler.

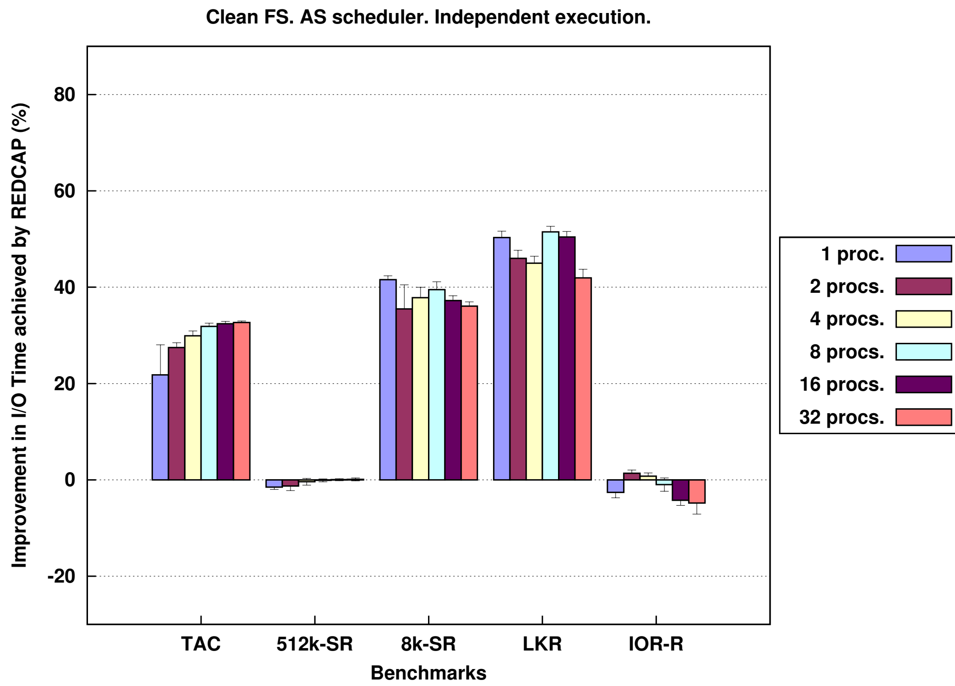


Figure 3.13: Improvement, in the I/O time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed independently, on the clean file system and with the AS scheduler.

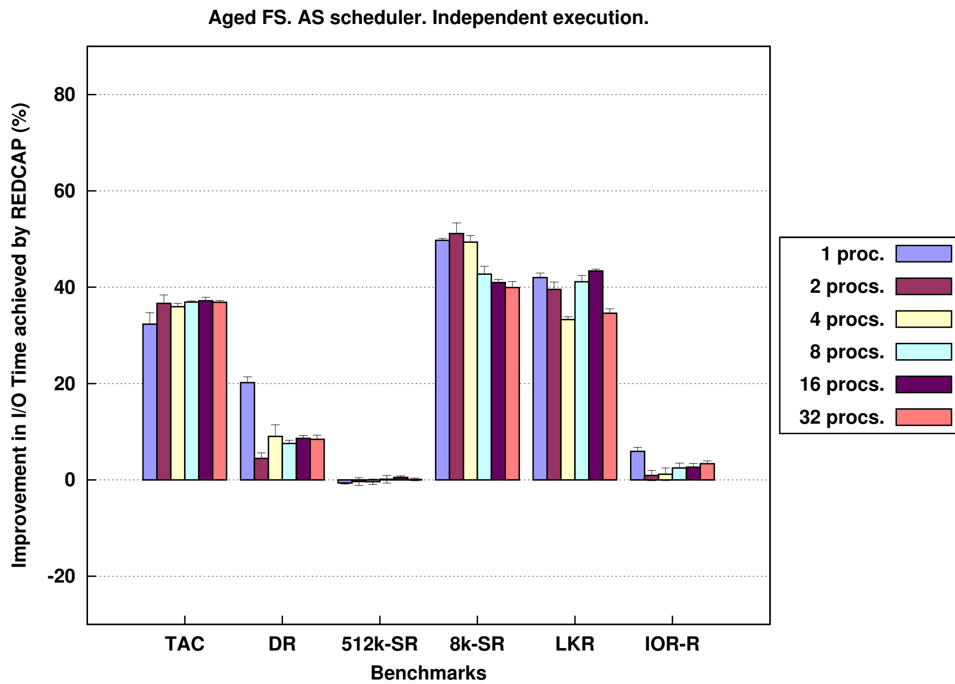


Figure 3.14: Improvement, in the I/O time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed independently, on the aged file system and with the AS scheduler.

single one (if both operations have the same type). The device usually remains plugged for a small time interval, normally 3 ms.

Second, the AS scheduler selectively stalls, also during a small time interval, the block device right after servicing a request in the hope that a new one for a nearby sector will be soon posted [47]. When a new request is scheduled that meets this condition, if the device is plugged, the scheduler unplugs it to serve the request.

These two policies produce an odd effect for sequential access patterns when REDCAP is on. Most of the issued requests have a large size, due to the prefetching performed by the operating system. On the active state, these requests usually need two REDCAP segments: one is a cache hit, and the other a miss. The cache hit means the copy of data from the REDCAP cache to the request's buffers, whereas the cache miss produces a disk read operation. Although the copy time is very small, when the REDCAP request for the cache miss is issued and arrives to the scheduler queue, the AS time interval has expired. If there are no more pending requests, the new one is not dispatched to disk until the device is unplugged, what increases application time. I/O time, nevertheless, does not usually increase, as we can see in Figure 3.14. Moreover, since the device plugging delays the read request, it could be merged with other read requests produced by the operating system's prefetching. This generates larger requests which can improve the I/O time.

The above problem appears in this benchmark with AS and 1 process because the access pattern is mainly sequential: although the directory has a size of 2.35 GB and 3528 files, 7 of those files add up to 2.3 GB.

512 kB Strided Read. For this access pattern, our mechanism provides no contribution because its cache is not effective. The algorithm detects this fact and turns it off, which is inactive almost all the time. The REDCAP–VD kernel behaves quite similar to the original kernel, and, statistically, both have the same performance. As we can see, our method gets the worst results for 2 processes, and the AS scheduler and the clean file system, and for 1 process, and both schedulers and the aged file system, with a degradation of only 2%. This small degradation is due to the time initially lost by REDCAP while the cache is active at the beginning of the test.

8 kB Strided Read. With the 8 *kB Strided Read* benchmark, REDCAP always performs better than the vanilla kernel for both file systems and both schedulers. For this access pattern, REDCAP achieves its best results when its cache is always active, as it happens in this case. The operating system does not detect this access pattern nor does implement any technique to enhance the performance under this sort of access. However, with our technique, most of the requests take advantage of the prefetching performed by REDCAP, since almost nine out of every ten application requests are cache hits. For the clean file system, reductions of up to 36.8% and 37.5% are achieved for CFQ and AS, respectively, and for the aged file system, they are of up to 44.5% and 40.5%.

Linux Kernel Read. The REDCAP–VD kernel always performs better than the original one with the *Linux Kernel Read* benchmark. Our cache is active all the time for both schedulers and file systems, what allows it to minimize the application time. For CFQ, REDCAP shows a qualitatively similar but quantitatively different behavior under both file systems, presenting

significant improvements, that increase as the number of processes grows. For 32 processes, the application time is reduced by up to 79.2% and 63% for the clean and aged file systems, respectively. When the AS scheduler is used, REDCAP improvements present roughly the same behavior, except for 1 process, with an application time reduction above 50% for 2, 4, 8 and 16 processes. Maximum reductions are 60.5% and 54.8% using the clean and the aged file systems, respectively, and for 8 processes in both cases.

An interesting point with this benchmark is that, although, for the CFQ scheduler, REDCAP provides the highest application time reduction, on average the improvement is larger for AS. Indeed, for 1, 2, 4, and 8 processes, the time reduction obtained is significantly larger for AS than for CFQ. However, REDCAP behaves the same for both schedulers, and almost the same percentage of requests are cache hits in both cases. The difference is due to the access pattern and the behavior of the schedulers, and not to REDCAP itself.

During the execution of this benchmark, each process reads all the files in a Linux kernel source tree by means of the `find` and `cat` commands (see Section 2.4.3 for the exact command line). As we have mentioned for the *Directory Read* test, the `find` command scans a given directory in search for matching files, and recursively processes subdirectories as they are appearing. A new process is also created by `find` to execute the `cat` command on every matching file. Again, these auxiliary `cat` processes usually submit nearby blocks because files in the same directory are usually stored together in disk due to the block group division and allocation policy performed by Ext3 [41]. Furthermore, in this LKR benchmark, every `cat` process only submits a few I/O requests, because file sizes are, on average, small.

As aforementioned, when traversing a directory tree, AS exploits the spatial locality in a better way than CFQ. In this test, the difference between both schedulers is even greater, since a large amount of small files is read. Hence, when comparing for this benchmark the performance of AS and CFQ on the original kernel, AS significantly outperforms CFQ, except for 1 process. Therefore, although REDCAP provides the same behavior for both I/O schedulers, the improvements are larger with AS than with CFQ.

IOR Read. Finally, in the case of *IOR Read*, the prefetching techniques implemented by both the operating system and the disk cache are optimized for its sequential access pattern. Moreover, due again to the prefetching of the operating system, most of the read requests issued in this test have a size of 128 kB (maximum disk request size allowed by the file system), which is the same size as REDCAP requests (i.e., REDCAP segments). Therefore, the contribution of our technique is rather small, and even a copy time is added to each cache hit. Because the I/O times of the real and virtual disks are very similar, sometimes the activation–deactivation algorithm alternates the state of the REDCAP cache between active and inactive. However, the right decision would be to keep the cache inactive.

For CFQ and both file systems, the behavior of the REDCAP–VD kernel is almost equivalent to that of the original kernel. Taking into account the confidence intervals, we can conclude that both kernels present the same performance.

The situation is slightly different when the AS scheduler is used. In this case, for 1 process a degradation of up to 15.6% and 13.6% is produced for the clean and aged file systems, respectively. However, if we compare I/O times, a degradation of only 2.6% is obtained for the clean file system, whereas, for the aged one, this time is improved by 6%, as we can observe in Figure 3.14. The reason is the same as described for the *Directory Read* benchmark with

AS and 1 process. The submitted requests are delayed into the scheduler queue until the device is unplugged. This delay implies an increase in the application time, but not in the I/O time.

When there are two or more processes, a different problem appears. When a new request arrives, the probability of finding the scheduler queue empty is small, and the device is rarely plugged. Therefore, when the AS time interval expires, a request of a different process is dispatched to disk (because a request of the current selected process has not been submitted yet). This behavior requires more disk–head movements that could cause an increase in seek time. Moreover, the increase in the number of seeks can downgrade the performance of the prefetching performed by the disk drive itself, because prefetched data can be evicted from the disk cache before being read, especially if the disk controller performs read–ahead for each request. Due to this problem, for 8, 16 and 32 processes a degradation of up to 4.3%, 8.9% and 8.2% is produced for the clean file system, respectively, and of up to 3.7%, 4.2%, and 1.8%, for the aged file system, respectively. When comparing I/O times, a degradation of up to 4.8% is obtained for the clean file system and 32 processes, but, for the aged file system, the I/O time is not increased, and even it is slightly improved (up to 3.4% for 32 processes).

All the benchmarks in a row.

Now we analyze the case of the benchmarks executed in a row. Application time improvements achieved by our technique with respect to the original kernel for the clean file system, and for the CFQ and AS schedulers are presented in Figures 3.15 and 3.16, respectively. The results for the aged file system are depicted in Figures 3.17 and 3.18. Again, to explain the results obtained with the AS scheduler, improvements in the I/O time achieved for the clean and aged file systems are showed in Figures 3.19 and 3.20, respectively.

The REDCAP–VD kernel presents an equivalent behavior to that obtained when the benchmarks are run independently. Improvements of up to 32.7%, 42.2% and 76.6% are achieved for the TAC, 8k–SR and LKR benchmarks, respectively. For the DR test, only run in the aged file system, it gets improvements of up to 12.1%. With 512k–SR and IOR, except for a couple of cases, the behavior of our mechanism is quite similar to the vanilla kernel's one. Hence, according to these results, we can claim that our virtual disk adapts very quickly to the workload changes that are caused by the execution of the benchmarks in a row. Only minor differences are observed due to both the buffer and REDCAP caches. Those differences are explained below.

Clean file system and 1 process. When the first benchmark, TAC, has finished, a significant amount of file blocks are already in the buffer cache, because the system has 1 GB of RAM memory, and the file is also 1 GB in size. Therefore, the 512k–SR benchmark has to read only a small amount of data from the end of the file (due to the backward access pattern of TAC, the last blocks of the file were evicted from memory). As we have explained, 512k–SR performs four read series on the file. With the REDCAP–VD kernel, after the first series, almost all the blocks requested by the other three series are either in the buffer cache or in our REDCAP cache (it has enough segments to accommodate the small amount of data read from disk). The operating system, however, does not perform any prefetching (due to the strided access pattern), and the original kernel has to read all the blocks of the four series

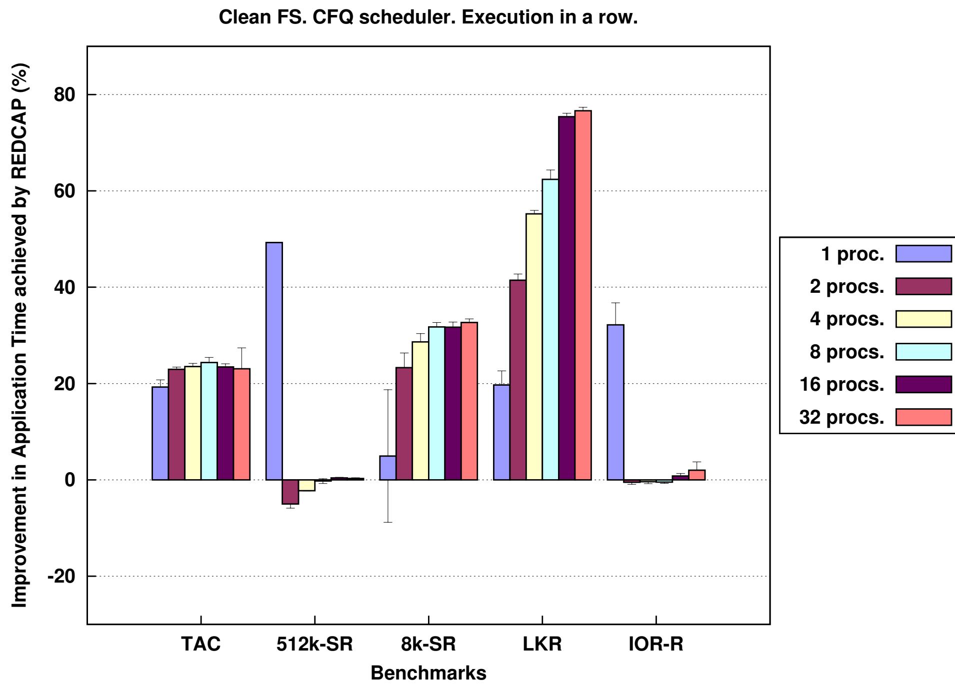


Figure 3.15: Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed in a row, on the clean file system and with the CFQ scheduler.

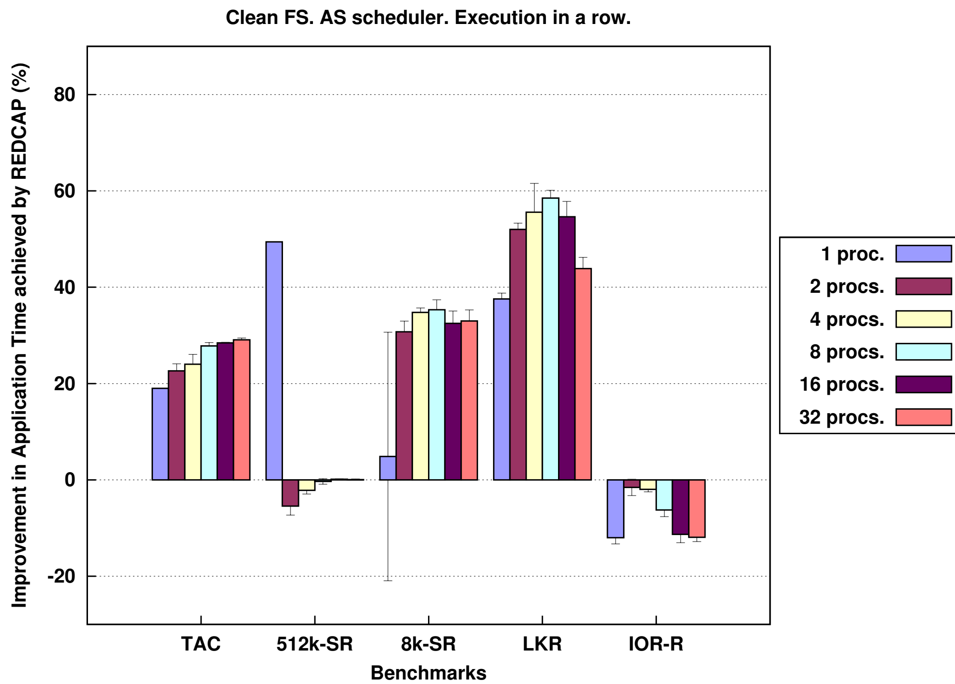


Figure 3.16: Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed in a row, on the clean file system and with the AS scheduler.

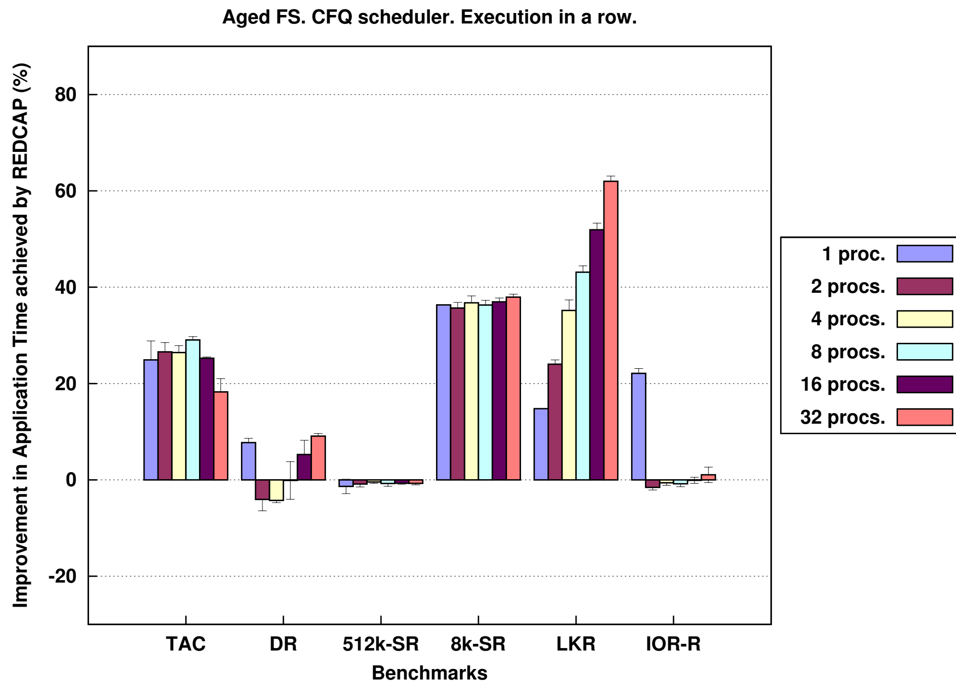


Figure 3.17: Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed in a row, on the aged file system and with the CFQ scheduler.

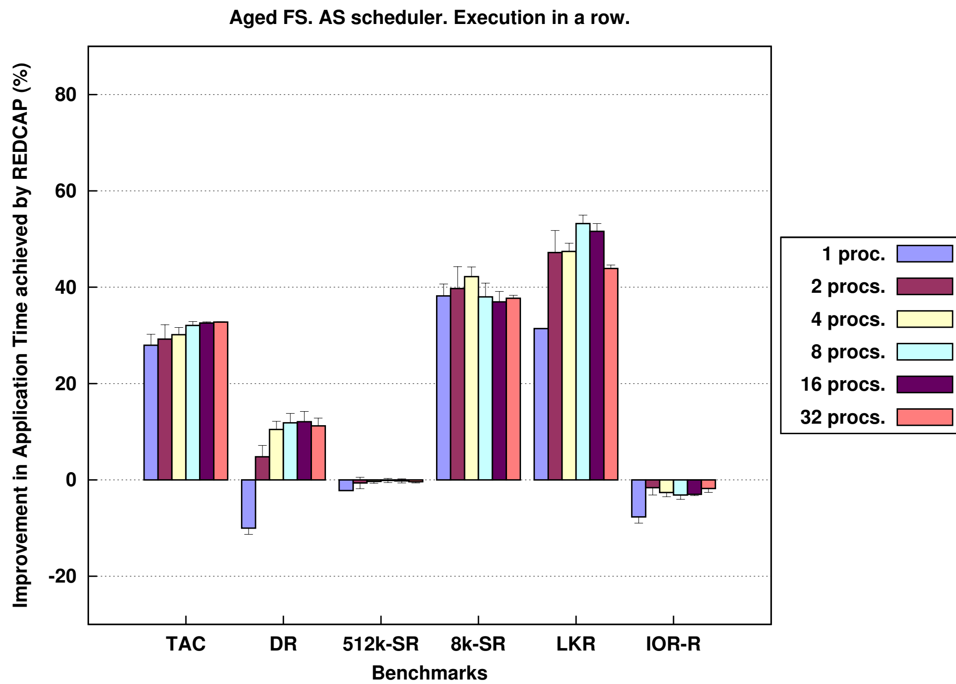


Figure 3.18: Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed in a row, on the aged file system and with the AS scheduler.

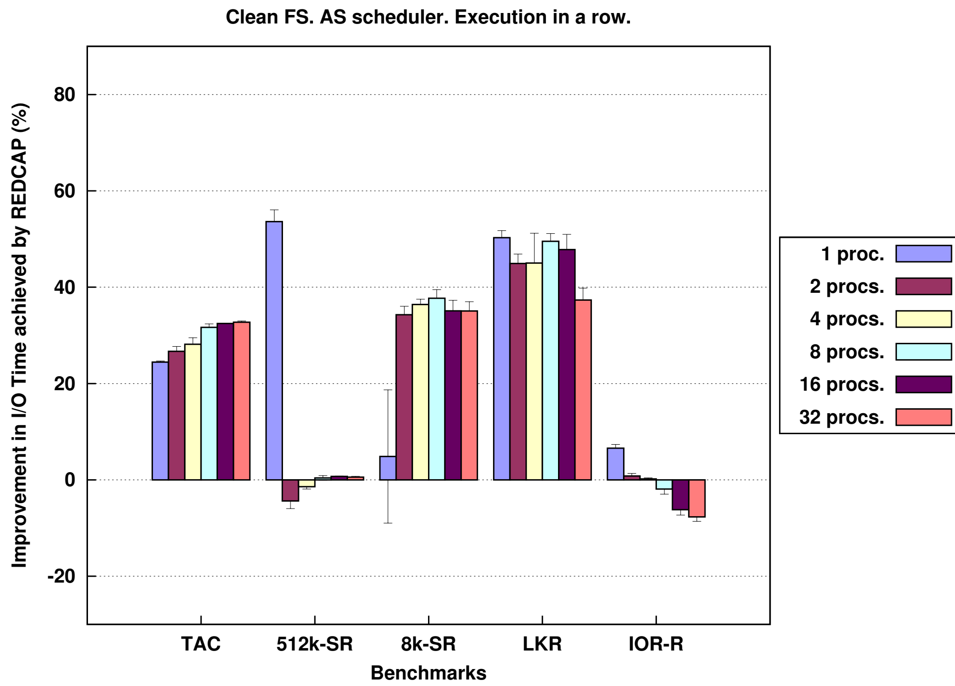


Figure 3.19: Improvement, in the I/O time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed in a row, on the clean file system and with the AS scheduler.

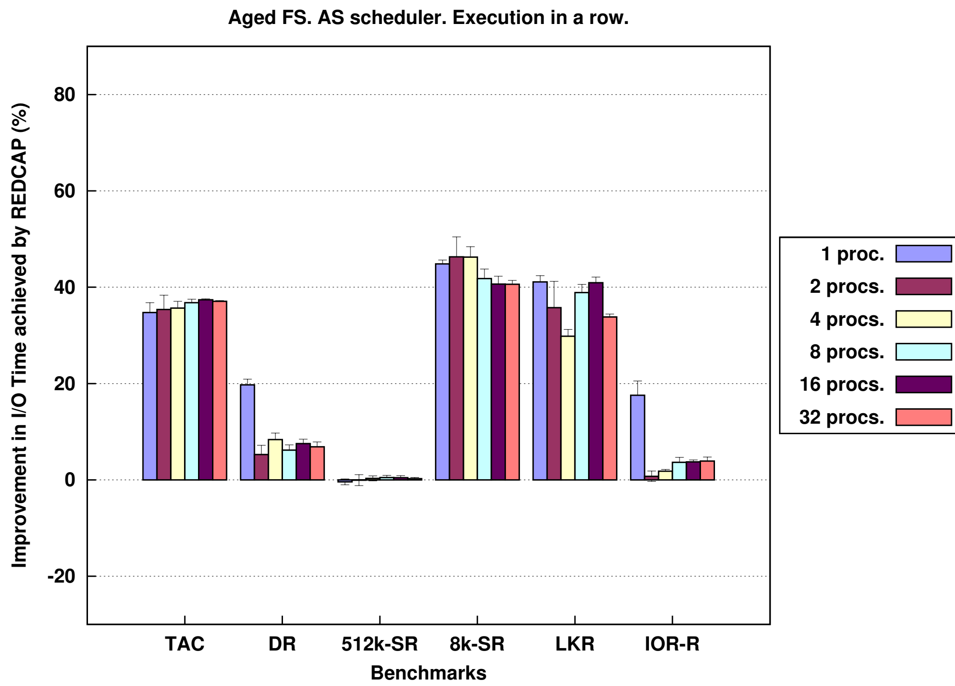


Figure 3.20: Improvement, in the I/O time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed in a row, on the aged file system and with the AS scheduler.

located at the end of the file by means of independent requests. For this reason, REDCAP unexpectedly gets an application time improvement of 50%.

When the next test, 8k–SR, is executed, our method reads more blocks than the original kernel and the improvement achieved is reduced from 37% (when the test is executed "alone") to 5%. This is due to the size of the kernel image. The amount of memory needed by the REDCAP–VD kernel image is larger than that needed by the original kernel one, because the former uses RAM memory to store the REDCAP cache and the disk simulator. After the execution of the two first benchmarks, with the REDCAP–VD kernel, there are less blocks of the file in memory and more blocks have to be read.

At the end of the 8k–SR benchmark, 4 out of every 12 blocks (of 1 kB) of the file are in RAM, that is approximately 341 MB of the file. Since the next test, LKR, uses only 344 MB of memory, all the 341 MB of the file are still in main memory when the last benchmark, IOR, is run. This produces a "strided" access pattern which prevents the operating system from performing large prefetching requests, being 20 sectors the average request size (when IOR is executed independently the average request size is 235 sectors). However, the REDCAP prefetching is used widely, and our method obtains an unexpected improvement of 32% for CFQ.

When the AS scheduler is used, the execution of the *IOR Read* benchmark produces an increase in the application time of up to 12%, but an improvement of up to 6.6% in I/O time, as we can see in Figure 3.19. The reason of this data is the same as given for the *Directory Read* and *IOR Read* benchmarks with the AS scheduler. The effect can also be observed for 8, 16 and 32 processes.

Clean file system and 2 processes. At the end of the TAC execution, parts of the files read by the two processes are in memory, but, due to the size of the kernel images, there are less file blocks in memory with the REDCAP–VD kernel than with the original kernel. Therefore, when the 512k–SR benchmark is executed, REDCAP reads more blocks than the vanilla kernel. However, unlike for 1 process, the blocks to be read from disk do not fit in the REDCAP cache. In this case, a degradation of up to 5% is produced for both schedulers. For 4 and more processes, although the same effect happens, the degradation is insignificant, because the extra blocks that REDCAP has to read represents a small percentage of the total.

Aged file system and 1 process. In this case, there is an unexpected result only for the IOR test and 1 process. As in the clean file system, the problem is that the operating system cannot perform large prefetching requests, whereas the REDCAP prefetching is completely exploited, achieving an application time reduction of 22% for CFQ. By using the AS scheduler, our proposal presents a degradation in the application time of 7.7%, but an improvement of 17.6% is obtained in I/O time (see Figure 3.20). The explanation is the same as that given previously in the *Directory Read* and *IOR Read* benchmarks.

The 512k–SR and 8k–SR benchmarks present the same behavior as when they are executed independently. The DR test reads a directory of 2.35 GB, and all the blocks read by the TAC benchmark are evicted from memory before the strided read tests are executed.

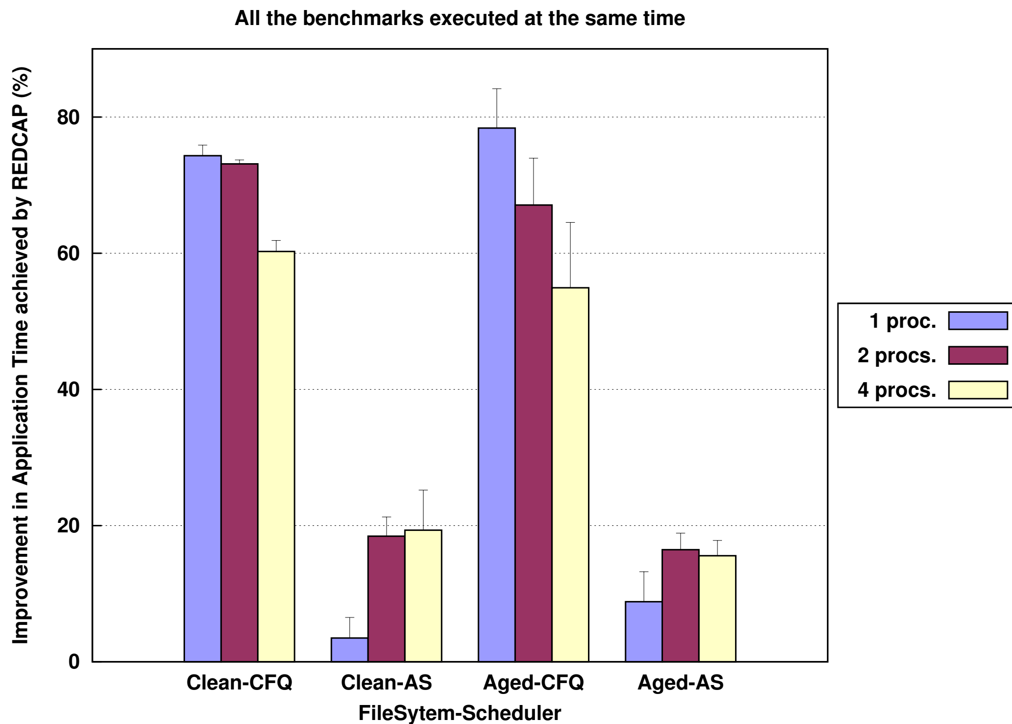


Figure 3.21: Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed at the same time.

All the benchmarks at the same time.

Results for application times achieved by REDCAP, as compared to the original kernel's ones, when all the benchmarks are executed at the same time, are presented in Figure 3.21. Data is grouped in the figure by file system and scheduler.

Our technique always performs better than the vanilla kernel, although its improvement depends on the I/O scheduler used. REDCAP presents its best behavior for the CFQ scheduler, reducing the application time by up to 80%. In this case, improvements slightly decrease as the number of processes increases, although for 4 processes per benchmark (20 or 24 processes altogether, depending on the file system used) reductions of 55% and 60%, for the clean and aged file systems, respectively, are still achieved. For the AS scheduler, improvements of up to 19.3% and 16.5% are obtained for clean and aged file systems, respectively. Again, although REDCAP provides the highest reductions for CFQ, its behavior is the same for both schedulers. Hence, the mean percentage of activation is almost equal: 92% for CFQ and 91% for AS on the clean file system. The different application time reductions, however, are due to the performance of the schedulers in this test. Table 3.4 depicts the average application time measured during the execution of this test for both schedulers and both file systems. As we can observe, application times achieved by the original kernel for AS are quite small compared to those obtained for CFQ. In this test, the AS scheduler behaves quite well, and, due to the workload produced by running all the benchmarks at the same time, the contribution of REDCAP, although noticeable, can not be higher for this scheduler.

Table 3.4: Average application time measured during the execution of the *All the benchmarks at the same time* test, for 1, 2, and 4 processes.

(a) Clean file system

Scheduler	Kernel	Processes		
		1	**2**	**4**
CFQ	REDCAP–VD	1578.13 s	4582.22 s	11444.07 s
	Original	6122.70 s	17183.80 s	22807.35 s
AS	REDCAP–VD	910.31 s	1805.04 s	3323.07 s
	Original	934.55 s	2208.32 s	4090.08 s

(b) Aged file system

Scheduler	Kernel	Processes		
		1	**2**	**4**
CFQ	REDCAP–VD	1501.36 s	5602.43 s	12689.72 s
	Original	6993.03 s	16948.33 s	28137.86 s
AS	REDCAP–VD	879.05 s	4523.09 s	8000.82 s
	Original	971.72 s	5421.19 s	9417.00 s

3.6. Solid–State Drives

SSD disks have several advantages over hard disks, but, from the point of view of this work, the most important one is their performance. Due to the absence of mechanical components, they have no seek time or rotational latency, and usually provide very high bandwidths compared to traditional disks.

SSDs are storage devices that use solid–state memory to store "persistent" data, and usually are composed of four key components [93, 100, 101, 102]:

- An array of NAND flash memories that are used as storage medium, and are connected in parallel.
- A host interface logic that performs the communication to the host via an interface connection (SATA, USB, etc.).
- An SSD controller, called Flash Translation Layer (FTL), that manages flash memory space, translates I/O requests into flash memory operations, controls data transmission, handles garbage collection, and so on.
- And a small cache that temporally holds data, and is used for executing write operations more efficiently [103]. But, not all SSDs have this internal buffer, and, to the best of our knowledge, they do not perform any prefetching.

An interesting point is that SSDs are addressed in LBA mode because they behave much like traditional hard disks and hide their internal components and features.

NAND flash memories are classified into two types: Single–Level Cell (SLC) and Multi–Level Cell (MLC) [93, 100]. In an SLC flash, each memory cell represents a binary value, and in an MLC one, each cell uses multiple levels and allows more bits to be stored using the same number of transistors. When comparing SLC and MLC, the former is faster in terms of writes, and needs less power. The latter provides larger capacity, and is cheaper.

In this section we are going to discuss how the in–kernel disk simulator can be used with SSD devices, the experimental environment to test them, the accuracy of the disk model for these devices, and, finally, the behavior of REDCAP and the virtual disk when the underlying device is an SSD.

3.6.1. Viability of the virtual disk for SSDs

Firstly, we have to consider whether the in–kernel disk simulator, proposed in Section 3.2, can also simulate the behavior of SSD devices. For this purpose, the most important aspect is to justify the validity of the disk model.

Even though the time taken by an I/O operation may depend on several factors, our disk model only takes three input parameters: the type (read or write), the request size, and the inter–request distance from the previous request.

With respect to the operation type, although SLC flash models may have a balanced read/write performance, writes are often much slower than reads. Again, two ables, one for each operation type, are needed to predict I/O times.

The disk access time of a request depends on the number of bytes transferred to/from the SSD device, so the request size is an important factor to model the behavior of these storage devices too, and it should be considered as input parameter.

Regarding the inter–request distance, since SSD devices are random–access devices, and do not have seek and rotational delays, the I/O time of an operation should not depend on the logical distance from the previous one, and the inter–request distance could be dismissed as a parameter of the disk model. However, several tests performed on these devices have shown that, for requests of the same size, there are small differences in I/O time between a sequential and a random access pattern. Therefore, we use the same model for both hard and SSD drives, although tables can have quite similar columns in the case of an SSD. In Section 3.6.3, we prove that the proposed table model accurately models the behavior of the SSD devices too.

The dynamic update of the disk model is also important for SSD drives. Although the service time of a request is mainly determined by the request's size and type in SSDs, this time usually increases with the use of the drive (unless the drive is reformatted through the `TRIM` command). For instance, between the first read table obtained by the training program after creating the file system and the files for the tests, and a second read table obtained by the same program after running all the benchmarks, the difference per cell is, on average, of 10%, and there is maximum differences up to 44%. This increase in service time is due to the performance degradation that SSD drives suffer as a result of their wear leveling and write combining mechanisms.

Other aspects of the in–kernel disk simulator (I/O schedulers, request management, training program, etc.) are valid for the SSD devices, because these features are not involved in the behavior of the device itself, but with the "right simulation" of the I/O process. Thus, we can claim that the virtual disk can also be used for simulating the behavior of SSDs.

Table 3.5: Specifications of the test SSD disks.

Features	Values	Values
Disk	Intel SSDSA2MH160G2C1	Intel SSDSA2SH064G1GC
Capacity	160 GB	64 GB
Flash memory	MLC	SLC
Read latency	65 μs	75 μs
Write latency	85 μs	85 μs
Sustained sequential reads	250 MB/s	250 MB/s
Sustained sequential writes	100 MB/s	170 MB/s
Nick name	"SSD–160"	"SSD–64"

As we have said, traditional hard disks are rather slow compared to CPU: I/O time to serve requests is huge compared with CPU time. Since the benchmarks used are I/O intensive (I/O–bound), the CPU is usually idle and waiting for disk operations. Therefore, the CPU is often assigned to the kernel thread of the virtual disk which can perform its tasks without delay. However, SSD disks are much faster than regular disks, although not as fast as CPU. Now, processes can issue a large amount (more than four thousands) of I/O operations per second. This high throughput results in almost no CPU idle time, because the request management requires more computation time. As a consequence, the CPU is rarely assigned to the kernel thread of the virtual disk, because there are more processes fighting for the CPU time, and thousands of I/O IRQs are handled per second. The virtual disk becomes slow, and can not do its evaluation on time. To solve this problem, we have set a higher priority, -20, for the kernel thread.

3.6.2. Experiments and methodology

This section describes the disks, benchmarks and I/O schedulers used for analyzing the behavior of our proposals with SSDs.

Two SSD disks have been tested in our experiments. Table 3.5 presents their main features. The first one is an Intel X–25M SSDSA2MH160G2C1 [104], with a capacity of 160 GB, which uses MLC flash memories. The second one is an Intel X–25E SSDSA2SH064G1GC [104] of 64 GB, which is built on SLC flash memories. To short and facilitate the explanation, we refer to these disks as "SSD–160"[2] and "SSD–64", respectively. The experiments are conducted in the computer **Hera** that is described in Section 3.4.1.

The benchmarks used for analyzing the behavior of the virtual disk and REDCAP with SSDs are introduced in Sections 2.4.3 and 3.4.2.

We carry out the experiments using CFQ and Noop as scheduling policies. This selection is based on the fact that the former is the default I/O scheduler in the Linux kernel 2.6.23,

[2]Note that in the name "SSD–160", "SSD" comes from Solid–State Drive, and "160" is the disk capacity.

and the latter usually achieves the best performance for SSD devices compared to the other available Linux schedulers [105, 106].

3.6.3. Accuracy of the virtual disk model with SSDs

Again, we have run the *All the benchmarks in a row* test to evaluate the accuracy of the disk model for SSD devices. The SSD and virtual disks serve the same requests, and REDCAP is neither active nor simulated. I/O times achieved by the two disks provide the comparison to perform the analysis. Remember that I/O time of the real disk used for performing the comparison, is the time given by the disk, and not the time from the table model.

The study has been done for the SSD–160 disk, the CFQ scheduler and 1, 8 and 32 processes. As we did for the hard disks, we have evaluated the four configurations of the virtual disk based on the number of values averaged per cell in the tables: eight ("VD 8" in Figure 3.22), sixteen ("VD 16"), thirty two ("VD 32"), and sixty four ("VD 64") values per cell.

Figure 3.22 presents the differences, in percentage of I/O time, of the virtual disk with respect to the real one. In the figure, the end of a benchmark and the beginning of the next one have been marked with a vertical black dashed line.

As we can observe in the graphs, the virtual disk behaves very much like the SSD device, and the difference among both disks is, on average, less than 0.3%. Major differences are observed just at the beginning of the test execution, when TAC is run, and appears because the values of the cells are still not updated to the current behavior of the SSD device. Differences decrease quickly, and the maximum difference is only 1.7%.

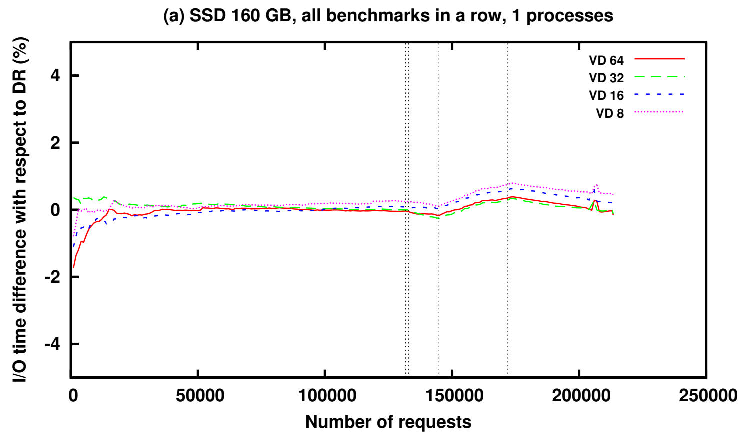
An interesting point is that differences among the four configurations of the virtual disk tested are negligible.

3.6.4. Performance of REDCAP on SSDs with the virtual disk

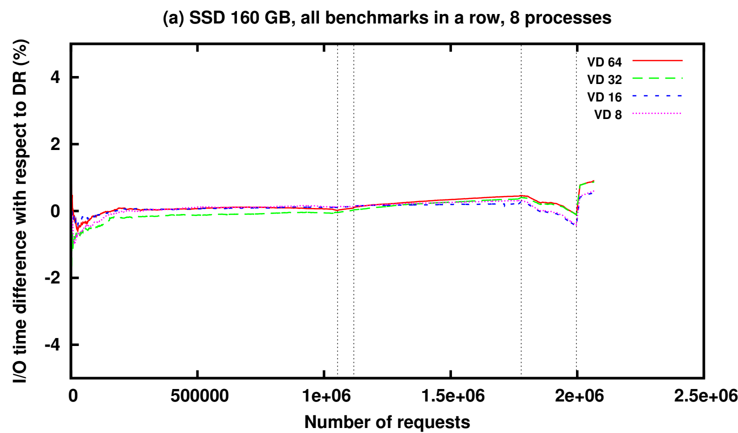
By using the REDCAP–VD kernel, we have analyzed the performance of REDCAP and the behavior of the virtual disk with SSDs. Again, the results have been compared to those obtained with the original kernel. The experiments have been carried out with the experimental conditions described in Section 3.5.2. The main aspects are:

- The REDCAP cache size has been fixed to 64 MB, and the segment size to 128 kB.
- The results showed are the average of five runs. The confidence intervals for the means, for a 95% confidence level, are also included as error bars.
- All tests have been performed with a cold page cache and a cold REDCAP cache.
- The tables of the off–line training are given to the virtual disk each time the system is initialized.
- The initial state of REDCAP is active.

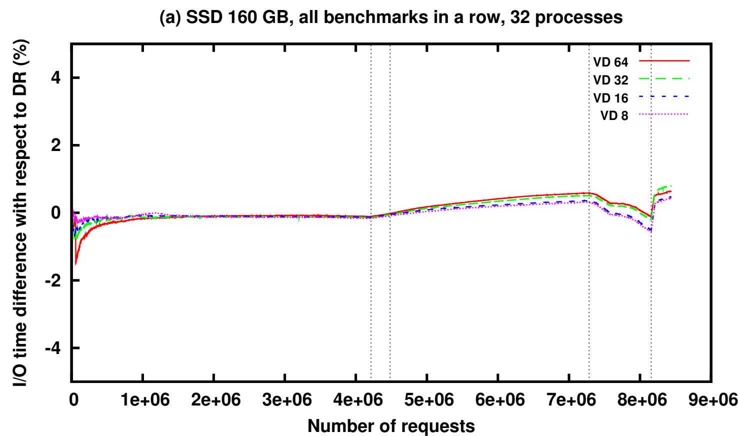
Before explaining the results in detail, it is important to clarify two key aspects. Firstly, when comparing Linux I/O schedulers on SSD devices, Noop and Deadline usually outperform CFQ and AS [102, 105, 106, 107]. The problem with CFQ and AS is that they insert delays with the hope of minimizing the seek time. Consequently, the delays can increase the application time without reducing the I/O time. We have performed several tests with the original kernel, CFQ and Noop, and the SSD disks tested in this work, and the results confirms that Noop outperforms CFQ with respect to the application time, although the



(a) 1 process.



(b) 8 processes.



(c) 32 processes.

Figure 3.22: Difference, in percentage of I/O time, of the virtual disk with respect to the real disk in the *All the benchmarks in a row* test, for the SSD–160 disk, CFQ, and 1 (a), 8 (b) and 32 (c) processes. Vertical dashed lines mark the end of a benchmark and the beginning of the next one. Benchmarks conforming the *All the benchmarks in a row* test are executed in the order TAC, 512k–SR, 8k–SR, LKR, and IOR. Note that the range of Y axis is $[-5, 5]$.

I/O time achieved by both schedulers is almost identical. The I/O scheduler used also has influence in the results obtained by REDCAP. Improvements in application time are usually larger for Noop than for CFQ, whereas improvements in I/O time are almost identical for both schedulers.

Secondly, the overhead introduced by REDCAP and the virtual disk is rather small. However, with SSD disks, this overhead becomes a bit larger, and it is more noticeable, due to the very high performance offered by these devices. As a consequence, application time can be slightly increased, specially in those benchmarks where REDCAP does not provide any improvement.

Benchmarks executed independently

The results for the benchmarks run in an independent way are firstly analyzed. Figures 3.23 and 3.24 show the improvements in application time achieved by REDCAP with respect to the original kernel for the SSD–160 disk, and for the CFQ and Noop schedulers, respectively. Analogously, the results for SSD–64 are presented in Figures 3.27 and 3.28. To explain the results obtained with these devices, the improvements achieved in I/O time are also depicted in Figures 3.25 and 3.26 for the SSD–160 disk, and in Figures 3.29 and 3.30 for SSD–64. Again, all the tests are ordered in the figures in the same order as they are run in *All the benchmarks in a row* test.

As we can observe, the results are quite similar to those obtained for the tested hard drives (see Section 3.5.2), and only a few differences are noticed because of, mainly, the high performance achieved by these devices.

TAC. With the backward access pattern, REDCAP always performs better than the original kernel, although the improvement depends on the I/O scheduler used. The best behavior is achieved for Noop, reducing the application time by up to 80%. For the CFQ scheduler, improvements of up to 60% are obtained. With respect to the I/O time, our approach gets reductions of up to 80% for both disks and schedulers. The REDCAP cache is always active.

512 kB Strided Read. As we already mentioned, for this access pattern, our mechanism provides no contribution because its cache is not effective, being almost impossible to profit the prefetching performed. The algorithm detects this fact and turns it off on the first check, and it is inactive almost all the time. However, REDCAP performs worse than a normal system and the application time is increased up to 38%, and the I/O time up to 22%. These results are easily explained. The overhead introduced by REDCAP and the virtual disk has a negative influence on this benchmark, and the application time is increased. Furthermore, with this benchmark, application times are quite small for both the REDCAP and original kernels (bellow 12 s for 1, 2 and 4 processes), and their absolute difference is also small, giving rise to large relative differences. It is interesting to note that differences are even smaller when we compare I/O times. Table 3.6 shows the average application time measured during the execution of this test for the SSD–160 disk, and Table 3.7 shows the average I/O time for the same disk. There we can see the small execution times obtained, and the small differences, and that the I/O times are almost the same. Similar times are obtained for the SSD–64 disk (not shown).

8 kB Strided Read. With the 8 *kB Strided Read* test, in all the cases, REDCAP performs better than the vanilla kernel. For the SSD–160 disk, reductions of up to 62.5% and 70.4% are achieved for CFQ and Noop, respectively, and for SSD–64, of up to 64.2% and 71.5%, respectively. The reason of these significant improvements is the same given for hard disk drives: the operating system does not perform any prefetching, but, with our technique, most of the requests take advantage of the prefetching performed by REDCAP, and the cache is always active. With these devices, improvements are higher than those obtained with hard devices, because they do not perform any prefetching as traditional ones do.

Linux Kernel Read. When the *Linux Kernel Read* benchmark is run, the REDCAP cache is almost always active for both disks and schedulers, and it gets almost the maximum possible improvements that it could achieve was the cache always on. For SSD–160, reductions of up to 26.8% and 17.7% are achieved for CFQ and Noop, respectively, and for SSD–64, of up to 27.9% and 18.5%, respectively. However, these reductions are not as large as those obtained for the hard drives. The problem is the overhead introduced by the creation of the large amount of small processes to read the Linux kernel source, which is relatively larger with respect to the application time on the SSD disks than on the hard drives. However, if we compare I/O times, reductions are much larger, by up to 74.5%, and they are comparable with those of the traditional devices.

To prove this fact, we have modified this benchmark in such a way that, now, a single process reads all the files of a Linux kernel source tree. We have integrated the `cat` command [44] in the code of the `find` command [108], and, when a regular file is found, the "`cat`" function is invoked. Therefore, the reading process is the same: directory by directory and one file at a time, but no processes are created to read the small files, and just one process performs all the reading. This new benchmark has been run in the same conditions than LKR, and improvements in application time of up to 59% and 68% are achieved for CFQ and Noop, respectively. Reductions in I/O time obtained are quite similar to those achieved with LKR, of up to 76% and 77%, for CFQ and Noop, respectively.

IOR Read. With the sequential access pattern imposed by this test, REDCAP performs worse than a normal system, and application time is increased by up to 5.6%. The overhead introduced by our proposal has a negative impact with this benchmark, by slightly increasing application time. But, these results are also due to the small application times, that give rise to large relative differences. For instance, for the SSD–160 disk, the Noop scheduler, and 32 processes, the biggest absolute difference among application times is just 6.83 s (the times are 163.36 s and 170.19 s with the original and REDCAP–VD kernels, respectively). Moreover, if we compare I/O times, both obtain the same results, as we can see in Figures 3.25, 3.26, 3.29 and 3.30. Therefore, we can say that the behavior of both kernels are quite similar in all the cases.

All the benchmarks in a row

The case of the benchmarks executed in a row is analyzed now. Application time improvements achieved by our technique for the SSD–160 disk, and the CFQ and Noop schedulers are presented in Figures 3.31 and 3.32, respectively. The results for SSD–64 are depicted in

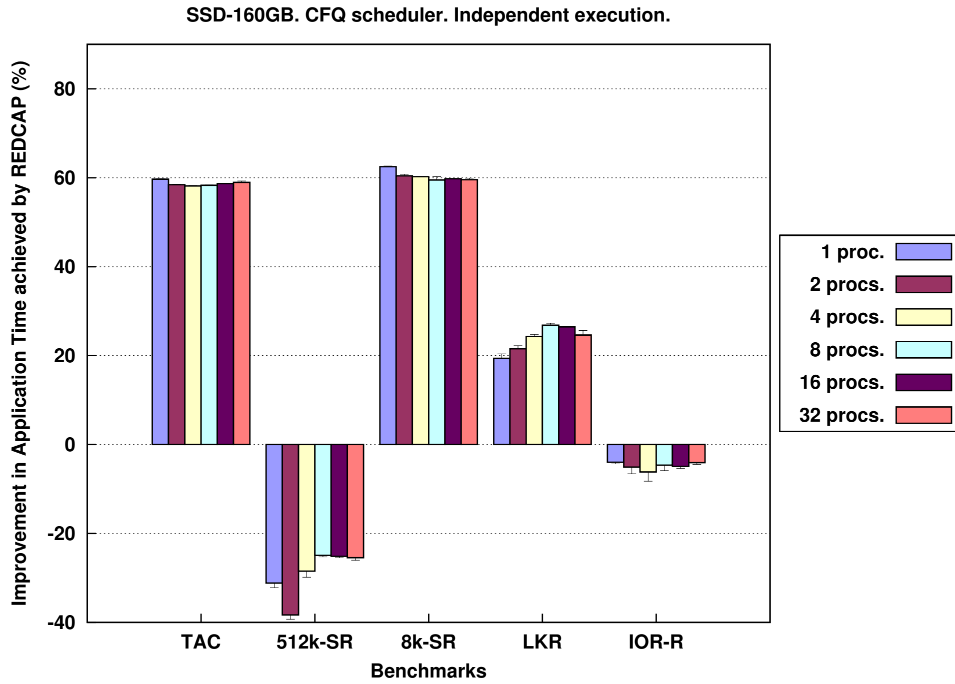


Figure 3.23: Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed independently, on the SSD–160 disk and with the CFQ scheduler.

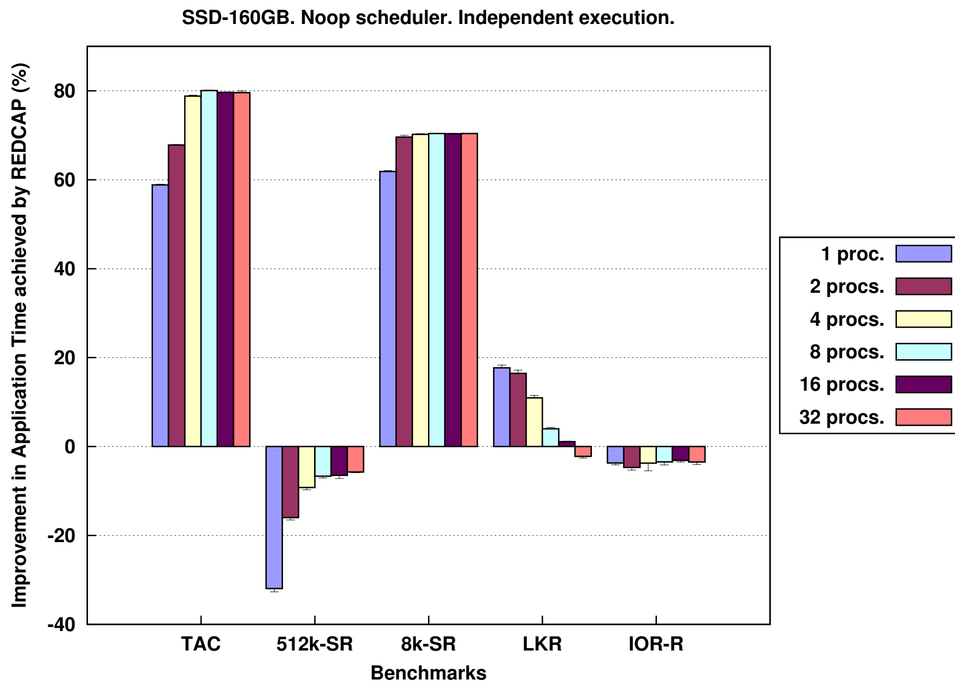


Figure 3.24: Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed independently, on the SSD–160 disk and with the Noop scheduler.

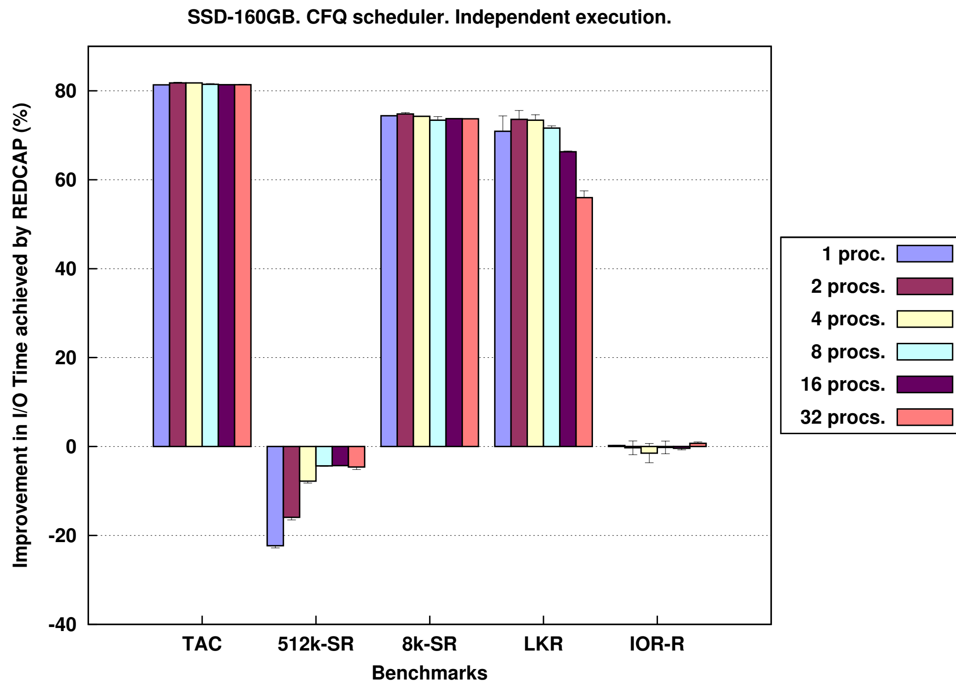


Figure 3.25: Improvement, in the I/O time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed independently, on the SSD–160 disk and with the CFQ scheduler.

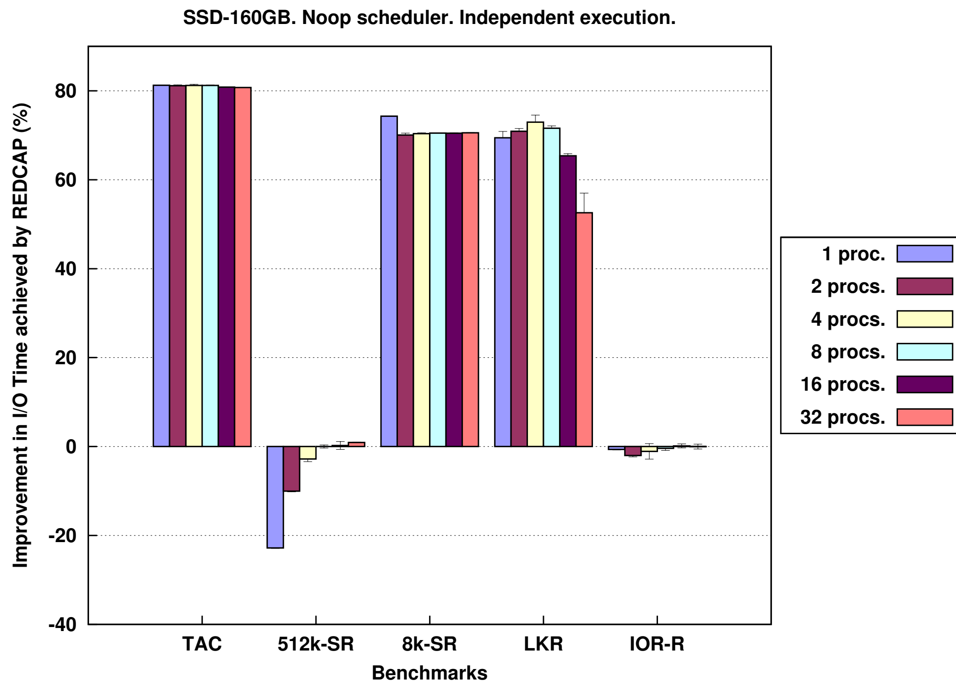


Figure 3.26: Improvement, in the I/O time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed independently, on the SSD–160 disk and with the Noop scheduler.

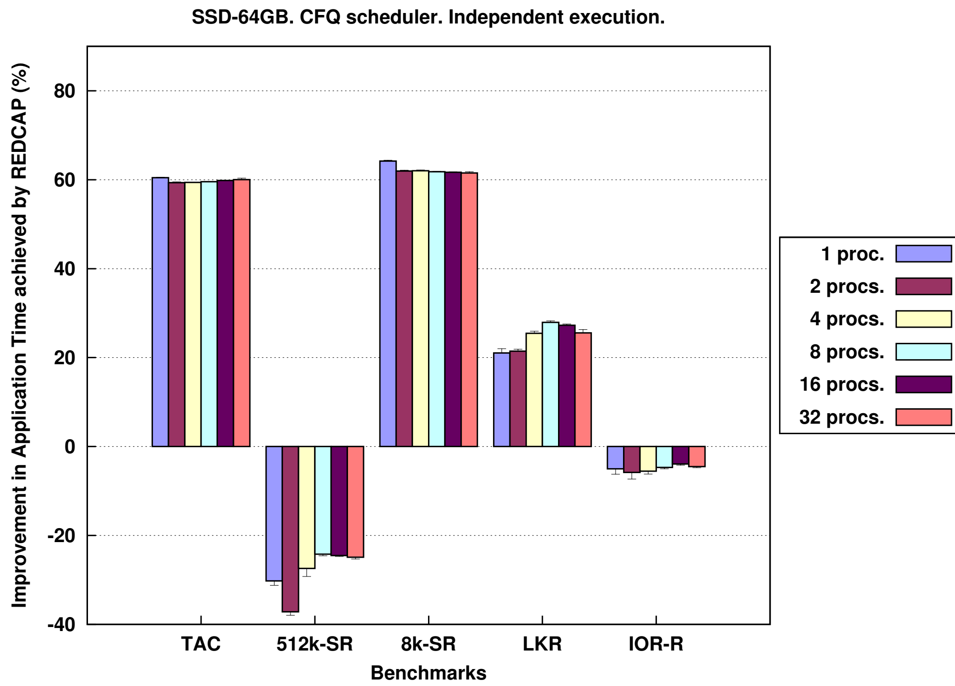


Figure 3.27: Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed independently, on the SSD–64 disk and with the CFQ scheduler.

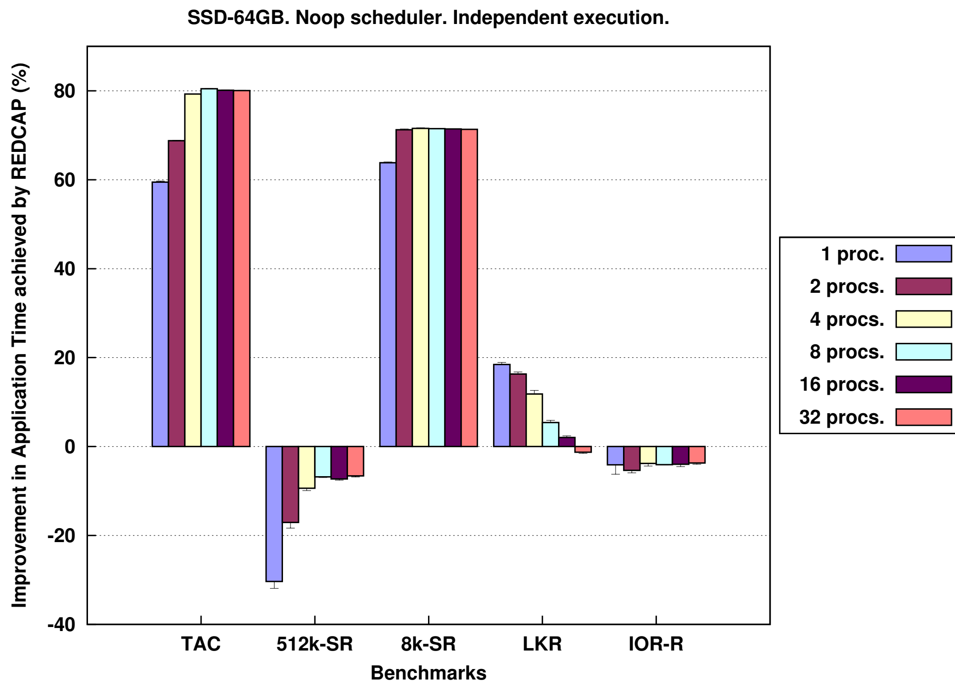


Figure 3.28: Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed independently, on the SSD–64 disk and with the Noop scheduler.

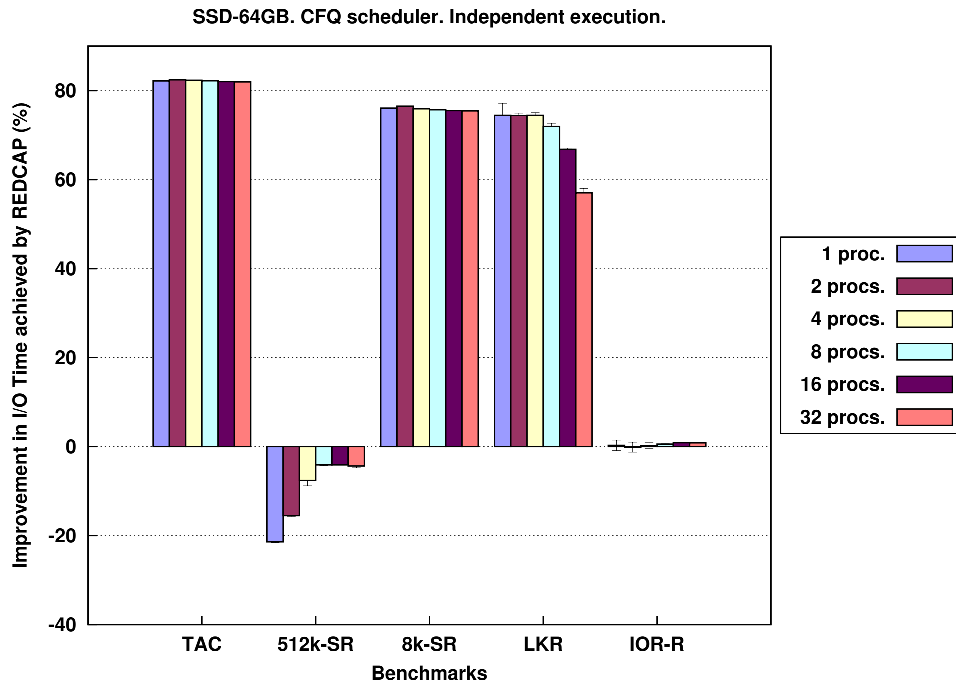


Figure 3.29: Improvement, in the I/O time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed independently, on the SSD–64 disk and with the CFQ scheduler.

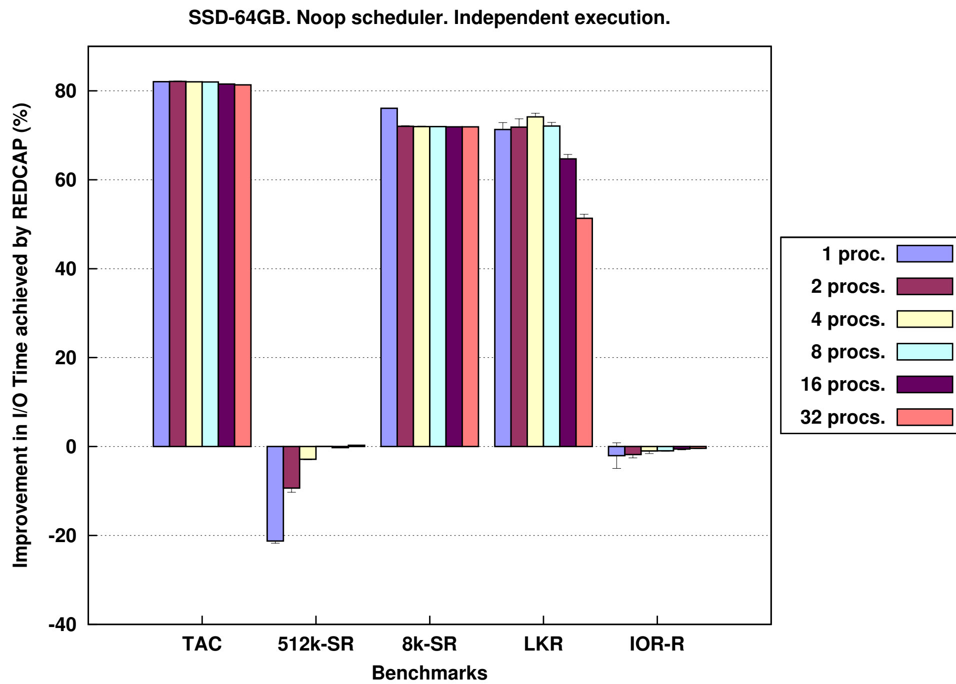


Figure 3.30: Improvement, in the I/O time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed independently, on the SSD–64 disk and with the Noop scheduler.

Table 3.6: For the SSD–160 disk, average application time measured during the execution of the *512k–SR* test, for 1, 2, 4, 8, 16, and 32 processes.

Scheduler	Kernel	Processes					
		1	**2**	**4**	**8**	**16**	**32**
CFQ	REDCAP–VD	3.11 s	6.44 s	11.75 s	22.53 s	45.08 s	91.37 s
	Original	2.37 s	4.66 s	9.15 s	18.04 s	36.03 s	72.84 s
Noop	REDCAP–VD	3.10 s	4.76 s	8.89 s	17.20 s	34.12 s	67.61 s
	Original	2.35 s	4.10 s	8.14 s	16.13 s	32.05 s	63.95 s

Table 3.7: For the SSD–160 disk, average I/O time measured during the execution of the *512k–SR* test, for 1, 2, 4, 8, 16, and 32 processes.

Scheduler	Kernel	Processes					
		1	**2**	**4**	**8**	**16**	**32**
CFQ	REDCAP–VD	2.31 s	4.42 s	8.21 s	15.84 s	31.62 s	63.44 s
	Original	1.89 s	3.81 s	7.62 s	15.18 s	30.33 s	60.65 s
Noop	REDCAP–VD	2.31 s	4.14 s	7.84 s	15.25 s	30.41 s	60.41 s
	Original	1.88 s	3.77 s	7.63 s	15.26 s	30.48 s	60.97 s

Figures 3.35 and 3.36, respectively. Improvements achieved in I/O time are also showed for the SSD–160 disk in Figures 3.33 and 3.34, and for the SSD–64 disk in 3.37 and 3.38.

With SSD devices, the REDCAP–VD kernel also presents a similar behavior to that obtained when the benchmarks are run independently, and we can claim that the virtual disk adapts very quickly to the workload changes that are caused by this test. Improvements of up to 80%, 71% and 28% are achieved for the TAC, 8k–SR and LKR benchmarks, respectively. With 512k–SR and IOR, the differences in application time between REDCAP and a normal system are due to the overhead introduced by our approach and to the small application times achieved, as we have explained above. Only minor differences are observed due to both the buffer and REDCAP caches, and the explanation is the same as that given in Section 3.5.2 for hard disk drives.

All the benchmarks at the same time

Results achieved by REDCAP for the application time, as compared to the original kernel's ones when all the benchmarks are executed at the same time, are presented in Figure 3.39. Data is grouped in the figure by disk and scheduler.

Our technique always performs better than the original kernel, although the improvement depends on the I/O scheduler used. REDCAP achieves its best improvements for the CFQ

scheduler, reducing application times by up to 88%. In this case, improvements decrease as the number of processes increases, but for 4 processes per benchmark (20 processes altogether) reductions of 63.6% and 62% are still achieved for SSD–160 and SSD–64, respectively. For Noop, improvements do not depend on the number of processes, and reach the 87%.

3.7. Related Work

There have been many studies and proposals about disk simulators, and its possible applications. Indeed, during the last decades, they have been widely used for analyzing the impact of disk trends [109], file system designs [110], buffer–cache replacement algorithms [111], and other architectural elements' designs and policies [112], on system's performance. For brevity, we only highlight a few of them that we consider representative for our work.

One of the most used and well known is DiskSim [113]. DiskSim is an efficient, accurate and highly–configurable disk system simulator that supports research into various aspects of the storage subsystem architecture. It was developed at the University of Michigan and enhanced at the Carnie Mellon University. DiskSim is composed of several modules that simulate disks, intermediate controllers, buses, device drivers, request schedulers, disk block caches, and disk array data organizations. Bast amount of studies have used this simulator, covering quite different research areas, for example, I/O scheduling [57, 58, 114], disk cache [54], or disk performance [115]. Note that, although DiskSim simulates modern disk drives in great detail and has been carefully validated against several disks, specific parameters of the disk drive are needed to perform the modeling. These parameters, which extremely depends on the disk drive, can not be obtained from the technical information provided by the disk manufacturers, and some of them are even considered trade secrets. Methods to directly extract information about the disk behavior [45, 50, 90, 91, 92] should be used. However, our disk simulator is able to simulate any disk just knowing its capacity. Moreover, DiskSim can not be used for on–line simulation, whereas our proposal can simultaneously evaluate different system mechanisms, and dynamically turn them on/off depending on the performance obtained.

Previously to DiskSim, the Pantheon simulator was designed to support rapid exploration of design choices in storage system and their components such as disks, tapes and array controllers [116]. Pantheon allows to specify the storage system in great detail, for example, disk head settle time or features of the disk cache. But, again, the difficulty to provide these values prevent us from using it. Note that these parameters have to be provided for each disk used. Moreover, Pantheon can not perform on–line simulation either.

Simulators have usually been implemented in user space as standalone applications, or integrated in a more general simulation environment. However, in some cases, they have been implemented inside the operating system's kernel. Wang *et al.* [110], for instance, implement a disk simulator inside the Solaris kernel to evaluate the theoretical performance potential of *eager writing* in the context of virtual log based file systems. By implementing it within the kernel, they profit other off-the-shelf system components, such as file systems. Their simulator, however, is specific to a hard disk model, and is not aimed at an on–line performance analysis of a system component, as ours is.

A few proposals have also developed disk simulators for SSD devices. Microsoft Research [117] has extended DiskSim to provide limited support for SSD simulation [118]. This is not a simulator for any specific SSD, but rather a simulator for an idealized SSD that

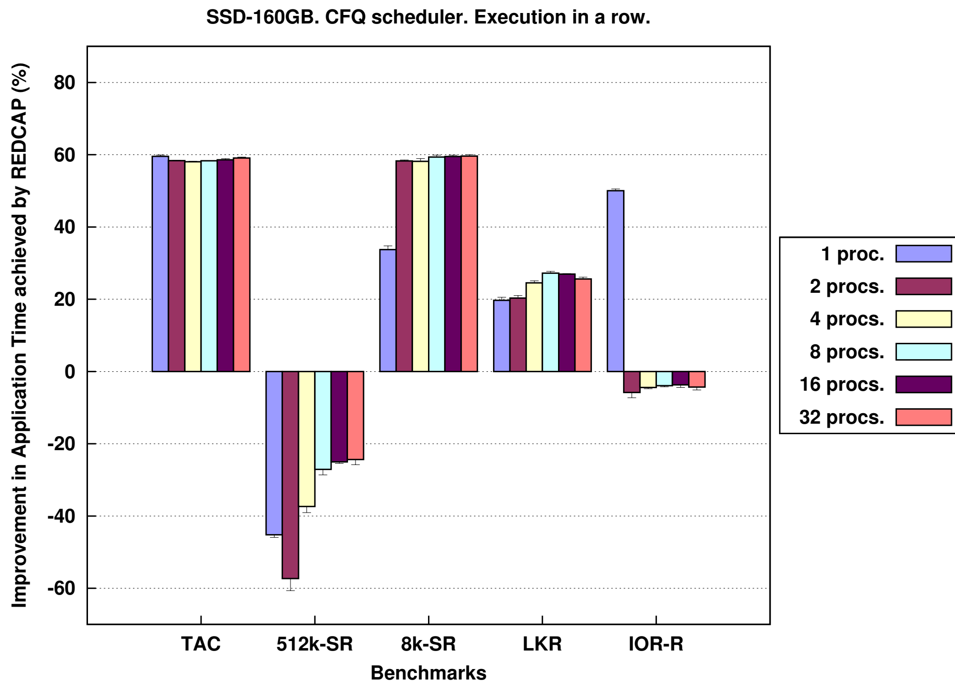


Figure 3.31: Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed in a row, on the SSD–160 disk and with the CFQ scheduler.

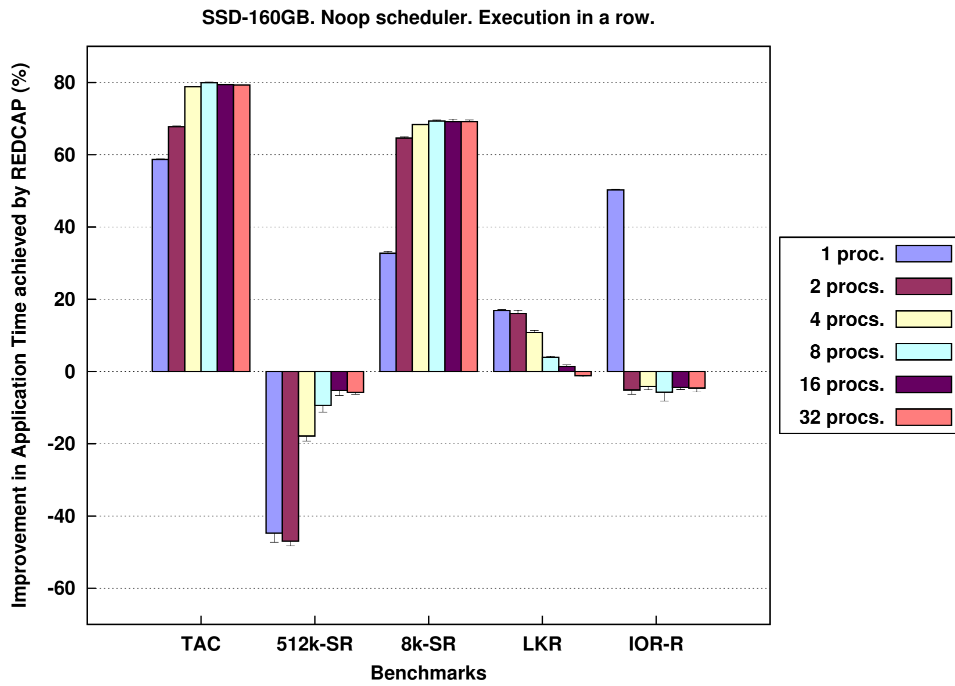


Figure 3.32: Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed in a row, on the SSD–160 disk and with the Noop scheduler.

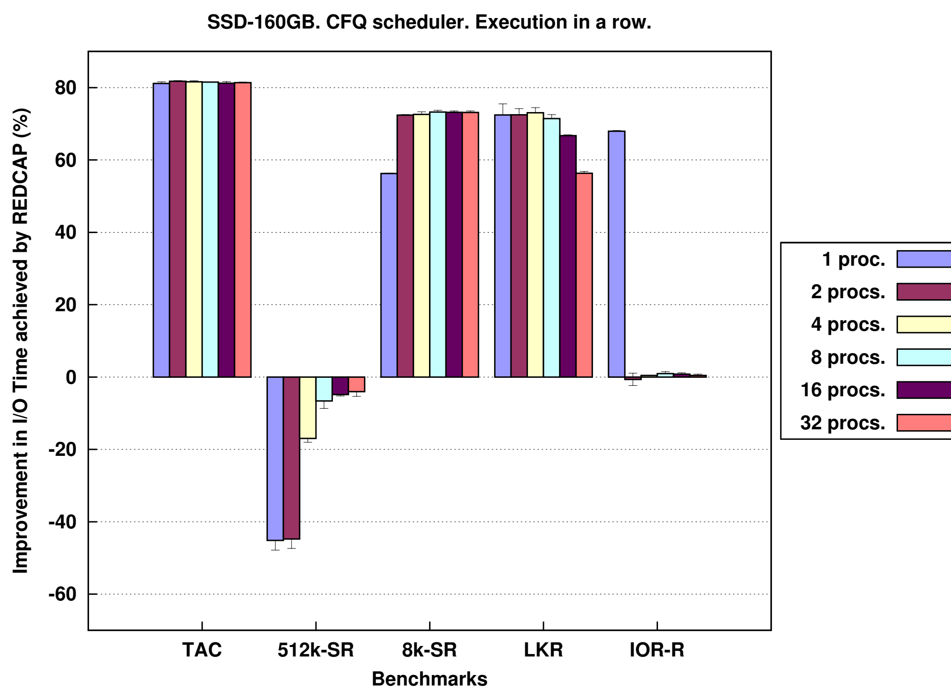


Figure 3.33: Improvement, in the I/O time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed in a row, on the SSD–160 disk and with the CFQ scheduler.

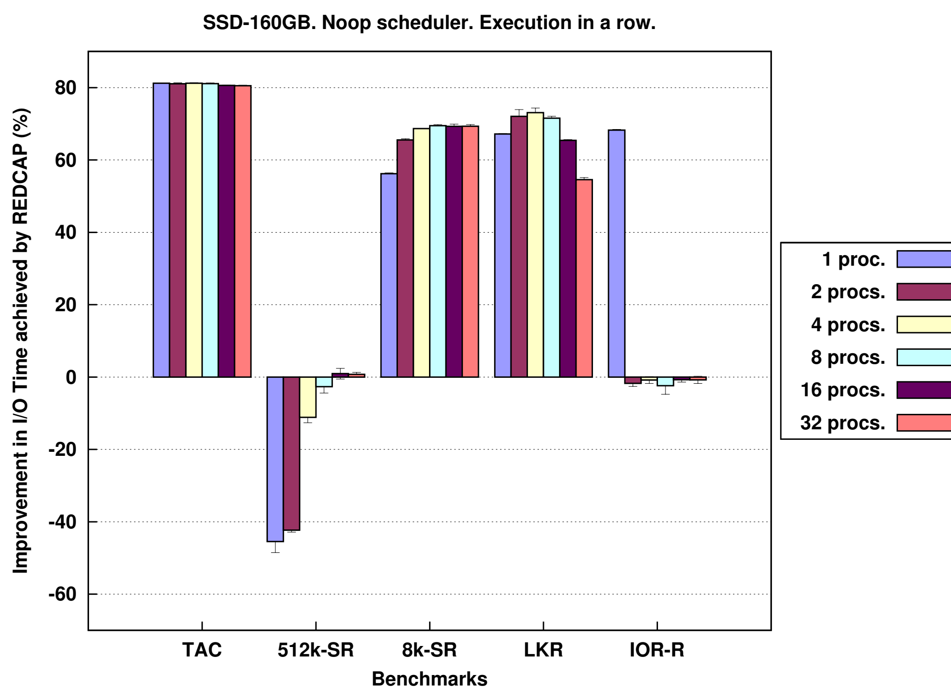


Figure 3.34: Improvement, in the I/O time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed in a row, on the SSD–160 disk and with the Noop scheduler.

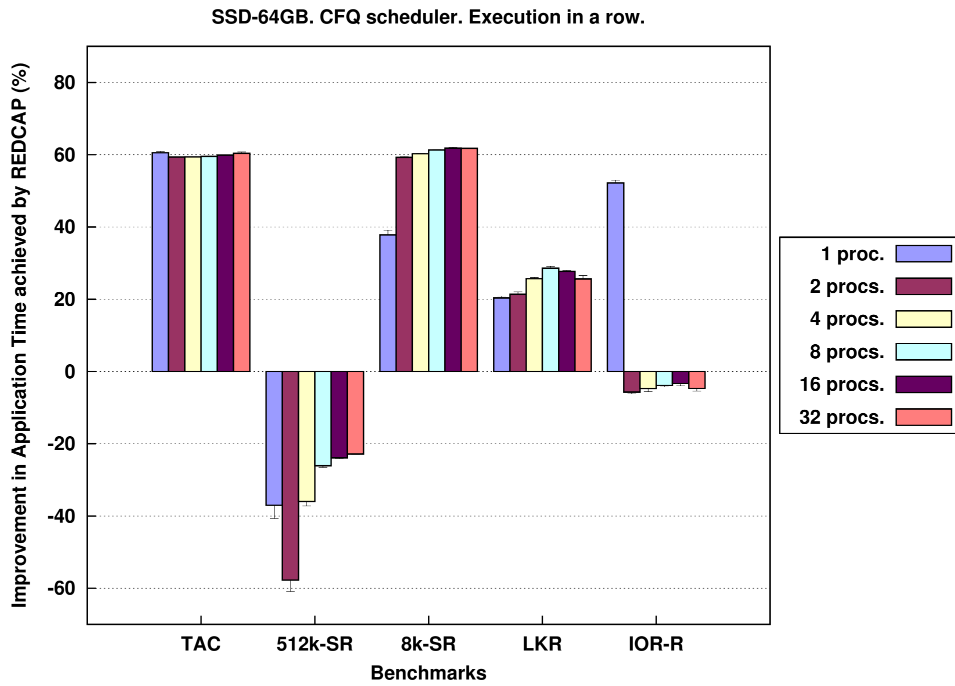


Figure 3.35: Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed in a row, on the SSD–64 disk and with the CFQ scheduler.

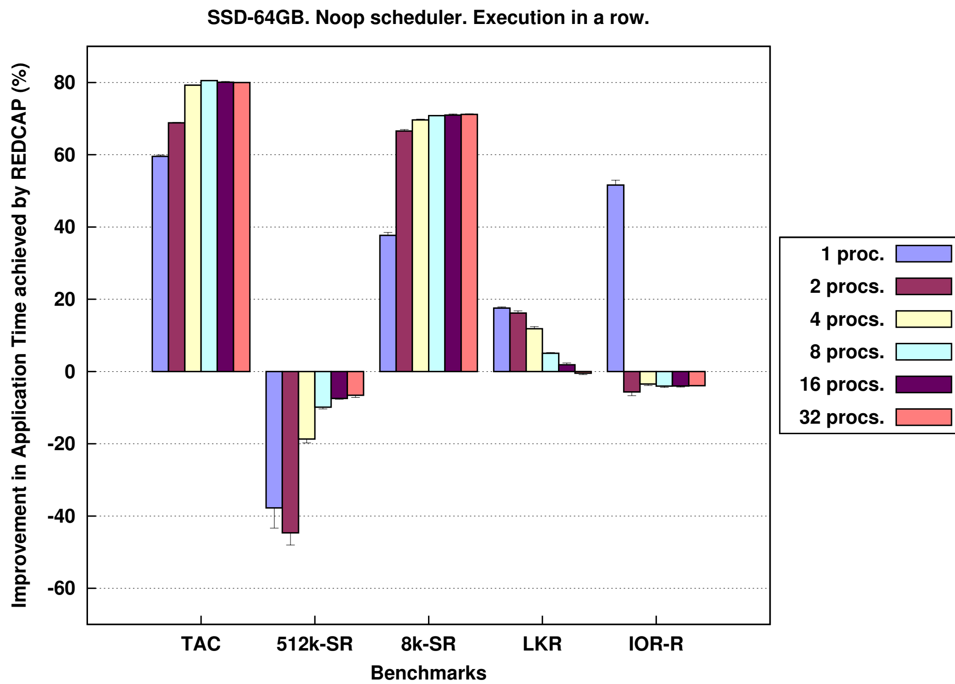


Figure 3.36: Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed in a row, on the SSD–64 disk and with the Noop scheduler.

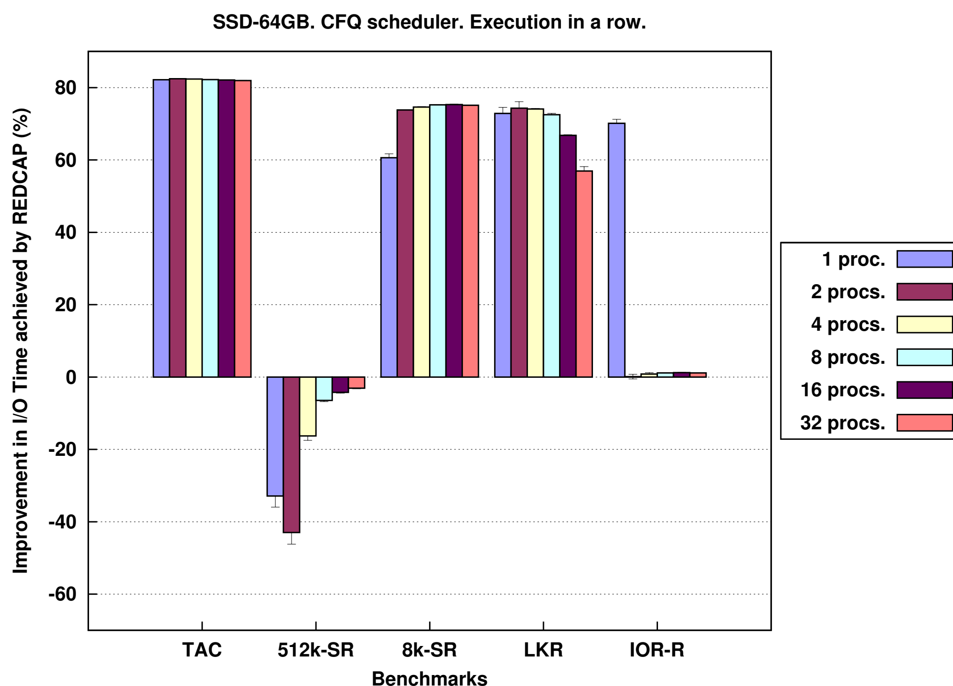


Figure 3.37: Improvement, in the I/O time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed in a row, on the SSD–64 disk and with the CFQ scheduler.

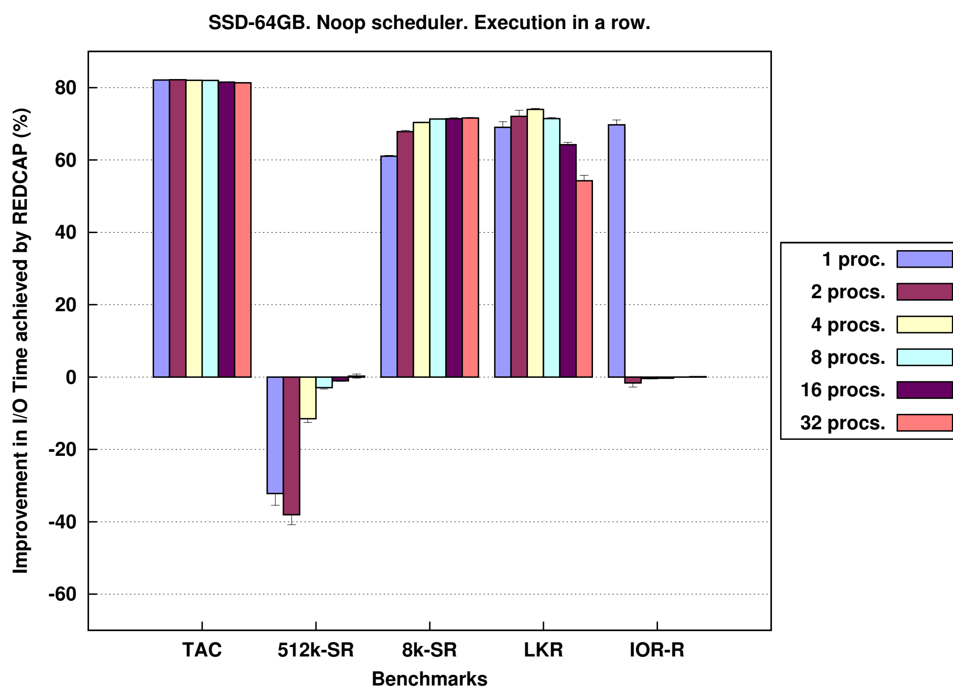


Figure 3.38: Improvement, in the I/O time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed in a row, on the SSD–64 disk and with the Noop scheduler.

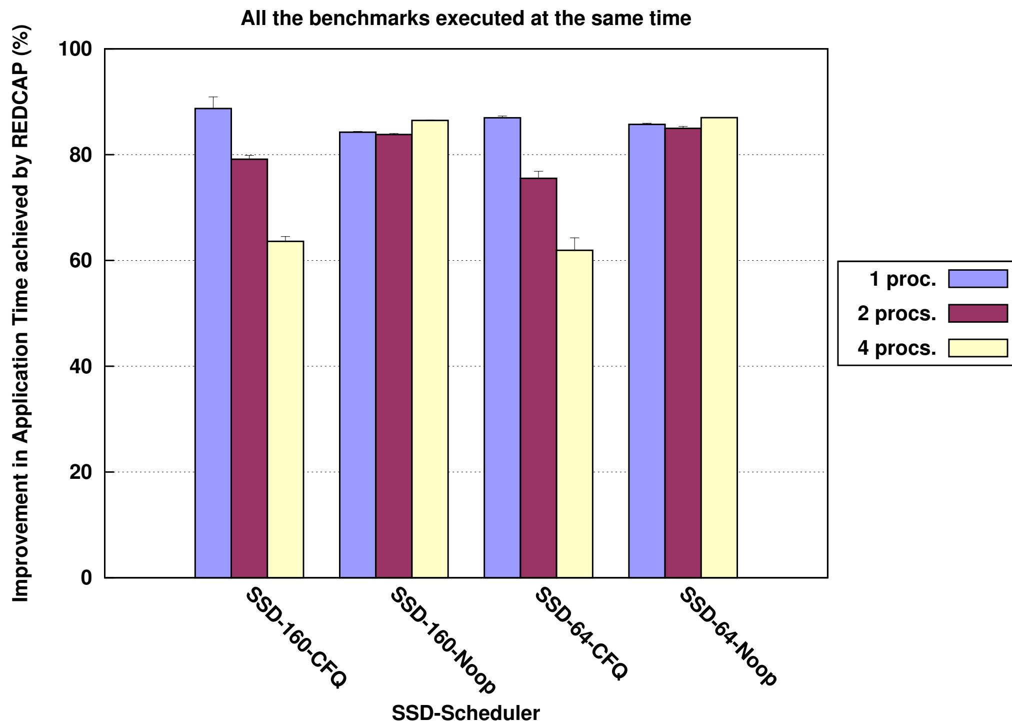


Figure 3.39: Improvement, in the application time, achieved by REDCAP over a vanilla Linux kernel when benchmarks are executed at the same time for the SSD disks.

is parametrized by the properties of NAND flash chips such as read, write, erase latency, number of chips, connectivity, and chip bandwidth.

Lee *et. al* propose an SSD simulator called *CPS–SIM* [119], which emulates operations of multiple flash memory chips and transactions of multiple buses. CPS–SIM is flexible and several parameters can be configured (number of chips, number of buses, bus bandwidth, etc.), so it can be used for designing new configurations of SSDs and predict the performance of the new models. It is a clock–driven simulator because it updates the state of all the chips and buses at each global clock pulse. Therefore, it provides accurate measurements of time and supports parallelism of chips and buses.

FlashSim is an event–driven simulator for SSDs devices proposed by Kim *et.al.* [120]. In FlashSim, each hardware or software component of SSDs is modeled, and multiple FTL (Flash Translation Layer) schemes can be simulated. FlashSim has been integrated with DiskSim, making the simulation of "hybrid" storage systems, employing combinations of SSDs and HDDs, possible.

Compared to the above simulators, our approach has the ability to simulate any SSD disk without knowing its properties. It is integrated inside the Linux kernel and can be used for performing on–line simulation and compare different I/O strategies at the same time. Furthermore, the dynamic update of our disk model allows it to adapt to the performance degradation that SSD drives suffer. On the other hand, our virtual disk does not simulates each component of the device independently, so it can not be used for the design of new configurations of SSDs.

Regarding disk modeling, a variety of models have been researched, but, for sake of simplicity, we only stand out a few. Ruemmler and Wilkes [1] introduce a disk model based on the behavior characteristic of disk drives (seek profiles, rotational latency, transfer rates, and so on), and they also build a disk simulator to test it. The approach proposed by Shriver decomposes a device into its physical components (queues, disk mechanisms and disk cache), models these individual components, and then composes the component models together to produce a model of the entire device [49]. Each component model determines its behavior from the specification of the entering workload and the lower–level device behavior. It is interesting to note that Shriver uses the Pantheon simulator [116] to validate its model, and she compares predictions of her model with those of Pantheon. Wang *et al.* propose CART [121], a different approach that explores the application of a machine learning tool to model storage devices and considers them as black boxes having no information about their internal components or algorithms. Another interesting approach is *relative fitness* which, by using CART, models the performance of storage devices [122]. Its goal is to predict performance differences between a pair of devices: the performance and resource utilization of one device, along with workload characteristics, are used for predicting the performance of another.

Table–models to simulate storage devices have also been explored in previous work [46, 123, 124, 125]. Gotlieb and MacEwen [123] propose the first and earliest table–model. They develop a queueing model of disk storage systems to measure performance as the mean response time. As queue scheduling algorithms, they use and compare FIFO and SCAN. Their multimodule disk system uses tables of approximate mean response times.

Thornock *et al.* [124] propose a stochastic method to estimate service times from collected trace data. They use tables with a row for each service time and a column for each seek distance; cells contain the probability of the associated seek distance and service time. Indeed, each column of the table is a probability distribution of service times for a particular seek. They construct a simulator that, for each request, uses the distance from the previous one to index the table, and returns the associated probability distribution. With this distribution, they generate a random number that is returned as the estimated service time. The tables are created by using disk traces containing block requests and its associated service times. Their simulation model also uses trace data containing timing information. By using the simulator, they study the impact that a disk data reorganization has on disk I/O service times. From our point of view, this proposal has three drawbacks. The first one is that they do not consider the request size as an input parameter. However, as our accuracy study shows (see Section 3.5.1), the request size is an important factor to predict I/O times, especially because it determines transfer times for small inter–request distances. The second one is that they train the tables with a specific workload, and they then use these tables to estimate service times for requests of other workloads. But, as we have seen, similar requests (in size and seek distance) can have quite different service times depending on the access pattern. Therefore, a dynamic approach, as the one implemented in our disk simulator, would be needed to adapt the stored values to the current workload. Finally, since they use trace data, they perform an off–line simulation, whereas our in–kernel virtual disk performs on–line simulation with "virtual requests" that are copied from the real requests.

Anderson [125] uses a table–model to measure the performance of disk arrays. Their approach memorizes performance values for a particular type of workload, and interpolates between these values for unseen workloads. The model returns the maximum estimated

throughput available for the input parameters. As input parameters he considers some that specify the device information (RAID level, number of disks) and others that are a summary of the request pattern or stream (request type, request size, sequentiality of the requests and average queue length). He creates separate tables for each raid level, number of disks in the raid group, and each operation type. For each of the streams parameters, he specifies the possible values for the parameter, and measures the maximum performance that can be achieved at those parameter points. He studies three different ways of performing the interpolation of nearby points withing the table, and determines that the hyperplane interpolation algorithm performs better than the closest point algorithm and the nearest neighbor averaging algorithm. He proposes to use the model to assist in the reconfiguration of disk arrays. Note that he only takes samples for a fixed amount of time, while we believe that a dynamic approach will produce better estimations.

Popovici *et al.* [46] also implement a table–based disk simulator of the underlying storage device inside the Linux kernel, called Disk Mimic, which returns the positioning time of a request. They also propose a disk scheduling algorithm, called shortest–mimicked–time–first, that select the request that is predicted by the Disk Mimic to have the shortest positioning time. To predict the positioning time, their table model only uses as input parameters the logical distance between two requests and the type of the current and previous requests. To represent ranges of missing inter–request distances, they use simple linear interpolation.

Our proposal is similar to Disk Mimic. As our virtual disk, Disk Mimic is portable across the full range of devices, uses automatic run–time simulation, and its computational overhead is small. However, there are also several important differences.

The first difference is that their table–model is used for predicting positioning time, while ours model predicts I/O time. Furthermore, they only uses as input parameters: the inter–request distance, and the type of the current and previous operations. However, we take into account the request size, but not the type of the previous request. Our accuracy study (see Section 3.5.1) shows that the size is especially important for small inter–request distances, where both the disk cache and the transfer time are the dominant factors in the I/O time. Regarding the type of the previous request, as we have explained in Section 3.2.1, the type of the last and current requests is, with a high probability, the same, so it is not worth taking it into account.

Memory overheads are also quite different. Popovici *et al.* say that given a disk of size 10 GB, the amount of memory required for their table can exceed 800 MB, but, the interpolation that they use leads to a 10–fold memory savings [46]. Therefore, given a disk of 400 GB, the one used in our experiments, the amount of memory required for their table is around 3 GB (there are more than 80 millions possible inter–request distances with interpolation). However, since in our table–model each column represents several millions of inter–request distances, except for the first columns (see Section 3.2.1), our tables only require 7 MB of memory.

Another difference is that Disk Mimic only captures the effects of simple prefetchings, but not the effect of the disk cache. However, our disk simulator implements a dynamic model which makes it possible to take into account the effect of the disk cache on the current workload to a large extent. Moreover, thanks to the dynamic model, our virtual disk is also able to forget the past history, which depends on the past workload (in many cases, quite different to the current workload). By being dynamic and forgetting the past, our model

performs a very quick adaptation to the real disk when there is a change of the workload. However, it seems that Popovici's on–line configuration takes a long time to equal the off–line configuration (which, in turn, needs much more requests than our off–line training) [46].

Finally, they propose a disk scheduling algorithm which only uses Disk Mimic to select the request that will have the shortest positioning time. However, our proposal can evaluate several mechanisms and I/O strategies in parallel, and changes from one to another depending on the obtained performance.

The idea of issuing two requests, a "real" one and a "virtual" one, at the same time is somehow similar to the I/O speculation in user space proposed by Chang and Gibson [126], or in kernel space proposed by Fraser and Chang [127]. They propose running ahead of the execution of a stalled application to anticipate page faults, or I/O operations on blocks which are not in main memory, to prefetch disk blocks and convert cache misses into hits. By using unused processing cycles, both try to discover and initiate prefetching for future data. Chang and Gibson specify an automatable procedure for modifying applications to perform this speculative execution. Fraser and Chang add a new type of process to the system, a speculative process, that is created by forking a normal process the first time it blocks on a disk request. This new process issues non–blocking prefetch whenever a normal process would have blocked on disk read.

3.8. Conclusions

In this chapter, we have presented the design and implementation of a virtual disk inside the Linux kernel that simulates the behavior of a disk drive. The virtual disk reproduces the part of the I/O subsystem: it works as a disk drive with a disk model to simulate the behavior of a real disk; it works as a block device driver having its own I/O scheduler; and it simulates the insertion process of requests to its I/O scheduler.

We have modeled the storage device with a dynamic table model that, given a request, returns the I/O time needed to serve it. We manage two tables, one for each operation type: the read table and the write table. The tables are addressable by request size and seek distance, where seek distance is calculated as the inter–request distance from the previous request.

The four Linux I/O schedulers, available in the kernel version 2.6.23, can be used in the virtual disk, although AS and CFQ have been slightly modified because they take into account information about the process issuing a request to perform the scheduling.

Since the order in which I/O requests are submitted depends on the I/O mechanisms used, and because the requests of a process have to be served in the same order as they were produced, the virtual disk controls dependencies among requests of a process, and also among requests of related processes (e.g. a child process and its parent process).

The accuracy of the disk model has been evaluated by making the real and virtual disks serve the same requests. This analysis has been made for two different disk drives: a traditional hard disk, and a new SSD disk.

With respect to the hard disk drive, the results state that, when the I/O time stored in each cell is computed by averaging the last sixty four values, our model presents a good behavior and matches the real disk in an accurate way. Differences are due to the difficulty of simulating the disk cache. Hence, when the cache is off, the virtual disk matches the real

disk in a very accurate way, with differences smaller than 0.2%. Finally, we have also showed that when our model does not take into account the request size, the virtual disk does not match the real one at all, and differences between them are rather significant.

Regarding the SSD disk, the analysis states that our model provides estimations of I/O time with a high precision. The virtual disk behaves very much like the SSD device, and differences between both disks are, on average, less than 0.3%.

Thanks to the implementation inside the kernel, the proposed virtual disk can be used for an on–line simulation of the performance obtained by different system mechanisms and algorithms, and for dynamically turning them on and off, or selecting between different configurations or policies, accordingly. Specifically, we have described the use of the virtual disk in REDCAP to decide the proper state of its cache depending on the throughput achieved. In the active state, the virtual disk simulates the behavior of a normal system (without REDCAP). In the inactive state, the virtual disk helps to simulate a REDCAP system. Since the virtual disk adapts very quickly to changes on the workload, the new activation–deactivation algorithm works better than the first version.

We have performed a set of experiments to analyze the performance of REDCAP when its algorithm uses the virtual disk. The experiments uses four different disk drives (two traditional hard disks and two SSD disks), different I/O schedulers, several workloads, and both a fresh and aged Ext3 file systems.

The results show that, by using the virtual disk, REDCAP is usually able to decide the state of its cache and to obtain the maximum possible improvements: up to 80% for workloads with some spatial locality, and the same performance as a system without REDCAP for workloads with random or large sequential read requests. These results are consistent with those obtained in previous studies (see Section 2.5 in Chapter 2). Hence, despite the fact that our first approach uses a much more simple disk model, improvements achieved in both cases are quite similar, and the new disk model only solves some specific problems in access patterns with small strides. Nonetheless, the new disk simulator, with its more precise disk model, is necessary to simulate and compare other I/O mechanisms, such as the I/O scheduler change proposed in Chapter 4.

The results also show other interesting aspects. Firstly, aged file systems do not have a negative impact on the REDCAP behavior, and, although our approach achieves a slightly lower performance than for a fresh file system, REDCAP still gets improvements of up to 78%.

Secondly, for SSD devices, REDCAP provides significant reductions of up to 88%, and its performance is quite similar to that obtained for hard disks. However, the small overhead introduced by REDCAP and the virtual disk is more noticeable with these devices due to the high throughput that they offer, which translates into small application times.

Lastly, REDCAP behaves the same for any I/O scheduler, but improvements slightly depend on the I/O scheduler used. The problem is that the scheduling policy determines I/O performance of disk drives to a large extent.

To summarize, our disk simulator has several interesting features: i) it creates a virtual disk which is able to simulate any disk, even SSDs, by using a table of I/O times; ii) thanks to the table, the system overhead produced by the simulator is negligible; iii) since it has the same interface as any other block device, it makes it possible to use any existing Linux I/O scheduler with minimal modifications; iv) it does not interfere with regular I/O requests

because virtual requests are handled out of the real I/O path; and v) it simulates the service order of the requests in a real disk by considering the possible dependencies among them.

Chapter 4

DADS: Dynamic and Automatic Disk Scheduling framework

The selection of the right I/O scheduler for a given workload can significantly improve the I/O performance of a system. However, this is not an easy task because several factors (workload, disk, file system, etc.) should be considered, and even the "best" scheduler can change at any moment, specially if the workload's characteristics change too. To address this problem, we present a Dynamic and Automatic Disk Scheduling framework (*DADS*) that compares different Linux I/O schedulers at the same time, and automatically and dynamically selects that which achieves the best I/O performance for any given workload.

The DADS implementation described here compares two I/O schedulers, although it can easily be improved to support more schedulers. The comparison is made by running two instances of a disk simulator inside the Linux kernel. One instance has the same scheduler as the analyzed real disk uses, and the other one has the scheduler with which the comparison is made. A scheduler's performance is measured as the sum of the service times of all the served requests. DADS compares the schedulers' times, and changes the scheduler on the real disk if the performance is expected to improve.

Our proposal has been analyzed by using different workloads, six different disk drives, both fresh and aged Ext3 file systems, and four I/O schedulers available in Linux. The experimental results show that DADS selects the best scheduler of the two compared at each moment, improving the I/O performance and exempting system administrators from selecting a suboptimal scheduler.

This chapter is organized as follows. We start by introducing the problem. The design of DADS is discussed in Section 4.2. The description of the modified disk simulator is given in Section 4.3. In Section 4.4, the implementation of DADS is discussed. The hardware platform, disk cache configuration, benchmarks and I/O schedulers used in the experiments are described in Section 4.5. A comparison of our results to those of a traditional system is made in Section 4.6. Section 4.7 contains a brief description of previous work related to the proposed technique. The conclusions of this chapter are provided in Section 4.8.

4.1. Motivation

The main goals of I/O schedulers are to optimise disk access time and maximize disk throughput. With these purposes, they decide the request–sequencing policy, i.e., the order of the incoming requests in the scheduler queue and when each request is dispatched to the disk drive. Without an I/O scheduler, the operating system would just dispatch each request

to disk in the same order as it had received them, and the I/O performance, and indirectly the system performance, would be normally awful.

Since hard disk drives became the dominant secondary storage device in the 1960s, many scheduling policies have been proposed to improve the I/O performance. Some of them try to minimize seek time (time spent moving the disk heads), other proposals also take rotational delay into account, and even there are algorithms that assign deadlines to requests and try not to violate them. Thus, each I/O scheduler sorts the requests to accomplish a target optimization: throughput, quality of service, fairness, etc. However, none of the scheduling algorithms is optimal in the sense that the improvement that they provide depends on several factors: workload characteristics, file systems, disk drives, tunable parameters, and so on. They even usually have a worst–case scenario which could downgrade the I/O performance.

For instance, in Linux, while the CFQ scheduler, in general, provides a good performance for hard disks, there are workloads where the AS scheduler obtains a much better result. This behavior can be observed in Figure 4.1, that depicts application time improvements obtained by the AS scheduler over the CFQ scheduler for the disks HD–250–32 and HD–320–16 (see Section 4.5.1) when the Linux Kernel Read and IOR Read benchmarks (see Section 2.4.3) are run in a vanilla Linux kernel 2.6.23. As we can observe in Figure 4.1(a), for the disk HD–250–32 and Linux Kernel Read benchmark, CFQ achieves the best performance for all the processes but 32, and differences between both schedulers are significant. However, for the same disk and IOR Read benchmark (see Figure 4.1(c)), CFQ only gets the best performance for 32 processes. For the disk HD–320–16 and Linux Kernel Read benchmark (see Figure 4.1(b)), CFQ only behaves clearly better than AS for 1, 2 and 4 processes, whereas for the IOR Read benchmark (see Figure 4.1(d)), the same is obtained for 32 processes.

Another example is the Noop scheduler: its FIFO policy usually produces the worst performance since it does not sort the requests. But for random–access devices, such as flash memory cards or SSDs, or for "intelligent" hard disks, Noop usually achieves a better performance than other policies, because these devices do not depend on mechanical movements to access data [42], or they perform their own scheduling (as "intelligent" hard disks do).

Operating systems like Linux provide several I/O schedulers and, at boot time or run time, system administrators can select one of them; even a different scheduler for each disk of the computer can be used. But choosing the I/O scheduler that achieves the best performance on each disk is not an easy task. Indeed, most of the times, system administrators do not make any selection, and the default I/O scheduler is used, when the use of a different scheduler could improve the system's throughput. Therefore, a mechanism that automatically changes from one scheduler to another, depending on the expected performance, could achieve the highest throughput. Motivated by these ideas, we present the design and implementation of a Dynamic and Automatic Disk Scheduling framework (called *DADS* to short) that is able to automatically and dynamically select the best of the Linux I/O schedulers by comparing the performance achieved by each one [37].

DADS's aim is to increase the I/O bandwidth provided to the applications by the operating system. This I/O bandwidth is determined by the disk drive and, specially, by the I/O scheduler used. The former determines the disk I/O time of a request. The latter not only establishes the order in which the requests are served by the disk (in the case of a hard drive, this order could determine the performance of the device), but also can add a waiting time in the scheduler queue that increases the service time of a request (and, hence, reduces the

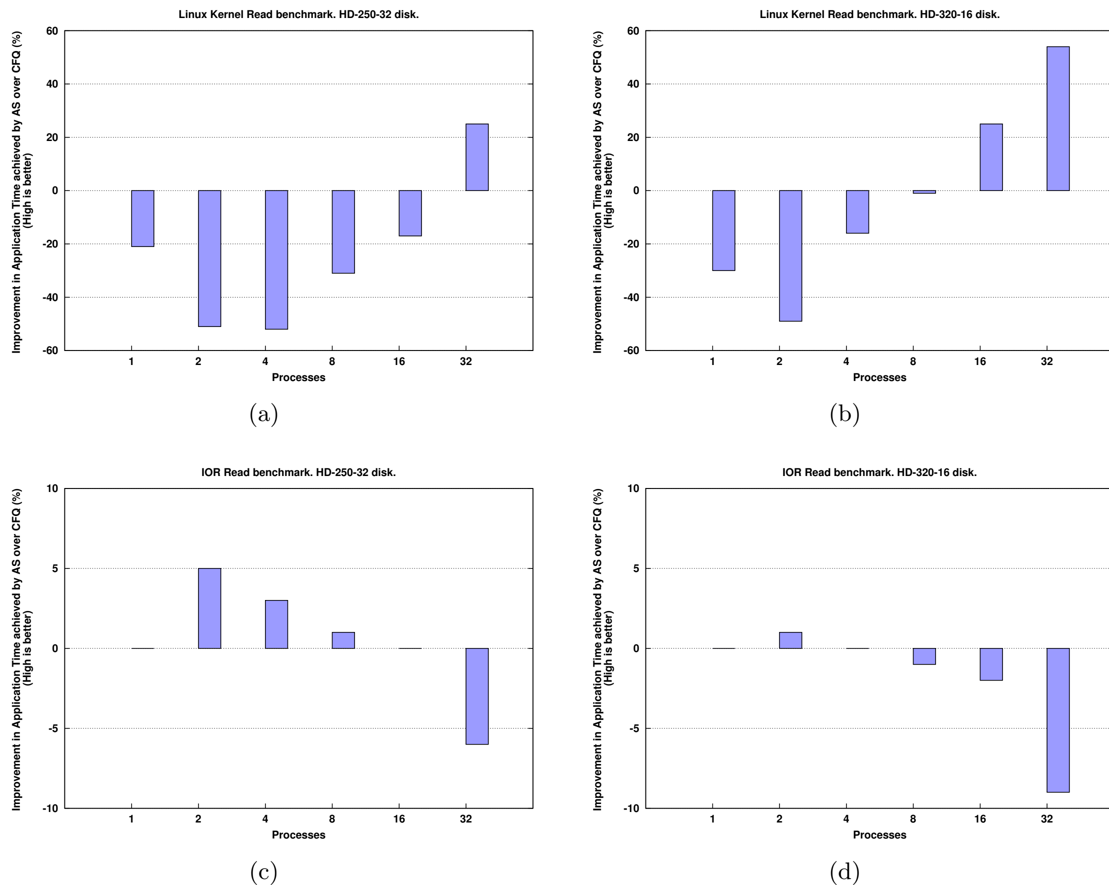


Figure 4.1: Improvements, in application time, achieved by the AS scheduler over the CFQ scheduler for the *Linux Kernel Read* ((a) and (b)) and *IOR Read* benchmarks ((c) and (d)), the disks HD–250–32 ((a) and (c)) and HD–320–16 ((b) and (d)), and 1, 2, 4, 8, 16 and 32 processes. Benchmarks have been run in a vanilla Linux kernel 2.6.23.

I/O bandwidth). Therefore, in our case, the best scheduler will be the one that provides the highest I/O bandwidth to the applications for a given disk drive.

4.2. DADS overview

As we have seen in Chapter 3, the availability of a disk simulator inside the kernel of the operating system enables the possibility of simulating an I/O strategy in parallel with the strategy enforced by the system at a given moment. One of the strategies suitable for this simulation is I/O scheduling: it is possible to compare several disk schedulers at the same time, and choose among them, according to the performance achieved by each one.

For simplicity, our first implementation of DADS only compares two I/O schedulers, and chooses, between the two compared, the one that obtains the greatest performance, however, the proposed mechanism can easily be improved to support three or more schedulers.

DADS evaluates the I/O performance of each scheduler and carries out the analysis by

using an enhanced and much–modified version of the in–kernel disk simulator introduced in Chapter 3. Our proposal runs, inside the Linux kernel, two instances of the new disk simulator (see Figure 4.2). The two instances have the same configuration, except for the I/O scheduler, and attend the same requests. The instances are:

- Virtual Disk of the Real Disk, called *VD_RD* to short, that has the same I/O scheduler as the disk that is being analyzed. Hence, it is simulating the behavior of the real disk.
- Virtual Disk of the Virtual Disk, called *VD_VD*, that has the I/O scheduler with which the comparison is made. This one is simulating the behavior of the real disk with a different scheduler.

It is important to realise that, by using two instances, and not the real disk and an instance of the disk simulator, the comparison is fairer, and allows us to know that the differences between them are due to the I/O performance of the schedulers and not to the simulation of the disk itself. Moreover, if our disk model has prediction errors, both simulations will share the same errors.

Note also that, to extend DADS to support more I/O schedulers, just more instances of the disk simulator, one for each new scheduler to compare, should be run.

In order to dynamically decide whether the real disk should change its I/O scheduler or keep the current one, the I/O performance of the schedulers has to be calculated and compared. I/O performance can be defined by means of different metrics, such as the average transfer rate achieved by the disk, the average response time per request, the maximum response time in the last X requests, and so on. In our case, I/O performance is defined as the service time provided to applications by the operating system. For each request, the disk instances calculate its service time as the elapsed time since the request is inserted into the scheduler queue until its completion. Therefore, a scheduler's performance is measured as the sum of the service time of all the requests that it serves, and our approach selects the scheduler that optimizes this total service time. Since both instances also transfer the same amount of bytes, reducing the service time implies to increase the I/O bandwidth.

4.3. Modification of the In–Kernel Virtual Disk

To make a fair comparison among different I/O schedulers, two aspects deserve a special attention. Firstly, the thinking time of the requests has to be consider to simulate their inter-arrival times. Otherwise, all the requests of a process would arrive in a row and almost at the same time, so the process's "real" I/O behavior would not be rightly simulated. Secondly, the disk cache has to be directly simulated, because the two instances of the disk simulator have different I/O schedulers, and the cache hit ratio produced by an I/O scheduler significantly determine the disk performance.

To take into account these aspects, the new disk simulator consists of two subsystems which work together simulating different parts of the I/O subsystem. The first subsystem, that we call *virtual disk* (*VD*), works as both a block device driver and a disk device. The second one, called *request arrival simulator* (*RAS*), simulates the arrival of requests to the virtual disk. This enhanced version allows us to compare the performance obtained by any I/O scheduler.

The virtual disk simulates the behavior of a real disk, and also has its own I/O scheduler with a queue where incoming requests are sorted before being dispatched. This subsystem is

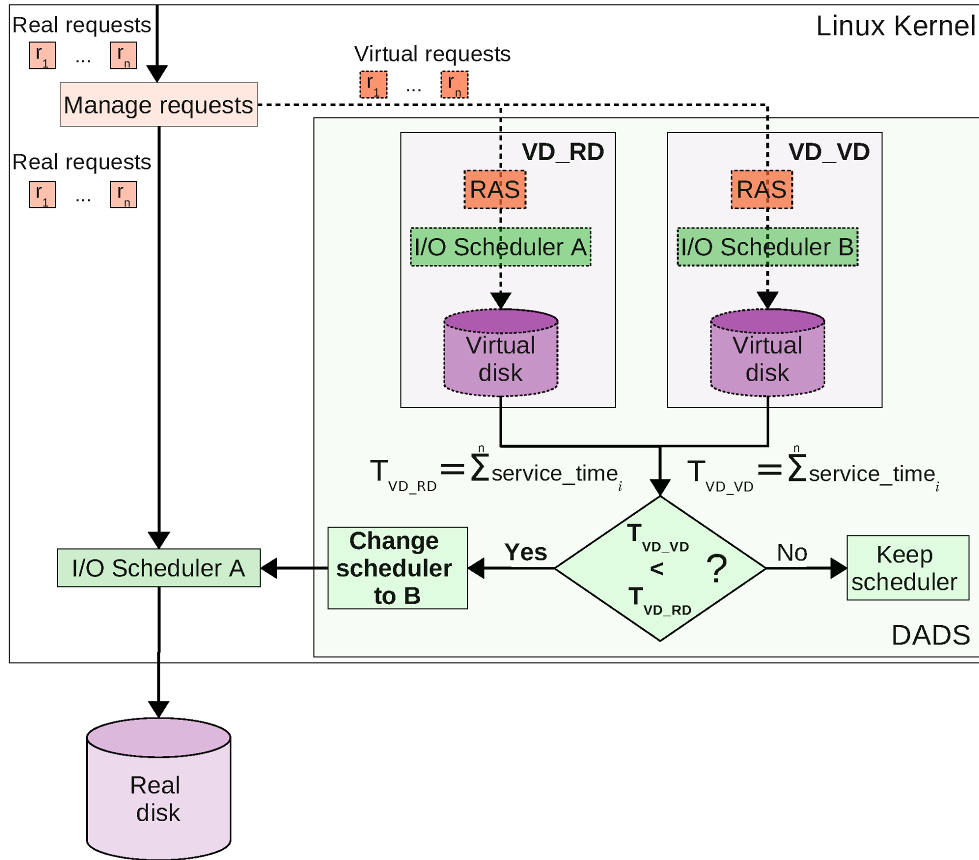


Figure 4.2: Overview of DADS.

implemented by using a kernel thread which, after an initialization phase, executes a routine that implements both a disk driver and a disk controller. The main steps of this routine are:

1. Fetch the next request from the scheduler queue.
2. Get the estimated I/O time needed to attend the request.
3. Sleep the estimated time to simulate that the disk operation is being performed.
4. After waking up, complete the request, delete it from the scheduler queue, and inform to RAS that the request has been finished.

Realize that it performs most of the tasks of the initial disk simulator, and only the management of the request arrival is done by RAS.

The virtual disk simulates the behavior of a real disk with a disk model that is a modification of the one proposed in Section 3.2.1. The new disk model is discussed in Section 4.3.1. The I/O schedulers that the virtual disk can use are the same schedulers as those used by the first implementation. Section 4.3.2 presents modifications regarding the I/O schedulers, for a full description of the schedulers, see Section 3.2.2.

Regarding the second subsystem, RAS is also a kernel thread that inserts requests into the I/O scheduler queue of the virtual disk, and simulates their arrival to the block device driver. When a new request is queued in the scheduler, if the virtual disk is plugged, RAS wakes

up the virtual disk. Another task of RAS is to control and simulate *thinking times*: times elapsed between the completion of a request and the arrival of the next request issued by the same application. This time depends on the application's behavior. Section 4.3.3 describes how the thinking time is calculated and simulated.

RAS also controls the arrival order of the requests and dependencies among them to allow the virtual disk to serve them in the "right" order. Virtual requests of a process have to be served in the same order as they were produced taking into account their arrival and completion times. This order control also has to be done among requests of related processes (e.g., a child and a parent process). Moreover, dependencies between related processes are also taken into account to compute some thinking times. The request management and control of dependencies are quite similar to those performed in our first implementation (see Section 3.2.3), and differences are discussed in Section 4.3.4.

To sum up, the main differences between the first version of the in–kernel disk simulator and the one presented here are the following:

- In the previous implementation, RAS was not implemented, so the virtual disk itself inserted requests into the scheduler queue, and also controlled dependencies between them. In the current implementation, the virtual disk only simulates the I/O process, and RAS handles the arrival of the requests.
- Thanks to RAS, the simulation of the arrival of the requests is more accurate: a request can be inserted into the scheduler queue while the virtual disk is serving a different request.
- The thinking time of the requests was not taken into account either: the requests were inserted into the scheduler queue just when their dependencies were solved.
- The disk model to calculate the I/O time of each request is also different. Now, as we will explain in the next section, a third table is used for read cache hits and a disk cache is directly simulated. In our previous version, just one table for read operations was used, and the effect of the disk cache was caught by the dynamic update of the disk model.

Figure 4.3 shows an overview of the new disk simulator with the division among RAS and the virtual disk. In the following sections, we discuss the features of our disk simulator, but focusing our attention only on the differences with the first implementation.

4.3.1. Disk model

Our initial table–based disk model has been modified by introducing a third table of I/O times for read operations that are cache hits. We have also added a disk cache and its corresponding management.

Since read and write operations take different I/O times [1, 45], we initially used two tables, one for each operation type. However, since read operations take different times depending on whether they are cache hits or misses, now we use two read tables, one for each type of read operation. Thus, the three tables managed by our model are:

- a cache–miss read table that predicts the I/O time of read requests that are cache misses;
- a cache–hit read table that predicts the I/O time of read requests that are cache hits;
- a write table that predicts the I/O time of write requests.

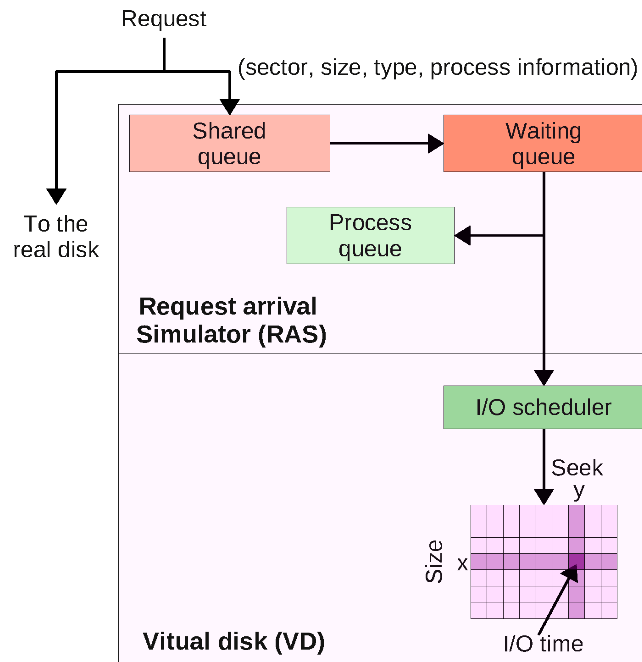


Figure 4.3: Overview of the disk simulator divided into RAS and the virtual disk.

Apart from the tables, the new disk model needs a disk cache model to determine whether a read operation produces a cache hit or miss, and then decide the corresponding read table to use.

Let us remark that if we use a single read table, as in the previous model, the estimated read times would not be as exact as we need, because a cell's value would be the average of cache hit and cache miss times, and differences between them are usually very large. Indeed, a cell would contain the average of values of a roughly bimodal distribution. For instance, for the hard drives used in our experiments (see Section 4.6), a read request of 4 kB in size takes between 150 μs and 220 μs for a cache hit, whereas it takes around 8000 μs for a cache miss. In our previous proposal, differences among cache hit and cache miss times were not important, because there was only one I/O scheduler and the effect of the disk cache was caught by dynamically updating the cells of the read table. However, for the new dynamic scheduling mechanism, the disk cache and the corresponding hit and miss times have to be directly simulated, because each instance has a different I/O scheduler, and, as we have said, the disk performance is significantly determined by the cache hit ratio produced by the I/O scheduler.

With respect to the write table, we assume that only one table is needed because write requests are usually either sporadic or bursty. In the former case, due to the write–back policy and immediate reporting normally used in disk caches [49], write requests are considered "done" as soon as they are in the cache. Therefore, I/O times are smaller. In the latter case, since the disk cache is saturated, a request usually has to wait for the previous one to reach the media surface. Hence, the net effect is like there does not exist disk cache, and I/O times are large. As a consequence, table cells will be updated with either short or large I/O times,

and the write table will adapt to both cases.

The following sections expound the main differences with the previous disk model (table structure, input parameters, disk cache, and so on).

Table structure

The three tables have the same structure, the only difference is the kind of values they store in each cell: I/O write times, cache–miss read times and cache–hit read times. In addition, they have the same structure as the tables used in our first model: rows represent request sizes, and columns represent inter–request distances. There are thirty two rows, corresponding to sizes from 1 (4 kB) to 32 blocks (128 kB). Columns represent ranges of inter–request distances. The number of columns is determined by the disk size.

It may seem obvious that, for a cache–hit read operation, the inter–request distance from a previous request could not have any influence on its I/O time. However, for two of the tested disks, we have realised that I/O times of read operation cache–hit also depend on the number of cache segments used at the same time. For instance, for a 4 kB request that is a cache hit, the I/O time is, on average, 177 μs if there is a single read stream, that is, when only one cache segment is used. However, it is, on average, 323 μs if the cache hits occur at different cache segments due to concurrent read streams. Since disk caches are divided into segments, we think that this time difference is due to the selection of a different segment to serve the current request from the cache. However, it is just a hypothesis because we have found no information about this behavior.

Input parameters

The new table–model uses one more parameter, but only for read operations: the information about whether the request is a cache hit or a miss.

So, to predict the I/O time, our new disk model uses four parameters: the type (read or write), the request size, the inter–request distance from the previous request, and, for read operations, the cache hit/miss information. Given a request, its type and the simulated disk cache determine the table to use, its size specifies the row of the table, and its inter–request distance the column. The corresponding cell gives the I/O time to attend the request.

Simulation of the disk cache

By far, the most difficult feature to model is the data–caching characteristics of a disk, because it is not easy to deduce the real behavior of a disk cache [1, 49], specially for those with a dynamic behavior. Information about features and operation of disk caches is considered a trade secret. Indeed, size is usually the only publicly available information. However, there are many properties and features that specify a disk cache [1, 31, 49], and it is quite difficult to model all of them due to this excessive secrecy.

In order to reduce the number of possibilities, we have only modeled the main aspects. A disk cache that uses both read–ahead and immediate reporting have been simulated. We have also considered that the disk cache is divided into segments of equal size, which are used for holding contiguous data. Some of the segments are used for read operations and other for write operations. The number of segments is set fixed, we have not considered dynamic

division of the cache. Least Recently Used (LRU) is used as replacement algorithm because it is the most popular in disk caches [31, 49].

Our simulated disk cache only performs read–ahead on cache misses. We have also considered an adaptive read–ahead policy that uses different read–ahead sizes. There are two basic sizes: one for sequential accesses (the maximum read–ahead size), and another one for random accesses (the minimum read–ahead size). The policy to determine which size should be used is the following:

- An access is considered sequential when the inter–request distance from the previous request is less than 64 kB.
- The first access to a cache segment always produces a miss.
    - If this access is sequential with respect to the previous disk access, the maximum read–ahead size is used for reading data.
    - Otherwise, the read–ahead size for random accesses is used.
- If the prefetched blocks produce cache hits, the read–ahead size is doubled to address any subsequent miss in the same segment, up to the maximum allowed size.
- Cache misses do not change the read–ahead size of the affected segment.

Since modern drives hide their physical geometry to the operating system, the exact layout of the blocks on disk is unknown, and it is impossible, for instance, to simulate that the prefetching is performed for track–aligned disk blocks. Therefore, we have considered that the first time a segment cache is used, the requested blocks, which produce the miss, are in the middle of the prefetched blocks. Other subsequent prefetched blocks will be behind or in front of the requested blocks, depending on the access direction.

Moreover, for the sake of simplicity, our cache disk model treats a partial cache miss as a complete cache miss, and it does not handle partial cache hits, which are treated as complete cache hits. A partial miss occurs when only a subset of the requested blocks are in the disk cache at the time the read request arrives, and the other blocks have to be read from disk [49]. A partial cache hit occurs when a read request arrives while the cache is performing the read–ahead that will serve the request [49].

Given a disk, the number of segments of its cache and the read–ahead sizes are calculated by using a capturing program (described in Section 4.3.6), whereas its cache size is obtained from the manufacturer's specification.

We are aware that our disk cache model does not fully simulate a disk cache, and it is just an approximation. However, our intent is not to develop the best possible disk cache model for a given disk, but to develop one "alike enough" that allows us to study the performance of the system with different I/O schedulers. Since the disk performance is greatly determined by the cache hit ratio produced by an I/O scheduler, we will consider that the cache of our virtual disk is "alike enough" if it achieves a hit ratio similar to that obtained by the read disk. The experimental results (see Section 4.6) will show that our cache model meets this requirement, and that it is suitable to make the comparison of two I/O schedulers.

With respect to SSD devices, although some of them also have a disk cache [13, 128], to the best of our knowledge, this cache is mainly used for internal operations. We have not found any information about if they perform any kind of prefetching or read–ahead. Therefore, in our disk model, we have established that SSD devices do not have a disk cache, and we only use the cache–miss read table. The disk model still manages three tables, though every read request is considered a cache miss, and the corresponding table is always selected.

Operation of the disk model

To sum up, the operation of our disk model is the following:

- For each read operation:
  - if the requested blocks are found in the simulated disk cache, a hit occurs:
    - the cache–hit read table is selected to obtain the I/O time,
    - the simulated disk cache is updated by applying the LRU algorithm.
  - if the requested blocks are not found, a miss occurs:
    - the cache–miss read table is selected,
    - the simulated disk cache is updated by allocating new data depending on both the read–ahead size and the sequentiality of the accesses.
- For each write operation, the simulated disk cache is updated, and the write table is used for obtaining the estimated I/O time.
- Once a table has been selected, the size of the request selects the row, and its inter–request distance from the previous request the column. The value of the cell is the I/O time needed to serve the request.

Dynamic behavior

As in the original model, tables are dynamically updated through the I/O times provided by the real disk, and the cell values are adapted to the workload characteristics. Again, the value of each cell is the average of the last sixty four corresponding samples. The number of sixty four has been chosen after performing an analysis of the sensitivity of the disk model to the number of averaged values per cell (see Section 3.5.1).

The main difference is how the read tables are updated. If a read operation is a cache hit, its I/O time updates the cache–hit read table; if it is a miss, the cache–miss read table is updated. For hard drives used in our experiments (see Section 4.5.1), we have considered that a cache miss is a read operation that takes more than 1000 μs; otherwise, it is a cache hit. Therefore, the cache–miss read table stores I/O times larger than 1000 μs, and the cache–hit read table values less than or equal to 1000 μs. For SSD drives, as we only use the cache–miss read table, just this read table is updated.

4.3.2. I/O schedulers for the virtual disk

A virtual disk, as storage device, has an I/O scheduler to manage its request queue and to decide the dispatching order. This scheduler can be changed on the fly without rebooting the system.

As we have seen in Section 3.2.2, our virtual disk can utilize any of the four Linux I/O schedulers, although AS and CFQ can not directly be deployed, and the virtual disk has to use their adaptations, AS–VD and CFQ–VD, respectively. Remember that the problem is that both schedulers take into account the process that submits each request to sort the queue. Since virtual requests are submitted by RAS and all the requests belong to its kernel thread, we need to modify these schedulers to get the process information in a different way. Section 3.2.2 details the problem with both schedulers, and the required modifications.

The other two schedulers, Noop and Deadline, do no need any modification to be used with the virtual disk, because they do not take into account any process information to perform the scheduling.

4.3.3. Thinking Time

Each application usually spends, before submitting its next I/O request, a thinking time. This thinking time is measured as the time elapsed since one request is completed until the next one is inserted into the scheduler queue. Usually, it is the time an application takes to process a completed request and to issue the next one. This time mostly depends on the application itself, although it includes not only the time that the application needs to process requests, but also the time needed by the file system layer to transform an application request into an I/O request for the block device.

In order to make a right simulation, thinking time of requests has to be considered because some Linux I/O schedulers, such as CFQ and AS, take this time into account to perform its scheduling. Moreover, if thinking time was not simulated, all the requests of a process would arrive at the same time to the scheduler of a virtual disk without simulating the "right" arrival of the requests, although, due to the control of dependencies, they all would not be queued at the same time.

The thinking time also has to be calculated among requests of related processes. For instance, the thinking time of the first request issued by a child process is the time elapsed since the completion of the last request submitted before its creation by its parent process. In the same way, the completion time of the last request of a child process is used for calculating the thinking time of the next request of its parent process.

Each request from an application has a non–zero thinking time. Nonetheless, due to the prefetching performed by the operating system, some can have a zero thinking time, which means that they were submitted before the previous one was completed.

In order to compute the thinking time of a request, the system records its arrival and completion times, and the following computation is done:

- For the first request of a new process, the thinking time is calculated by subtracting the completion time of the last request issued by its parent process before child creation from its arrival time. If the parent process is not performing I/O, the thinking time is set to 0.
- For other requests of the process, the thinking time is calculated by subtracting the completion time of the previous request of the same process from its arrival time.
- If a parent process, that is performing I/O, waits for the completion of its child process, the thinking time of the new request of the parent is calculated by subtracting the completion time of the last request issued by the child process from its arrival time.

Obviously, in all the cases, the computation is performed if, and only if, the previous request has finished before the current request has been submitted.

Once the dependencies of a request have been solved (see next section), the thinking time is counted. Then, RAS sleeps the thinking time of the request to simulate the disk operation is being submitted. Finally, the request is inserted into the scheduler queue when its thinking time has elapsed.

4.3.4. Request management

The virtual disk should serve requests in the same order as the application has produced them, and it should also control whether they are synchronous or not. Therefore, again, dependencies between requests have to be controlled.

The first version of the virtual disk manages three auxiliary queues and a control heuristic to maintain the arrival order and dependencies (see Section 3.2.3). In the current version, RAS is responsible for managing these queues and applying the control heuristic (see Figure 4.3). The queues have the same purposes: the shared queue communicates with the operating system; the waiting queue controls dependencies and maintains the arrival order of requests; and the process queue controls processes that have pending requests in the I/O scheduler. The control heuristic is the same defined in our first implementation.

4.3.5. Training the tables

As in the previous proposal, the three tables of the disk model can be initialized on–line or off–line. For the off–line initialization, the cache–miss read and write tables are initialized with the training program implemented in our first disk simulator. We have also implemented a new program for the cache–hit read table. This new training program produces several access patterns that take advantage of the disk cache, like, for instance, a sequential pattern, or a "strided" pattern with small strides.

The three training programs have to be executed only once for every disk model. The total training process is usually "fast". In our system, it took 100 minutes for a 400 GB disk to built the three tables, and only 2 minutes for a 64 GB SSD disk. Note that the dynamic update of the tables will allow them to catch the increase in service time that SSD drives suffer over time due to their use.

For each read operation, depending on whether it is a cache hit or a miss, its I/O time is stored in one of the two read tables. As we have mentioned, the cache–miss read table stores I/O times higher than 1000 μs, and the cache–hit read table values less than or equal to 1000 μs.

With an on–line configuration, there is no training overhead; cells of the three tables are just zeroed, and then dynamically updated as disk requests are served. For a not–yet–updated cell, the model will return the average of the corresponding column as I/O time, if it is not zero; otherwise, it will return the average of the nearest column with non–zero cells.

4.3.6. Calculating the parameters of the simulated disk cache

Our disk model needs four parameters to simulate a disk cache: the cache size, the number of segments, and the two read–ahead sizes. The cache size is always specified by the manufacturers; however, the number of segments and the read–ahead sizes are not. For this reason, using the information given by Worthington *et al.* [45], and also by Schindler and Ganger [50], about how to obtain a disk cache's information, we have implemented a capturing program to get these three parameters.

To calculate the number of segments (N) of a disk's cache, we initially assume that $N = 32$, and then issue N reads of 4 kB at different logical block addresses LBA_k, where $1 \leq k \leq N$ and the distance between LBA_k and LBA_{k+1} is 400 MB. The next step is to check the

cache content by issuing N reads of 4 kB at $LBA_k + 4$ kB. These reads allow us to know the number of segments. If any of them takes longer than a quarter of the disk's rotational time, a cache miss has occurred, and N has to be decreased. Otherwise, all the operations are cache hits, and N has to be incremented. This procedure is repeated until we found a value of N, where N segments produce N cache hits, but $N + 1$ segments produce, at least, a cache miss.

In order to calculate the read–ahead size for sequential accesses (S_{RAs}), we choose 128 kB as the initial value. Then, we issue three contiguous reads of 4 kB each, starting at a random LBA. Finally, we issue a new read of 4 kB at $LBA + S_{RAs}$. This last request will help us to determine the sequential read–ahead size. If the request is a cache miss, S_{RAs} has to be decreased; otherwise, it has to be incremented. The procedure is repeated with the new value of S_{RAs}.

Finally, the read–ahead size for non-sequential accesses (S_{RAr}) is obtained in a similar way, but using as many reads as number of segments calculated (N). We set the initial value to 128 kB, and issue N reads of 4 kB at different logical blocks LBA_k, where $1 \leq k \leq N$ and the distance between LBA_k and LBA_{k+1} is 400 MB. Then, we issue N reads of 4 kB at $LBA_k + S_{RAr}$. If all the reads are cache misses, S_{RAr} has to be decreased; otherwise, S_{RAr} has to be incremented. The procedure is repeated with the new value.

Due to the complex behavior of disk caches in modern hard drives, we run several tests for the same number of segments or the same read–ahead size. If all the tests agree, the value has to be increased or decreased accordingly. For example, when determining the number of segments, if all the tests produce, at least, a cache miss, N has to be decreased. However, if the tests do not agree, we assume that the value of the corresponding parameter has to be increased because, at least, a test confirmed the hypothesis. Note that the disk cache is also "re–initialized" after every test by rebooting the computer.

The program has been written in C, and, in order to minimize cache effects, the `O_DIRECT` flag has been used for opening the device file.

4.3.7. Operation of the disk simulator

Let us close this section by summarizing the operation of the disk simulator explained above, and how the system takes part in this simulation process.

To make this chapter self–contained, we include here the full description of the simulation process, although, some of the details given here were already presented in Section 3.2.7.

Operating system

In the current version, the operating system performs the same tasks as in the premier one, but with two exceptions: 1) it works with RAS, and 2) for each request, it records the arrival and completion times, which are needed to compute the thinking time.

- For each request submitted to the real disk, the system copies its main parameters to the shared queue of the disk simulator. The parameters given to RAS are:
  - LBA sector number;
  - request size;
  - operation type;

  - priority of the request;
  - arrival time;
  - PID of the process that has issued the request;
  - PID of the parent process;
  - creation time of the process;
  - creation time of the parent process;
  - for read operations, it also records whether the operation is or not a read–ahead request.

  The system also checks whether RAS is waiting for new requests. If this is the case, it wakes up RAS. Then, the "real" request is queued in the I/O scheduler of the real disk, without any modification.

- For each served request, the system copies its completion time to the shared queue, and updates the corresponding table with the I/O time of the request. Now, the system neither computes nor stores the service time of the request because the performance comparison is done between the two instances of the disk simulator.

Disk simulator

In the simulation process, each subsystem of the disk simulator executes its own routine that are described individually.

RAS. The kernel thread of RAS executes the following tasks:

- For each new request copied to the shared queue:
  1. Create a virtual request by using the parameters:
     - LBA sector number;
     - request size;
     - operation type;
     - priority of the request.
  2. Create the `io_context` structure with the process information:
     - PID of the process that has issued the request;
     - PID of its parent process;
     - creation time of the process;
     - creation time of the parent process;

     and associate `io_context` with the request. Note that, if the request is the first one issued by the process, the structure is created; otherwise, RAS looks for it in the shared or process queue.
  3. Establish the dependencies of the new request by using its process information and, for read operations, its read–ahead information.
  4. Calculate its thinking time by using its arrival time and the completion time of the previous request of the same process.
  5. Insert the new virtual request into the waiting queue.
- For each served request copied to the shared queue: store its completion time.
- For each virtual request in the waiting queue that has its dependencies solved:

1. Wait its thinking time, if any.
2. Insert it into the scheduler and process queues.
3. Delete it from the waiting queue.
4. Unplug the virtual disk if it is plugged waiting for a new request.

Realize that these steps are made only if the scheduler queue is not marked as congested.

- For each served virtual request:
  1. Delete it from the process queue.
  2. Solve the possible dependencies that other requests can have with the current one.

RAS continuously runs these steps until there are no more pending requests to insert into the scheduler queue. Then, it is put to sleep, waiting for new I/O operations.

Virtual disk. The kernel thread of the virtual disk continuously performs the following actions:

1. Fetch the next request from the scheduler queue.
2. Get the estimated I/O time needed to attend the request from the table–based model of the real disk.
3. Sleep the estimated time to simulate that the disk operation is being performed.
4. Finally, after waking up, finish the request by doing the following tasks:
   a) Complete the request.
   b) Compute and store the service time of the request for a later comparison.
   c) Delete it from the scheduler queue.
   d) Inform to RAS that the request has been finished.

The virtual disk continuously runs the above steps until there are no more pending requests to serve. Then, as a regular disk, it is plugged until new requests arrive.

4.4. Implementation of DADS

Once we know how the new disk simulator works, we are ready to discuss the implementation of the dynamic I/O scheduling.

As explained in Section 4.2, DADS selects one between two Linux I/O schedulers by using two instances of the disk simulator, each one with a different scheduler. The two instances are *VD_RD*, that has the same I/O scheduler as the analyzed real disk, and *VD_VD*, that has the I/O scheduler with which the comparison is made. The selection is done by comparing the total service time provided by each simulation.

Both instances use the same disk model, i.e., the same tables of I/O times and the same configuration of the simulated disk cache. They also serve the same requests. Obviously, the order in which the requests are served is different since each I/O scheduler establishes its own dispatching order.

In this section, we first discuss the simulation process, and then describe the scheduler change, and we finish by analyzing the performance control.

4.4.1. Simulation process

The simulation process has been divided into several phases because, otherwise, the two I/O schedulers compared can tend to be alike. The problem is that the arrival order of the requests to the disk simulator depends on the order in which the I/O scheduler of the real disk attends the requests, since it frees an application from waiting on data whereas other applications are still blocked on their own I/O operations. Therefore, a mimicry problem arises and *VD_VD* tends to behave quite similar to *VD_RD*.

This mimicry problem is better understood with an example. Let us assume that the comparison is done among CFQ and Noop. The former gives to each process exclusive access to disk for a period of time by serving a few requests of the same process in a consecutive way. The latter, on the other hand, imposes a FIFO order, and no sorting is done. Let us establish that CFQ consecutively serves at least four requests of a process (it could serve even much more). Let us also assume that there are 8 processes (P_i, with $i = 1, \ldots, 8$) issuing requests to the real disk, that each process issues four synchronous requests ($R_{i.j}$, with $i = 1, \ldots, 8$ indicating the number of process, and $j = 1, \ldots, 4$ the corresponding request), that all the processes start at the same time, and that they have quite similar thinking times.

The service order of the requests, that depends on the I/O scheduler of the real disk, will be:

- If the real disk and *VD_RD* have the Noop scheduler, and *VD_VD* has CFQ–VD, the service order of the formers will be:

  $R_{1.1}$, $R_{2.1}$, $R_{3.1}$, $R_{4.1}$, $R_{5.1}$, $R_{6.1}$, $R_{7.1}$, $R_{8.1}$, $R_{1.2}$, $R_{2.2}$, $R_{3.2}$, $R_{4.2}$, $R_{5.2}$, $R_{6.2}$, $R_{7.2}$, $R_{8.2}$, $R_{1.3}$, $R_{2.3}$, $R_{3.3}$, $R_{4.3}$, $R_{5.3}$, $R_{6.3}$, $R_{7.3}$, $R_{8.3}$, $R_{1.4}$, $R_{2.4}$, $R_{3.4}$, $R_{4.4}$, $R_{5.4}$, $R_{6.4}$, $R_{7.4}$, and $R_{8.4}$.

  And the service order in *VD_VD* will be:

  $R_{1.1}$, $R_{1.2}$, $R_{2.1}$, $R_{2.2}$, $R_{3.1}$, $R_{3.2}$, $R_{4.1}$, $R_{4.2}$, $R_{5.1}$, $R_{5.2}$, $R_{6.1}$, $R_{6.2}$, $R_{7.1}$, $R_{7.2}$, $R_{8.1}$, $R_{8.2}$, $R_{1.3}$, $R_{1.4}$, $R_{2.3}$, $R_{2.4}$, $R_{3.3}$, $R_{3.4}$, $R_{4.3}$, $R_{4.4}$, $R_{5.3}$, $R_{5.4}$, $R_{6.3}$, $R_{6.4}$, $R_{7.3}$, $R_{7.4}$, $R_{8.3}$, and $R_{8.4}$.

  Due to the FIFO order imposed by the Noop scheduler, in the better case, *VD_VD* consecutively serves only two requests of the same process, while, in a "normal situation", it could attend consecutively the four requests. When the real disk serves $R_{1.1}$, the process P_1 issues $R_{1.2}$, but there are already seven requests (of the other seven processes) waiting in the scheduler queue. Therefore, the real disk does not serve $R_{1.2}$, and P_1 can not issue its third request.
- If the real disk has the CFQ scheduler, *VD_RD* has CFQ–VD, and *VD_VD* has Noop, the service order of the first two schedulers will be:

  $R_{1.1}$, $R_{1.2}$, $R_{1.3}$, $R_{1.4}$, $R_{2.1}$, $R_{2.2}$, $R_{2.3}$, $R_{2.4}$, $R_{3.1}$, $R_{3.2}$, $R_{3.3}$, $R_{3.4}$, $R_{4.1}$, $R_{4.2}$, $R_{4.3}$, $R_{4.4}$, $R_{5.1}$, $R_{5.2}$, $R_{5.3}$, $R_{5.4}$, $R_{6.1}$, $R_{6.2}$, $R_{6.3}$, $R_{6.4}$, $R_{7.1}$, $R_{7.2}$, $R_{7.3}$, $R_{7.4}$, $R_{8.1}$, $R_{8.2}$, $R_{8.3}$, and $R_{8.4}$.

  And the order in *VD_VD*:

  $R_{1.1}$, $R_{2.1}$, $R_{3.1}$, $R_{4.1}$, $R_{5.1}$, $R_{6.1}$, $R_{7.1}$, $R_{8.1}$ $R_{1.2}$, $R_{1.3}$, $R_{1.4}$, $R_{2.2}$, $R_{2.3}$, $R_{2.4}$, $R_{3.2}$, $R_{3.3}$,

$R_{3.4}$, $R_{4.2}$, $R_{4.3}$, $R_{4.4}$, $R_{5.2}$, $R_{5.3}$, $R_{5.4}$, $R_{6.2}$, $R_{6.3}$, $R_{6.4}$, $R_{7.2}$, $R_{7.3}$, $R_{7.4}$, $R_{8.2}$, $R_{8.3}$, and $R_{8.4}$.

In this case, CFQ establishes the FIFO order to the Noop scheduler. *VD_VD* consecutively serves three requests of each process, when it should serve a request of each process. Therefore, the final order established by both schedulers is quite similar.

In order to avoid this mimicry problem, the simulation process is made in three phases:

1. The initial phase collects requests from the real disk. During this phase no simulation is done, and the following tasks are performed:
   - The system copies the submitted requests to the shared queue of RAS.
   - RAS creates the corresponding virtual requests, establishes their dependencies and calculates their thinking time. No request is inserted into the scheduler queue.
   - The virtual disk is just blocked.

   The duration of this phase can be configured by the system administrator using the */proc* virtual file system.
2. After the initial interval, the second phase runs the simulation itself:
   - The system does not copy any new requests to the shared queue.
   - RAS queues requests in the I/O scheduler of the virtual disk, by controlling their dependencies and thinking times.
   - The virtual disk serves requests by simulating the real disk, and, for each request, calculates the service time.

   This phase finishes when the virtual disk has served all the collected requests.
3. The last phase controls the performance and decides whether a scheduler change is needed or not:
   - The system does not copy any new requests, and the virtual disk is just blocked (no simulation is done).
   - The total service time of each scheduler is calculated as the sum of the service times of all the requests it has served.
   - DADS compares these total service times and changes the I/O scheduler, if the I/O performance is expected to improve with the new scheduler.

   Once this phase is finished, the process starts over by collecting new requests.

Note that the duration of the whole process depends on the first two phases.

When the two instances are running, there are a couple of interesting points. The first one is that, to not introduce too much overhead, both instances use the same shared queue, so the system makes just one copy of each request. The second point is that the comparison is done when both simulations have finished the second phase, i.e., both virtual disks have served all the requests.

Let us emphasize that the differences between this simulation process and that explained in Section 4.3.7 :

- Now, the system gives requests to RAS only during the first phase.
- The operation of RAS has been divided into two phases:
  - Firstly, it collects requests (first phase).
  - Secondly, it inserts requests to the scheduler queue (second phase).
- The virtual disk runs its simulation during the second phase.

4.4.2. Scheduler change

When a scheduler switch is decided, the schedulers of the real disk and *VD_VD* are exchanged. The change is also done in the *VD_RD*[1]. In the system, the following steps are carried out for the real disk:

- The request queue of the scheduler is drained by serving all the pending requests.
- The new scheduler queue and its internal structures are created and initialized.
- The scheduler is changed.
- The old scheduler and its data structures are free.

The process to change the schedulers of the virtual disks is similar. The difference is that, when the queue of the old scheduler is drained, pending requests, if any, are dispatched with an I/O time of 0 seconds.

4.4.3. Performance control

In the third phase of the simulation process, DADS decides to change the scheduler of the real disk if the performance achieved by *VD_VD* improves the performance obtained by *VD_RD*, because it is expected an improvement in the performance of the real disk too. In other words, if T_{VD_RD} denotes the total service time of *VD_RD*, and T_{VD_VD} specifies the total service time of *VD_VD*, a scheduler switch is done if the condition

$$T_{VD_VD} < T_{VD_RD} \tag{4.1}$$

is true.

As we are aware that an I/O scheduler switch is a time–consuming process (draining the old scheduler's queue and initializing the new one are expensive operations), we take into account an estimation of the time needed to do the change. Furthermore, since the simulation performed is very precise, but not exact, a scheduler change is only done if the improvement achieved by *VD_VD* is larger than 5%, thereby allowing a certain margin of error. Consequently, Equation 4.1 is modified to

$$T_{VD_VD} + T_{Change} < 0.95 \cdot T_{VD_RD} \tag{4.2}$$

where T_{Change} denotes the time estimated to carry out the scheduler change, and is calculated by using the number of pending requests in the scheduler queue of the real disk and an average I/O time per request calculated in the previous scheduler change.

Since, for some access patterns, the I/O schedulers in the virtual disks may present a similar behavior, time differences can be slightly greater than 5% many times during a workload. This could produce a frequent scheduler change, and hurt the performance. Hence, in order to avoid so many changes, we have implemented a mechanism that, when 5 changes in a row are detected, permanently assigns the *default* scheduler to the real disk. We have considered that the default scheduler is the one specified by the system administrator based on her experience, personal preferences, expected performance, etc., and not the default imposed by the operating system. The disk simulators, however, keep comparing the service times

[1] Although, in the current implementation, the virtual disks really exchange their schedulers, this is not strictly necessary. Each instance can always use the same scheduler, and DADS can assign the best one to the real disk at every check.

achieved by the schedulers, and the change mechanism is re–activated when no change has been predicted in the last 7 checks.

4.5. Experiments and methodology

In order to investigate the behavior of our scheduler change mechanism, DADS has been tested with six different disks, several access patterns, and the four I/O schedulers available in Linux. This section presents the hardware platform, configuration of the simulated disk cache of each tested disk, benchmarks and I/O schedulers used for carrying out the analysis.

4.5.1. Hardware platform

Two computers, with several disks each, have been used in our study. In each computer, one disk is the system disk that contains a Fedora Core 8 operating system, and is used for collecting traces to evaluate the proposal. The other disks are the test drives, some of them are hard drives and other SSD drives. The main features of the computers and the four test hard disks are presented in Table 4.1. Table 3.5, in Chapter 3, presents the main features of the other two disks that are SSD drives.

A computer is **Hera**, also used in Chapter 3 (Section 3.4.1 provides the description of this computer). **Hera** has four test disks: two hard drives, and two SSD drives. The four drives but one are the same as those used in Chapter 3; we have had to replace the Seagate ST3400620AS disk [6] because it stopped working.

The new test drive is a Seagate ST3250310NS disk [6], that has a capacity of 250 GB and a 32 MB built–in cache. As the failed ST3400620AS disk, it has a clean *Ext3* file system, containing nothing but the files used for the tests. It was formatted, and then the files were created. During the explanation of the results, we refer to this test disk as “HD–250–32”[2].

The second hard disk drive is a Samsung HD322HJ disk [48], already described in Section 3.4.1. The same partition is used for carrying out the benchmarks. Here, this disk is referred as “HD–320–16”.

The two SSD drives are an Intel X–25M SSDSA2MH160G2C1 and an Intel X–25E SSDSA2SH064G1GC [104], whose descriptions are given in Section 3.6.2. To facilitate the explanation, we again refer to these disks as “SSD–160” and “SSD–64”, respectively.

The second computer, **Hecate**, is a 1.86 GHz Intel dual–core system with 2 GB of RAM, and has three disks: a system disk and two test hard drives. The system disk is a ST3400620AS disk [6]. Its first test drive is a Seagate ST3500630AS disk [6], with a capacity of 500 GB and 16 MB of built–in cache. The second disk is a Seagate ST3500320NS [6], with a capacity of 500 GB and 32 MB of built–in cache. Both test drives have a clean *Ext3* file system, containing nothing but the files used for the tests. They were formatted and then the files were created. We refer to these test hard disks as “HD–500–16” and “HD–500–32”, respectively.

4.5.2. Disk caches configurations

Regarding the configuration of the simulated disk caches, after executing the capturing program (see Section 4.3.6), we have established the following values:

[2] Note that in the name “HD–250–32”, “HD” stands for Hard Disk, “250” is the disk capacity, and “32” is the size of its cache.

Table 4.1: Main parameters of the four test hard disk drives.

Features	Values	
	Hera	**Hecate**
1st Test Disk	Seagate ST3250310NS	Seagate ST3500630AS
Capacity	250 GB	500 GB
Cache	32 MB	16 MB
Read	Adaptive	Adaptive
Write	Yes	Yes
Average latency	4.16 ms	4.16 ms
Rotational Speed	7200 RPM	7200 RPM
Seek time		
Read	8.5 ms (average)	$<$ 8.5 ms (average)
Track–to–track, read	0.8 ms (typical)	$<$ 0.8 ms (typical)
Nick name	"HD–250–32"	"HD–500–16"
2nd Test Disk	Samsung HD322HJ	Seagate ST3500320NS
Capacity	320 GB	500 GB
Cache	16 MB	32 MB
Read	Adaptive	Adaptive
Write	Yes	Yes
Average latency	4.17 ms	4.16 ms
Rotational Speed	7200 RPM	7200 RPM
Seek time		
Read	8.9 ms (average)	8.5 ms (average)
Track–to–track, read	0.8 ms (typical)	0.8 ms (typical)
Nick name	"HD–320–16"	"HD–500–32"

- "HD–250–32": we have considered that its cache of 32 MB is divided into 63 segments, and that for both sequential and non–sequential accesses the read–ahead size is 256 sectors.
- "HD-320-16": the 16 MB of disk cache are split into 64 segments. When a sequential access is detected, the read–ahead size is 256 sectors, and, for a non–sequential, it is 96 sectors.
- "HD–500–16": we have considered that the cache of 16 MB is split into 20 segments. The read–ahead size for a sequential access is 256 sectors, and, for a non–sequential pattern, it is 32 sectors.

Table 4.2: Parameters of the simulated disk caches.

Disk Model	Cache size	# seg	read–ahead size	
			sequential	non sequential
HD–250–32 (ST3250310NS)	32 MB	63	256 sectors	256 sectors
HD–320–16 (HD322HJ)	16 MB	64	256 sectors	96 sectors
HD–500–16 (ST3500630AS)	16 MB	20	256 sectors	32 sectors
HD–500–32 (ST3500320NS)	32 MB	128	256 sectors	256 sectors
SSD–160 (SSDSA2MH160G2C1)	N/A	N/A	–	–
SSD–64 (SSDSA2SH064G1GC)	N/A	N/A	–	–

- "HD–500–32": the 32 MB of cache are divided into 128 segments. We have considered that the read–ahead size for both contiguous and random accesses has been set to 256 sectors.

For "SSD–160" and "SSD–64", as explained previously, no disk cache has been simulated. Table 4.2 summarizes these values for the six test disks. Note that, for each disk, the size of the simulated disk cache has been set to the same size as the original one.

4.5.3. Benchmarks

We have analyzed the performance that can be achieved with DADS by running a test that executes several benchmarks in a row, one after another, without restarting the computer until the last is done. In this way, we show how our mechanism switches the I/O scheduler and adapts itself to changes on the workload. The selected benchmarks are the same as those used in Chapter 3: *Linux Kernel Read (LKR)*; *IOR Read (IOR)*; *TAC*; *512 kB Strided Read (512k–SR)* and *8 kB Strided Read (8k–SR)*. Section 2.4.3 gives a detailed description of all these benchmarks, except the 8k–SR that is described in Section 3.4.2.

To establish the execution order of the benchmarks, we have taken into account the performance of the I/O schedulers on each test. Therefore, we have sorted them trying to cause a scheduler change. The resultant execution order is: IOR; LKR; 512k–SR; TAC; and 8k–SR.

The tests have been executed for 1, 2, 4, 8, 16, and 32 processes. Since some of the benchmarks use the same files, we have established that until 16 processes, each benchmark, except LKR, uses files which have not been used by the previous benchmark, reducing the effect of the buffer cache. However, for 32 processes, as there are only 32 files, it is not possible to meet this restriction.

4.5.4. I/O schedulers of the experiments

To get a better insight into the mechanism proposed to achieve a dynamic scheduling, we have tested the four I/O schedulers available in Linux (AS, CFQ, Deadline and Noop). DADS has been tested by comparing these schedulers two by two except for SSD drives, where CFQ and AS present the worst behavior for these devices, and no comparison between them has

been made. Section 3.2.2, in Chapter 3, briefly describes the main features of these four I/O schedulers.

4.6. Results

DADS and the in–kernel virtual disk simulator have been implemented in a Linux kernel 2.6.23. The new kernel is called the *DADS kernel*. We have carried out several experiments by comparing two by two the Linux I/O schedulers. The goal is to study the behavior of the proposed mechanism and the performance that it can achieve. The results are compared with those achieved by the corresponding schedulers in a vanilla Linux kernel 2.6.23, that has not been modified, and that, to short, we call the *original kernel*.

Activity of the test disks has been traced by instrumenting both kernels to record when a request starts, finishes, and arrives at the scheduler queue. The DADS kernel also records information about the scheduler changes.

For all the experiments, the duration of the first phase, where requests are collected, has been set to 5 seconds. We have also performed several tests using an interval of 10 seconds, and the sole difference is that scheduler changes take longer to occur. These results have not been included in the present thesis.

Tests have been carried out with all the possible configurations of the DADS kernel, depending on the I/O schedulers to compare, and the I/O scheduler initially assigned to the real disk. Realize that the initial scheduler assigned to the real disk is also the *default* scheduler, that is, the one that is selected when a high rate of scheduler changes occurs (see Section 4.4.3). For example:

- *AS–Deadline* means that, initially, the real disk has the AS scheduler, which is set as the default scheduler, *VD_RD* also has AS–VD, and *VD_VD* uses Deadline. The comparison is made between AS and Deadline.
- *Deadline–AS* means that, initially, the real disk and *VD_RD* use Deadline, which is now the default scheduler, and *VD_VD* has AS–VD. Again, AS and Deadline are compared.

All the benchmarks have also been carried out by using the original kernel and the four I/O schedulers. So, for each disk, we compare the performance of two schedulers A and B with the following configurations: DADS kernel with the configuration *A–B*; DADS kernel with *B–A*; original kernel with the scheduler A; and original kernel with the scheduler B.

Figures show how the DADS kernel adapts to the best of the two compared schedulers. They also show the improvement achieved by each configuration over the worst one. That is, if T represents the application time, figures show:

$$\frac{T_{conf}}{Max(T_{A-B}, T_{B-A}, T_A, T_B)}, \tag{4.3}$$

where T_{conf} is the application time of the compared configuration, T_{A-B} and T_{B-A} are the application times produced by the DADS kernel for configurations *A–B* and *B–A*, respectively, and T_A and T_B are the application times achieved by the original kernel for the A and B schedulers, respectively. Moreover, to facilitate the comparison, solid and dashed lines are used for showing the results obtained by the original kernel, and histograms for the results of the DADS kernel.

In the figures, besides the results for the individual benchmarks, the result for the total application times of the test, calculated as the sum of the application times of the benchmarks, is also showed. This new data summarizes the behavior of our approach during the whole execution, and shows how our mechanism can reduce the overall I/O time. Furthermore, results are grouped by the number of processes, what allows us to observe how the scheduler is changed from one test to the following, if required.

The results shown for every configuration are the average of five runs. The confidence intervals, for a 95% confidence level, have also been calculated, and are less than 5%, however, for clarity, they have been omitted.

All the tests have been done with a cold page cache (i.e., the computer is restarted after each run). The tables obtained from the off–line training are given to the virtual disk each time the system is initialized. Therefore, tables are initially the same in all the cases.

Firstly, we are going to explain the results for the hard disk drives, then for the SSD drives.

4.6.1. Hard disk drives

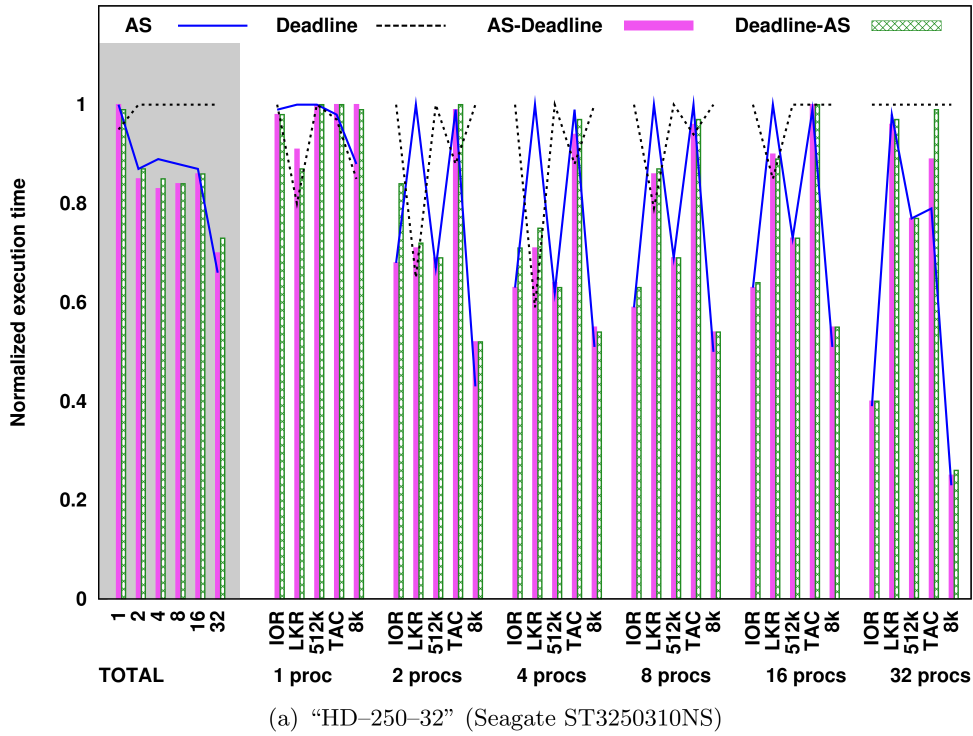
Figure 4.4 shows the experimental results for the four hard disks and schedulers AS and Deadline, i.e., configurations *AS–Deadline* and *Deadline–AS* of the DADS kernel are compared with the results of those schedulers achieved by the original kernel. Experimental results for AS and Noop are shown in Figure 4.5. Figure 4.6 presents the results for the schedulers CFQ and AS, and Figure 4.7 depicts the results for CFQ and Deadline. Finally, the combination CFQ and Noop is presented in Figure 4.8. For a given number of process, figures show how a change in the benchmark (due to the execution of the benchmarks in a row) can produce a change of the I/O scheduler depending on the performance of each scheduler on the next benchmark. The next paragraphs describe some interesting details, firstly, for the global execution of the test, and then, for each benchmark independently.

TOTAL

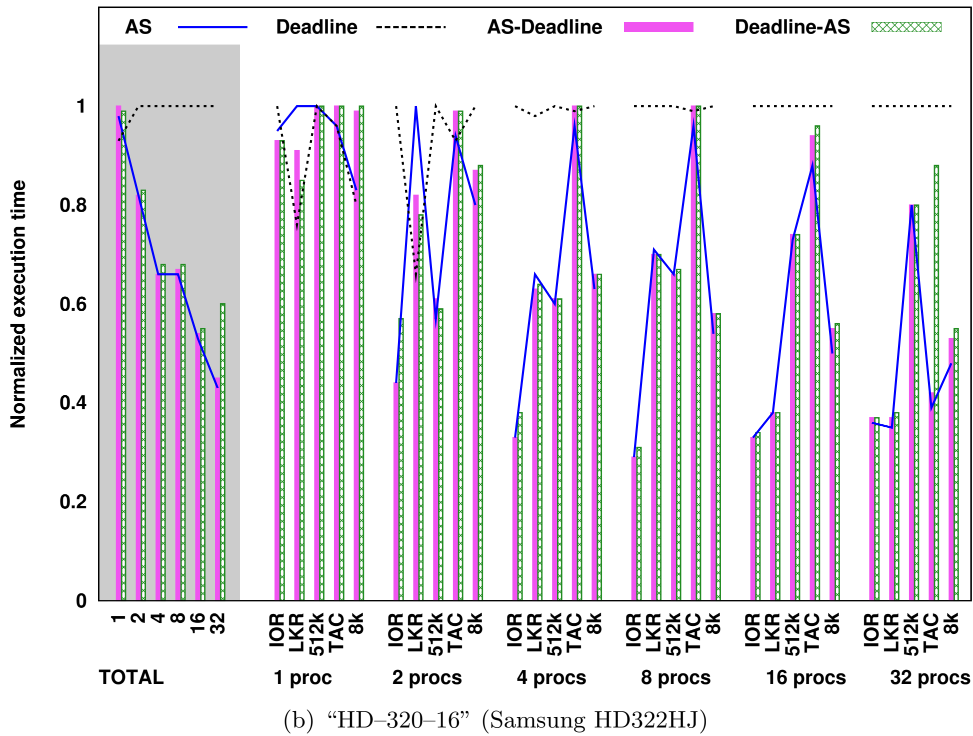
The results for the total application time, that appear in the first histogram in the figures, show the global behavior of DADS. As we can see, DADS follows the best scheduler, changing the scheduler, if necessary, when the number of processes also changes. Adaptation can easily be seen in figures, specially for the disks “HD–250–32” and “HD–500–32”. Furthermore, our proposal even outperforms the scheduler that presents the best behavior in several cases. For instance, when AS is compared to Deadline or Noop, DADS outperforms the best scheduler (AS in both cases), for 2, 4 and 8 processes, and for the disks “HD–250–32” and “HD–500–32”, achieving improvements of up to 7.4% and 3.4%, respectively, over the performance of the AS scheduler (see Figures 4.4(a), 4.4(d), 4.5(a) and 4.5(d)). For the *CFQ–Deadline* and *Deadline–CFQ* configurations, and disks “HD–500–16” and “HD–500–32”, our method improves the performance of CFQ in a 3.5% for 32 processes, being CFQ the scheduler that achieves the largest throughput for both disks in a vanilla Linux kernel (see Figures 4.7(c) and 4.7(d)).

IOR Read

For the sequential access pattern imposed by IOR, DADS works as expected, and it adapts to the best scheduler. Only a few details have to be clarified.

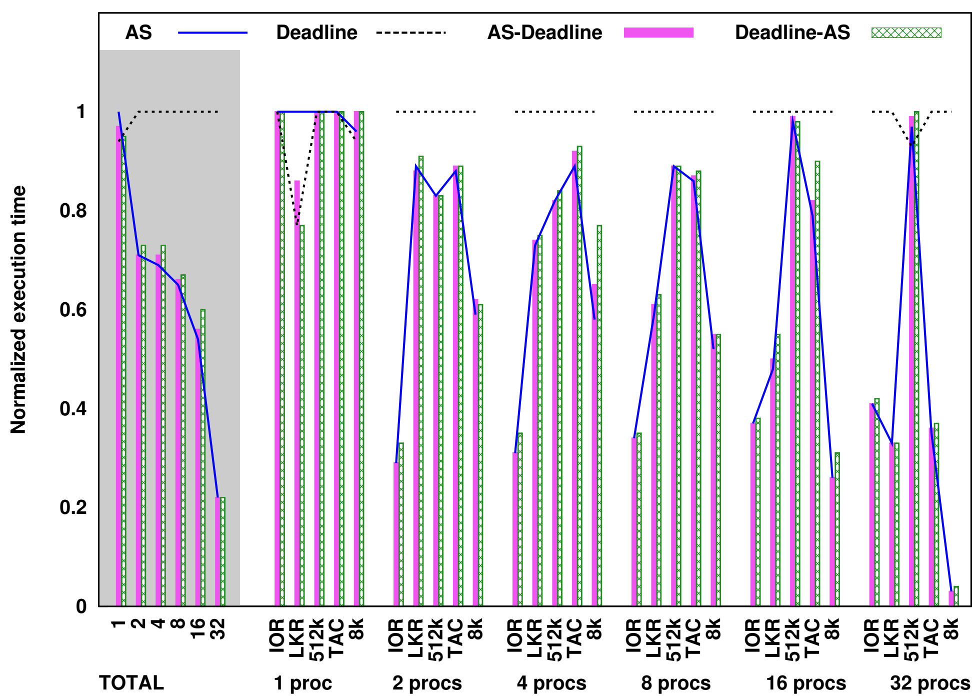


(a) "HD–250–32" (Seagate ST3250310NS)

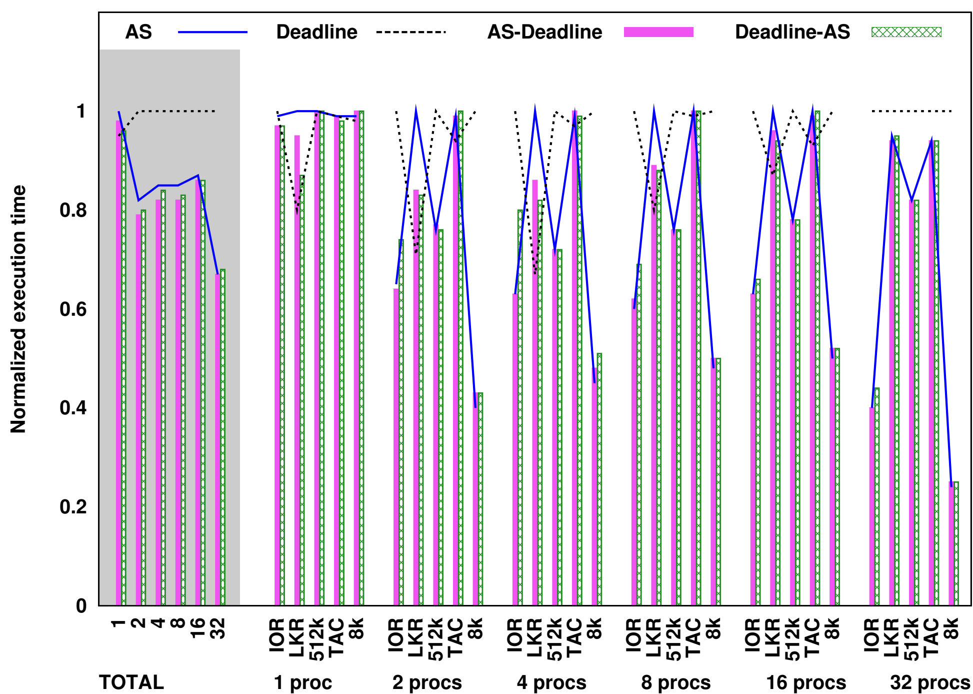


(b) "HD–320–16" (Samsung HD322HJ)

Figure 4.4: Configurations *AS–Deadline* and *Deadline–AS* for hard disk drives.

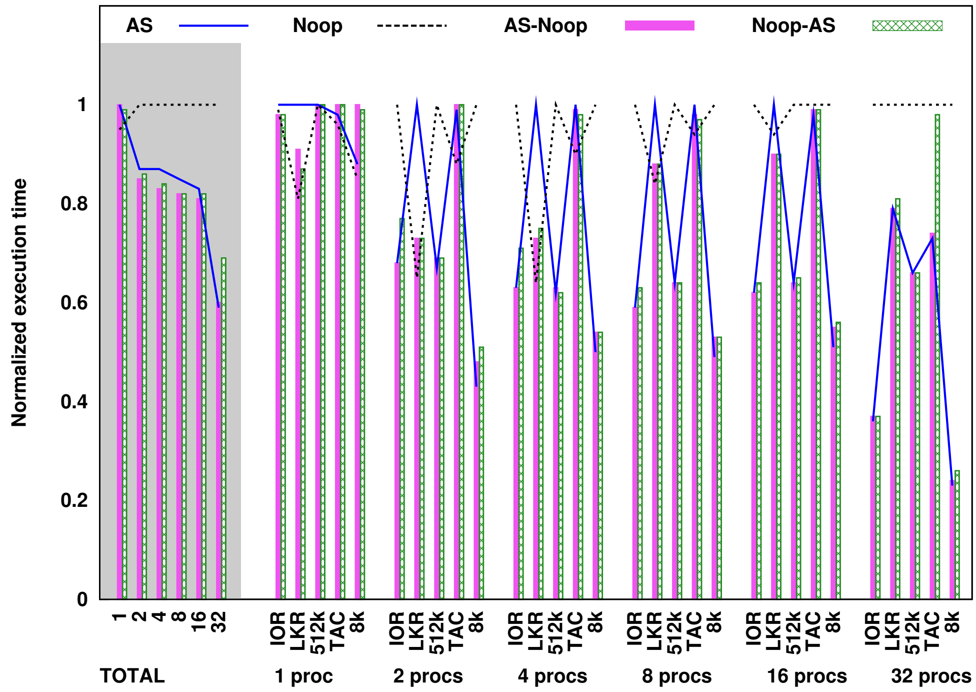


(c) "HD–500–16" (Seagate ST3500630AS)

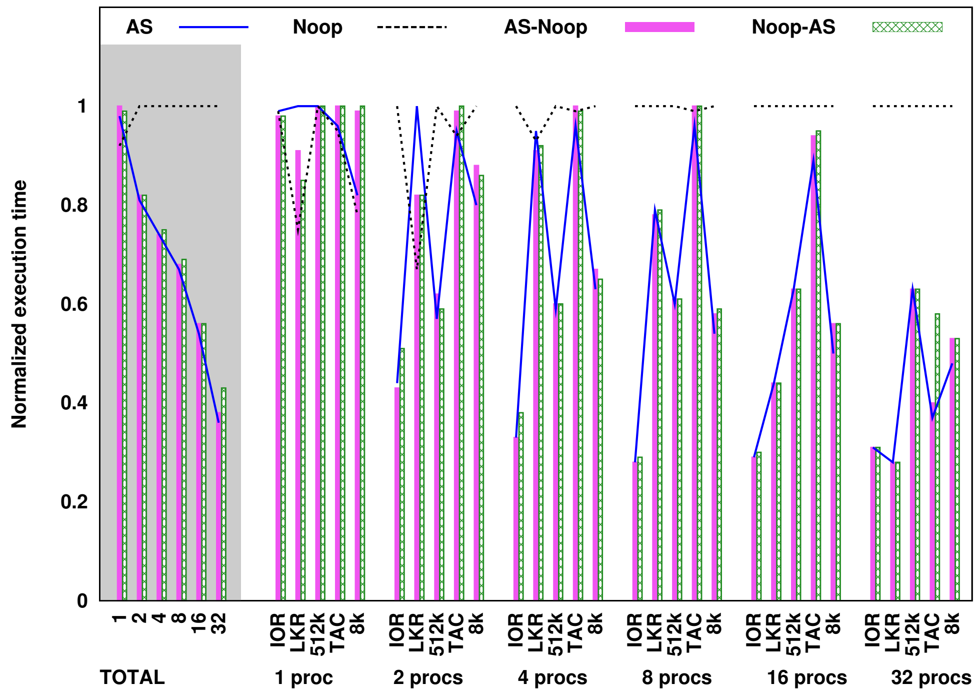


(d) "HD–500–32" (Seagate ST3500320NS)

Figure 4.4: (Cont.) Configurations *AS–Deadline* and *Deadline–AS* for hard disk drives.

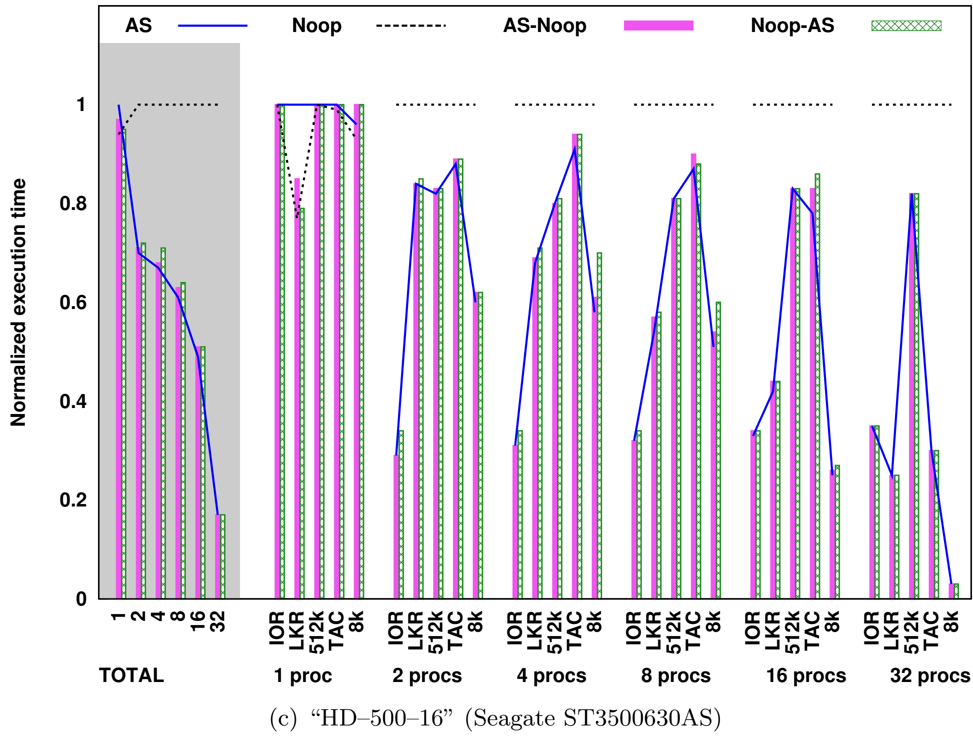


(a) "HD–250–32" (Seagate ST3250310NS)

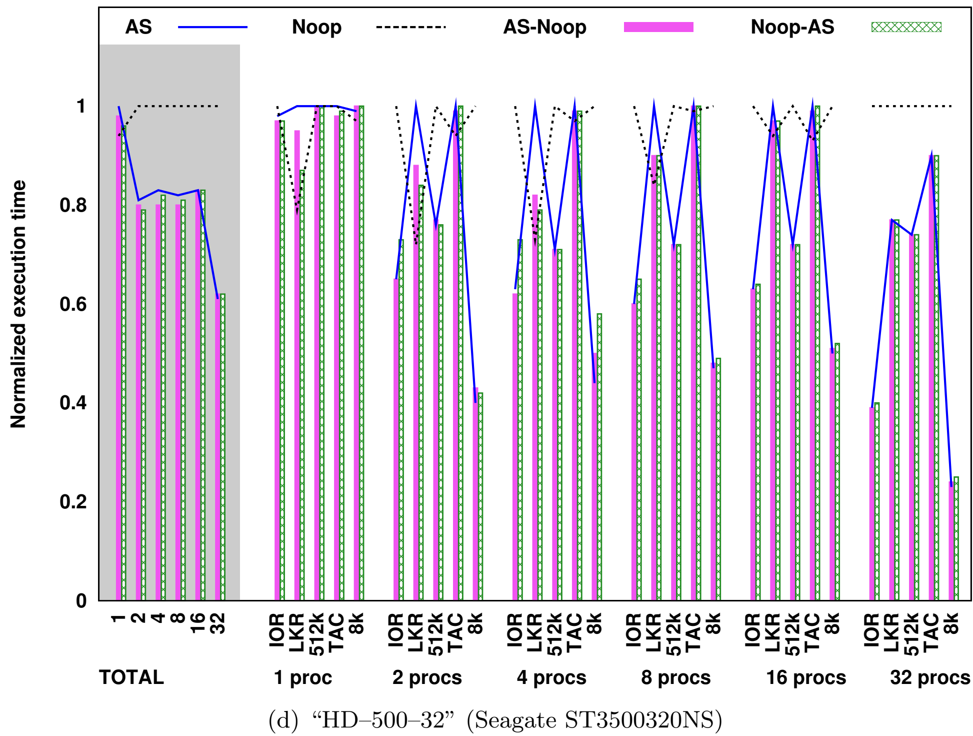


(b) "HD–320–16" (Samsung HD322HJ)

Figure 4.5: Configurations *AS–Noop* and *Noop–AS* for hard disk drives.

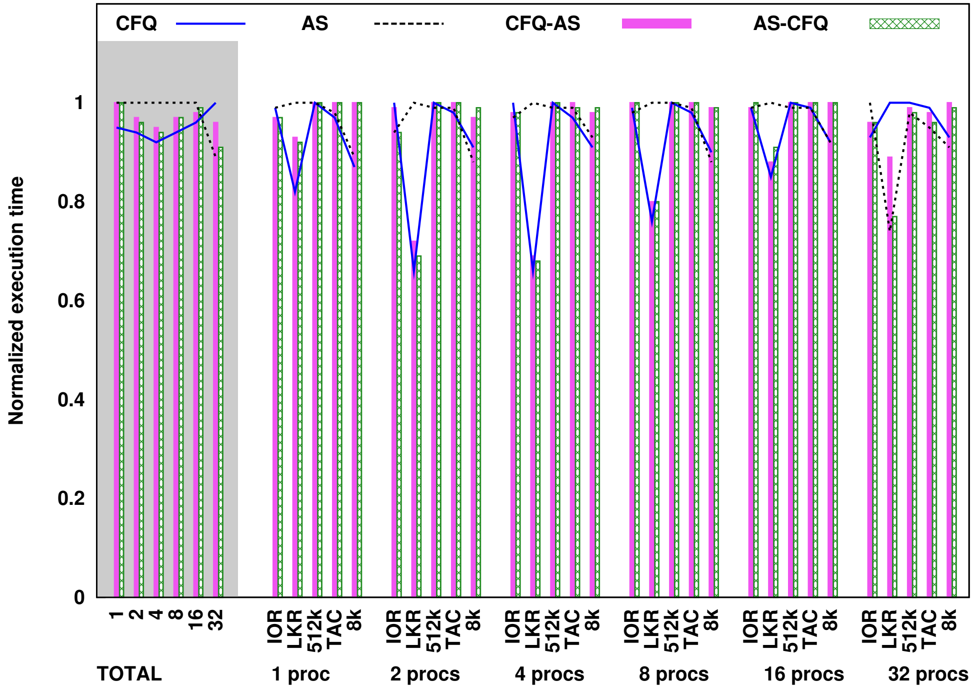


(c) “HD–500–16” (Seagate ST3500630AS)

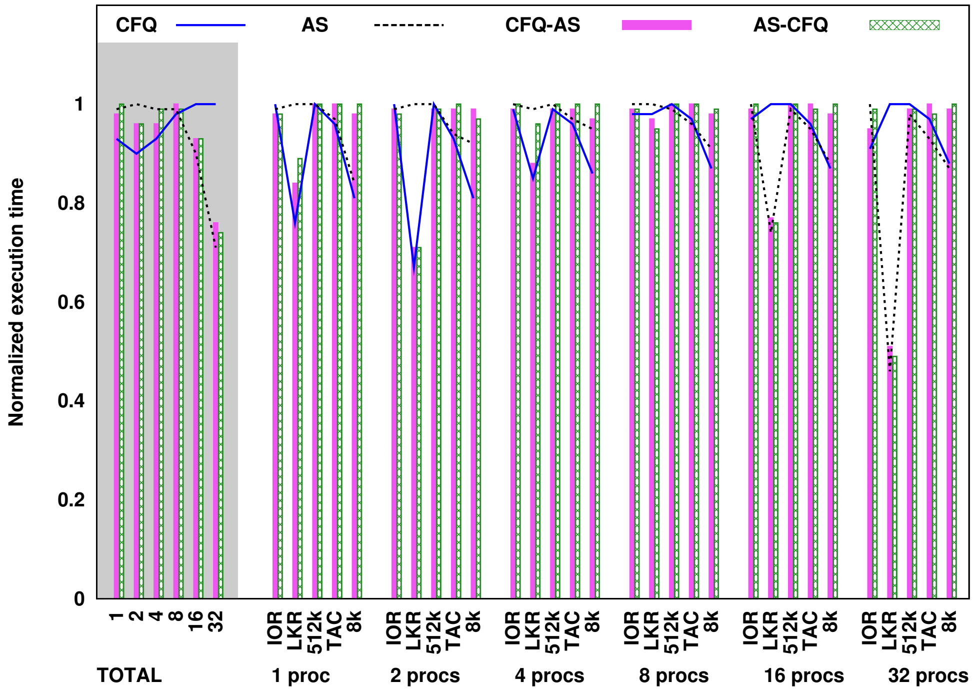


(d) “HD–500–32” (Seagate ST3500320NS)

Figure 4.5: (Cont.) Configurations *AS–Noop* and *Noop–AS* for hard disk drives.

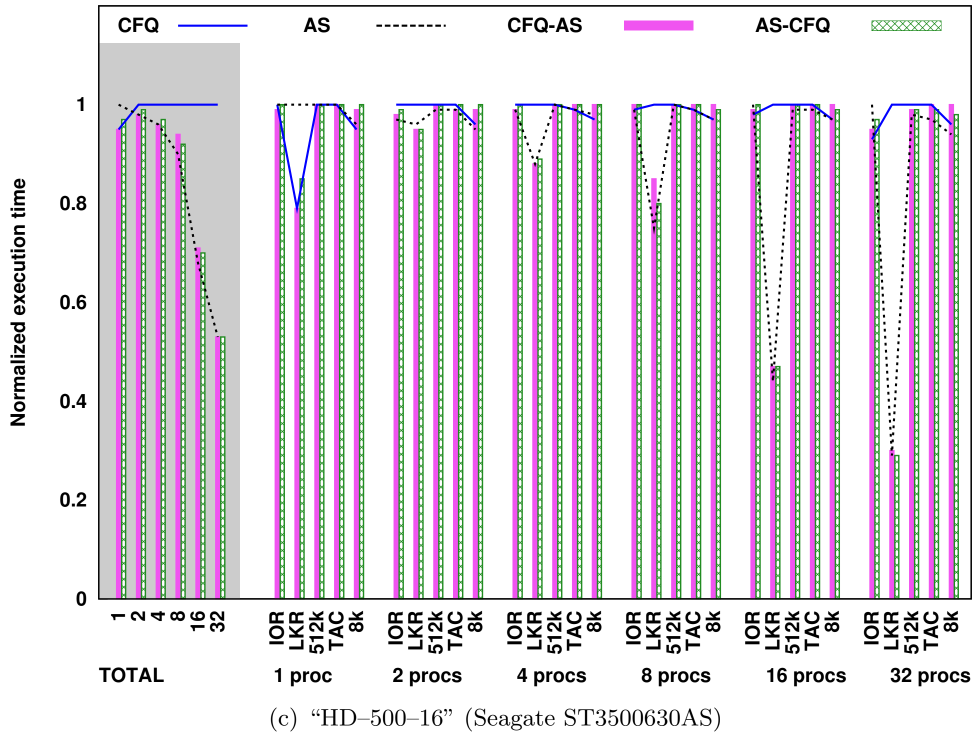


(a) "HD–250–32" (Seagate ST3250310NS)

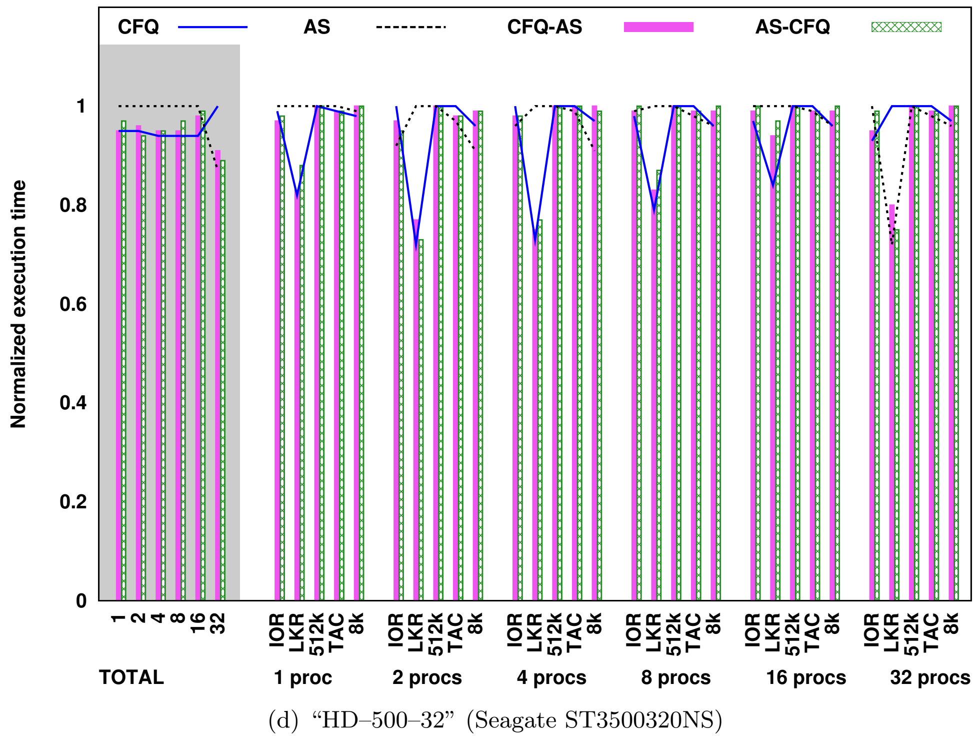


(b) "HD–320–16" (Samsung HD322HJ)

Figure 4.6: Configurations *CFQ–AS* and *AS–CFQ* for hard disk drives.

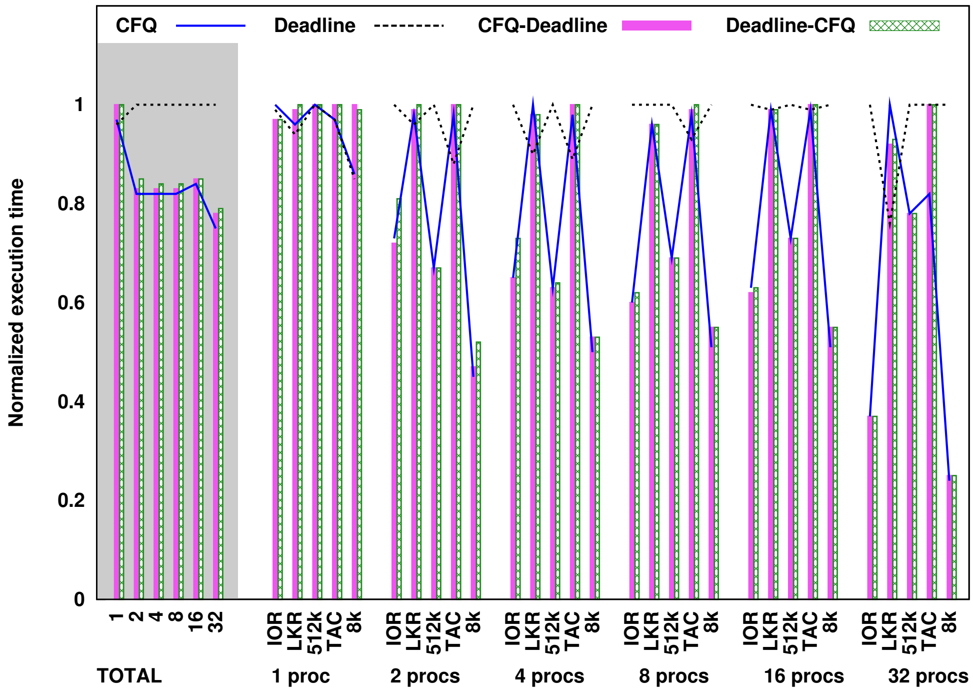


(c) "HD–500–16" (Seagate ST3500630AS)

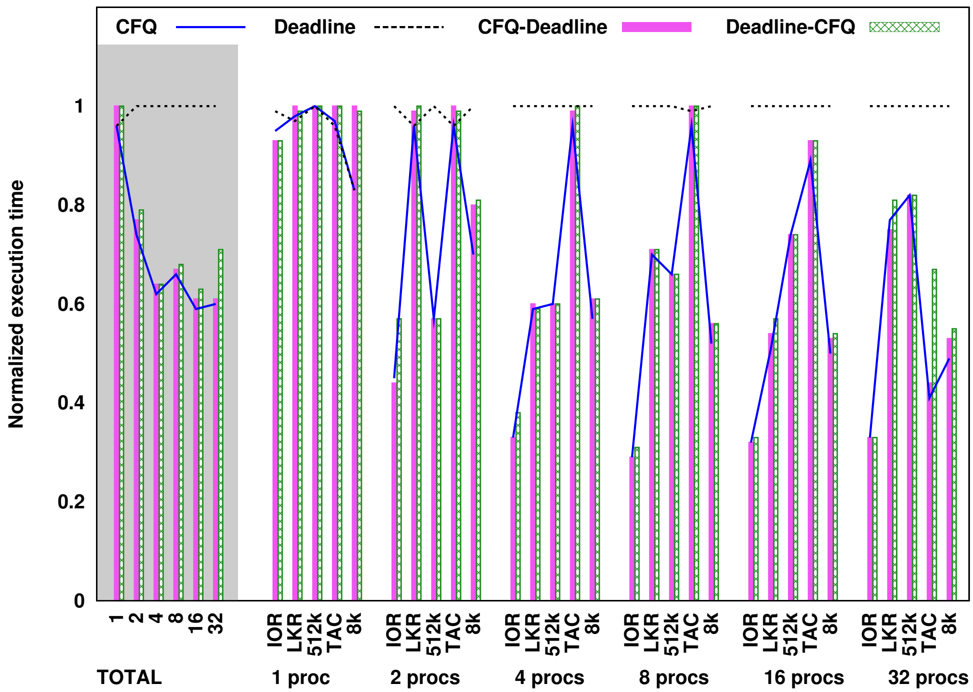


(d) "HD–500–32" (Seagate ST3500320NS)

Figure 4.6: (Cont.) Configurations *CFQ–AS* and *AS–CFQ* for hard disk drives.

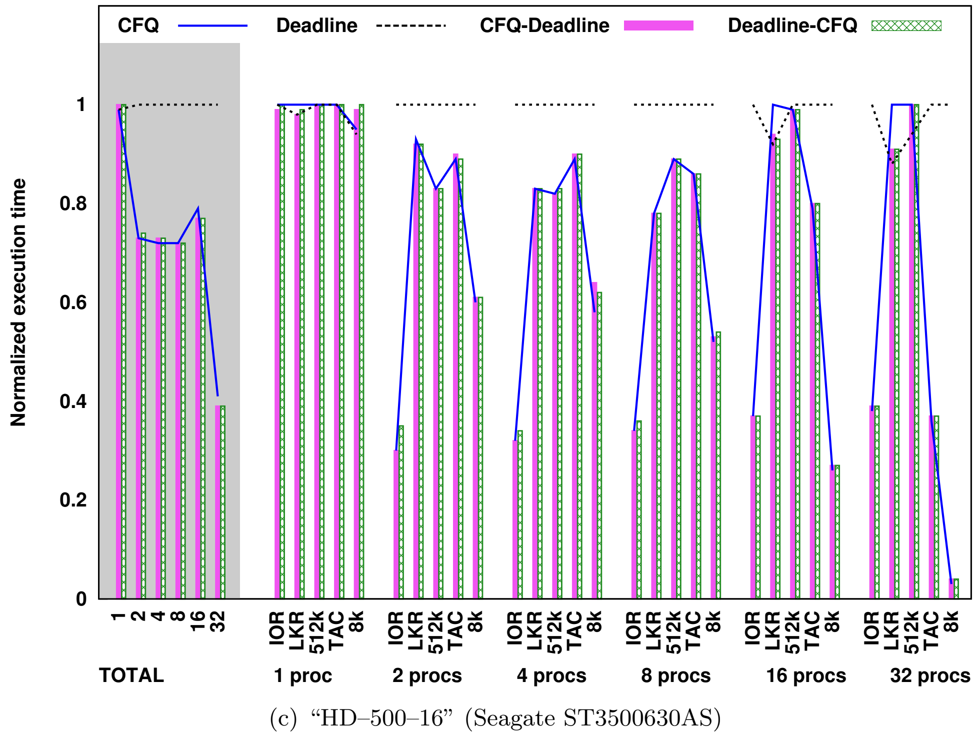


(a) "HD–250–32" (Seagate ST3250310NS)

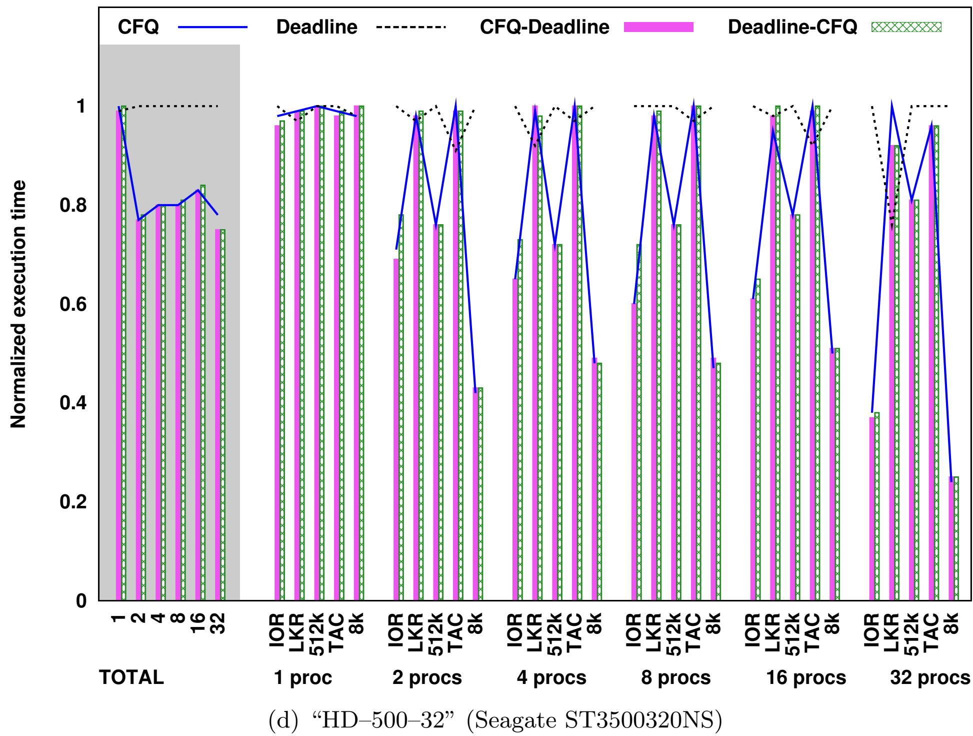


(b) "HD–320–16" (Samsung HD322HJ)

Figure 4.7: Configurations *CFQ–Deadline* and *Deadline–CFQ* for hard disk drives.

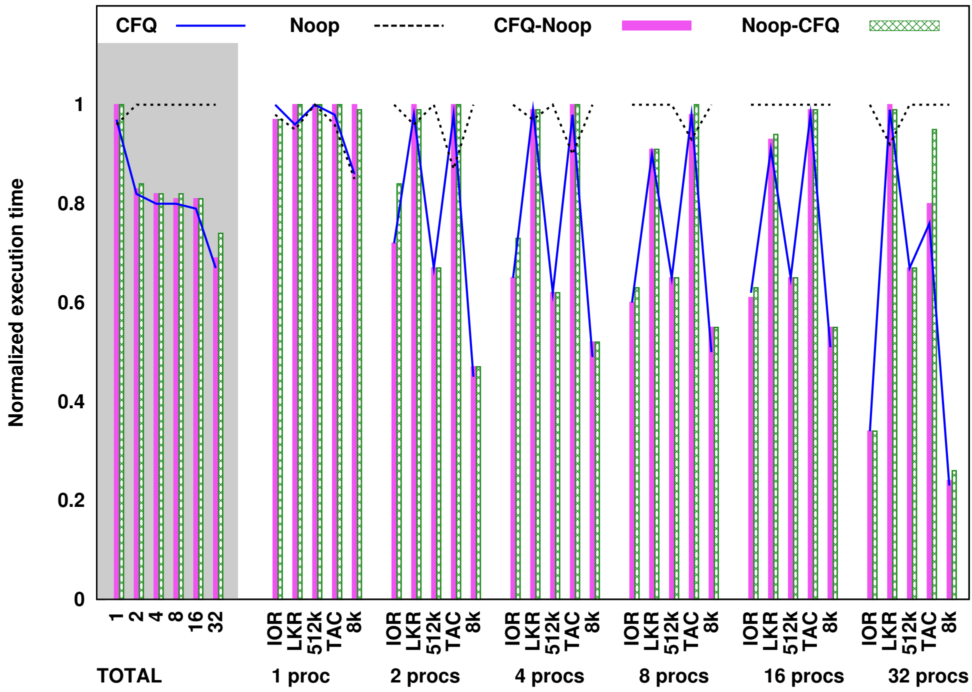


(c) "HD–500–16" (Seagate ST3500630AS)

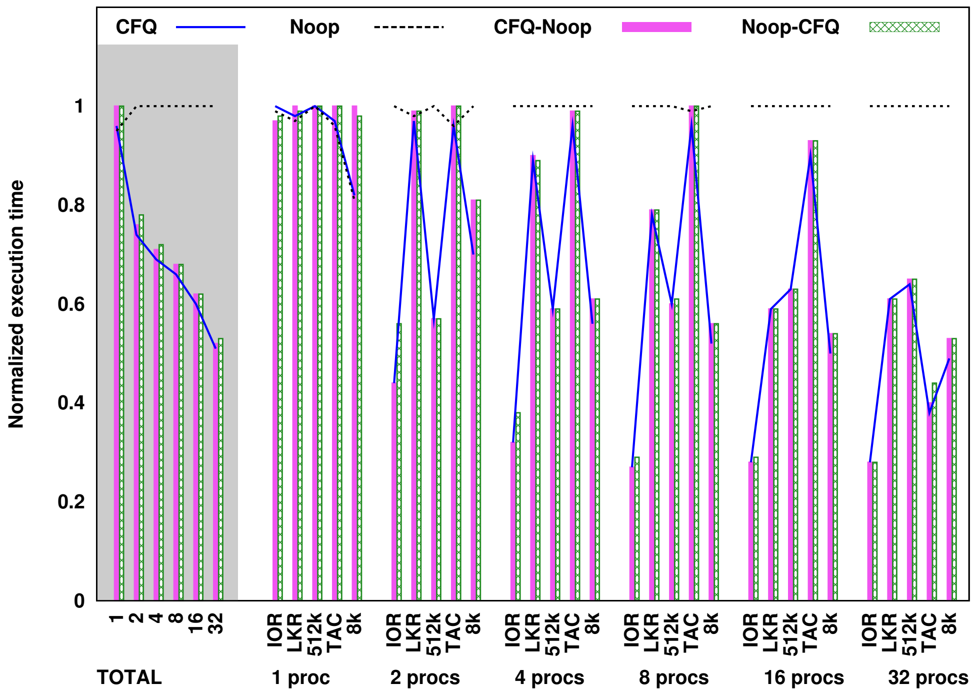


(d) "HD–500–32" (Seagate ST3500320NS)

Figure 4.7: (Cont.) Configurations *CFQ–Deadline* and *Deadline–CFQ* for hard disk drives.

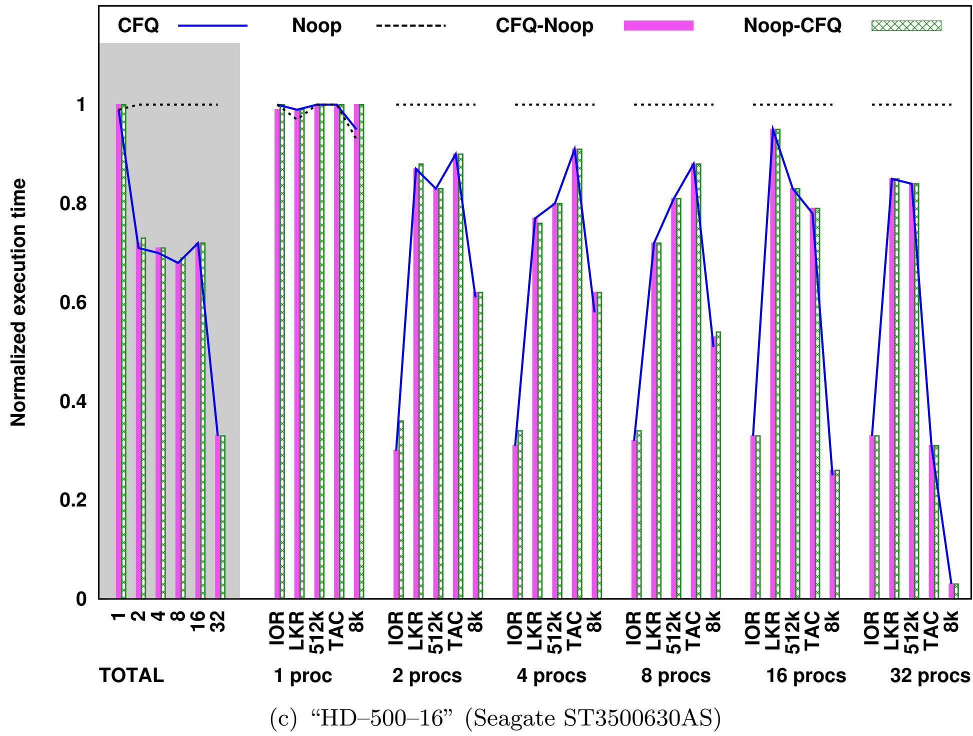


(a) "HD–250–32" (Seagate ST3250310NS)

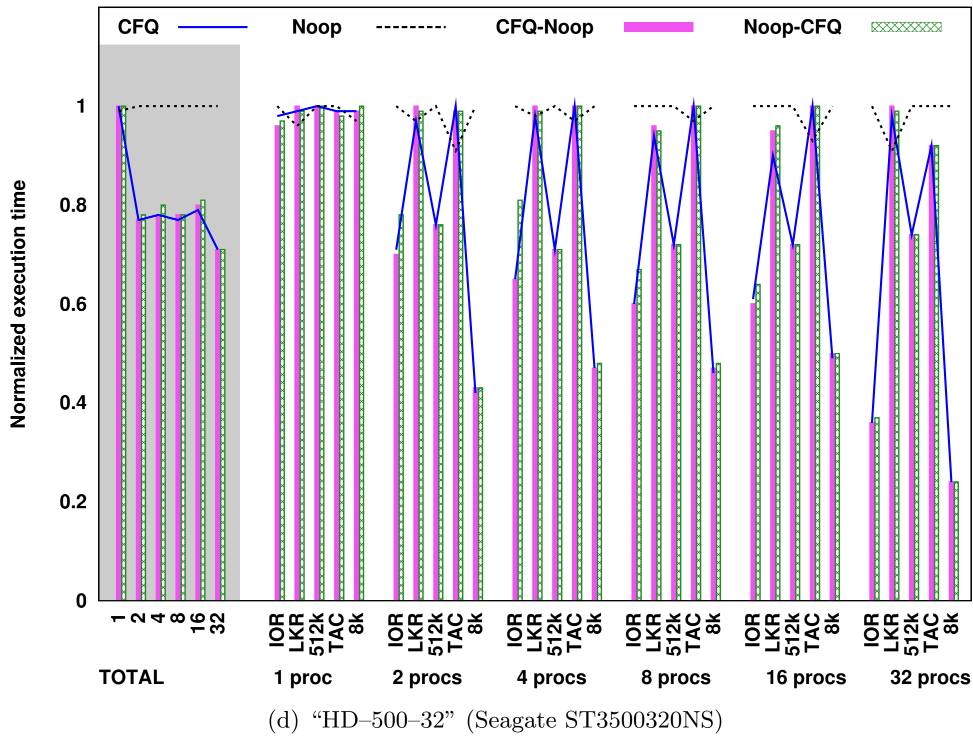


(b) "HD–320–16" (Samsung HD322HJ)

Figure 4.8: Configurations *CFQ–Noop* and *Noop–CFQ* for hard disk drives.



(c) “HD–500–16” (Seagate ST3500630AS)



(d) “HD–500–32” (Seagate ST3500320NS)

Figure 4.8: (Cont.) Configurations *CFQ–Noop* and *Noop–CFQ* for hard disk drives.

For the configurations *Deadline–AS*, *Noop–AS*, *Deadline–CFQ* and *Noop–CFQ* (see Figures 4.4, 4.5, 4.7 and 4.8, respectively), and any number of processes but 32, there is a small degradation with respect to the best scheduler, because the worst scheduler of the two compared is initially used. DADS usually changes the scheduler in the first check, but the time lost during this initial interval cannot be recovered in the short duration of the test.

When the comparison is among AS and CFQ (see Figure 4.6), I/O times of both schedulers are very similar, and the DADS kernel adapts itself quite well except for 16 and 32 processes. In these cases, our technique is not able to decide which scheduler achieves the best performance, activating the mechanism to avoid a high rate of changes (see Section 4.4.3). The default scheduler is selected most of the times, so the configuration *AS–CFQ* tends to have a similar behavior to the original kernel with the AS scheduler, whereas the behavior of *CFQ–AS* is similar to that of the original kernel with CFQ.

Linux Kernel Read

With the *Linux Kernel Read* benchmark, our mechanism adapts itself to the best scheduler, although, in some cases, there is a small degradation. The problem is that LKR initially has the worst scheduler for its access pattern, because the scheduler that provides the best result for this benchmark is different from the one that presents the highest performance for IOR (the test executed previously). Nonetheless, although DADS changes the scheduler in first check, the increase in I/O time initially produced by the worst scheduler hurts the final result. This problem appears for all the disks except "HD–500-16".

512 kB Strided Read

The DADS kernel presents, with all the configurations, the same behavior than the best scheduler in the original kernel, and only two cases are remarkable. When comparing AS vs Deadline and CFQ vs Deadline, for the "HD–500-16" disk and 32 processes, DADS does not select the best scheduler (Deadline in both cases), and spends all the time with AS or CFQ, respectively (see Figures 4.4(c) and 4.7(c)). For 32 processes and the comparison between AS and Deadline, the mechanism to avoid frequent changes is even put into effect, because many and frequent changes from one scheduler to another are made. In the vanilla kernel, application time differences between the compared schedulers are quite small, less than 3.8% among AS and Deadline, and 5.7% among CFQ and Deadline. Differences in I/O time, also in the original kernel, are even smaller, less than 1.5% and 3.5%, respectively. The problem is that our mechanism does not detect such small differences, and wrongly selects, most of the times, AS or CFQ, in each case.

TAC

With the backward access pattern imposed by TAC, the behavior of DADS depends on the test disk. For "HD–500–16", our proposal selects the best scheduler and achieves the same performance as the original kernel with the same scheduler. However, for the other three disks, our technique does not make the right selection, and slightly increases the service time with respect to the best achievable result. For any number of processes but 32, the problem is that the four schedulers obtain almost the same performance, and our mechanism

is not able to catch such small difference. In these cases, the DADS kernel introduces a small overhead which increases slightly the application time, although the I/O time remains the same. This small overhead is due to DADS operation, and is noticeable in this test, and also in 8k–SR, because the number of requests served per second by both benchmarks is much higher than in the other tests. For 32 processes, our proposal frequently alternates from one scheduler to another, and the mechanism to avoid such frequent changes is put into effect. Thus, performance of each configuration is close to its default scheduler's one.

8 kB Strided Read

Finally, for this access pattern, the obtained results depends on the configuration. For all of them, except for the comparison between AS and CFQ, DADS always selects the scheduler that achieves the best performance (see Figures 4.4, 4.5, 4.7, and 4.8). Nevertheless, in some cases, it introduces a small overhead that slightly increases the application time with respect to the original kernel and the best scheduler. This degradation is more noticeable for 1 process because the application time is quite small. In this case, this overhead is due to the DADS operation, as explained for the TAC test, but also to delays in the request arrivals (see below).

Regarding the comparison between AS and CFQ (see Figure 4.6), in the original kernel, both schedulers produce almost the same throughput, and the application time difference between them is rather small, only 2.7% on average. However, DADS increases the application time due to the small overhead introduced by its management.

An unexpected result in this benchmark is that the DADS kernel slightly increases the I/O time (and, hence, the application time) with respect to the original kernel when both use the same scheduler. The cause is the small overhead that DADS adds when it copies a real request to the shared queue in the disk simulator. This small overhead delays the arrival of the requests to the scheduler queue of the real disk. The delay is quite small, but big enough to make the disk spin almost a full rotation, because the requested sectors have just passed. Nevertheless, it is important to note that this problem only affects requests which are cache misses, and jump a small amount of sectors with respect to a previous request; it also depends on the disk model (different hard drives can have different sector layouts). Therefore, the problem does not appear in the other benchmarks and is almost negligible for the "HD–500–16" and "HD–500–32" drives.

Table 4.3 shows the average I/O time per request for the DADS kernel and configurations *CFQ–Deadline* and *CFQ–Noop*, and for the original kernel and the CFQ scheduler, for the four disks tested. As we can see in Table 4.3(b), for "HD–320–16", differences, in average I/O time per request, are larger than for the other three. Hence, for this disk, the small degradation introduced by the DADS's overhead is more noticeable. Time differences are larger for 1 and 2 processes, because, in both cases, to attend the issued requests, the disk has to perform less seeks, which implies a small seek time. Therefore, the rotational delay, which is increased by the delay of requests, has a great impact in I/O times.

For "HD–250–32" (see Table 4.3(a)), there are also small time differences, that can be seen as a small degradation in the obtained results. For "HD–500–16" and "HD–500–32" (see Tables 4.3(c) and 4.3(d)), time differences are almost negligible. For the sake of simplicity, the average I/O times per request obtained for the AS scheduler are omitted, albeit, differences are quite similar to those presented here for the CFQ scheduler.

Realize that, since the overhead is more noticeable in the two test disks that are in the same

Table 4.3: Average I/O time per request measured during the execution of the *8 kB Strided Read* test, for the CFQ scheduler with the original kernel, and the configurations *CFQ–Deadline* and *CFQ–Noop* of the DADS kernel, for 1, 2, 4, 8, 16, and 32 processes.

(a) "HD–250–32"

	Processes					
Kernel: Scheduler	**1**	**2**	**4**	**8**	**16**	**32**
Original: CFQ	147 μs	200 μs	203 μs	190 μs	182 μs	187 μs
DADS: *CFQ–Deadline*	159 μs	205 μs	209 μs	205 μs	194 μs	195 μs
DADS: *CFQ–Noop*	159 μs	206 μs	209 μs	211 μs	194 μs	192 μs

(b) "HD–320–16"

	Processes					
Kernel: Scheduler	**1**	**2**	**4**	**8**	**16**	**32**
Original: CFQ	221 μs	234 μs	232 μs	233 μs	237 μs	247 μs
DADS: *CFQ–Deadline*	254 μs	260 μs	239 μs	239 μs	243 μs	253 μs
DADS: *CFQ–Noop*	257 μs	269 μs	240 μs	239 μs	243 μs	252 μs

(c) "HD–500–16"

	Processes					
Kernel: Scheduler	**1**	**2**	**4**	**8**	**16**	**32**
Original: CFQ	155 μs	168 μs	170 μs	174 μs	180 μs	187 μs
DADS: *CFQ–Deadline*	155 μs	169 μs	188 μs	179 μs	183 μs	190 μs
DADS: *CFQ–Noop*	156 μs	169 μs	181 μs	179 μs	186 μs	192 μs

(d) "HD–500–32"

	Processes					
Kernel: Scheduler	**1**	**2**	**4**	**8**	**16**	**32**
Original: CFQ	159 μs	200 μs	200 μs	184 μs	183 μs	187 μs
DADS: *CFQ–Deadline*	159 μs	203 μs	202 μs	187 μs	186 μs	190 μs
DADS: *CFQ–Noop*	158 μs	202 μs	201 μs	186 μs	186 μs	189 μs

computer, we have checked that these time differences are not due to the computer itself. We have run the tests in **Hecate** but activating only 1 GB of RAM. The results obtained show that the problem depends on the disk model and not on the features of the computer.

4.6.2. SSD drives

Figure 4.9 shows the experimental results for the two SSD disks and schedulers AS and Deadline, i.e., configurations *AS–Deadline* and *Deadline–AS* of the DADS kernel are compared with the results of both schedulers on the original kernel. Experimental results for AS and Noop are shown in Figure 4.10. Figure 4.11 presents the results for the schedulers CFQ and Deadline, and Figure 4.12 depicts the results for CFQ and Noop.

Before explaining the obtained results, two key aspects given in Section 3.6.4 should be remembered. Firstly, for SSD devices, the Noop and Deadline schedulers usually outperform CFQ and AS [102, 105, 106, 107], although the I/O times achieved by all of them are almost identical. The problem with CFQ and AS is that these schedulers introduce delays with the hope of minimizing the seek time, and theses delays increase the application time.

Secondly, although the in–kernel disk simulator and DADS introduces a rather small overhead, for SSD disks, this overhead is more noticeable due to the very high performance offered by these devices. Consequently, application time is slightly increased. We have also realized that the overhead depends on the number of requests per seconds issued by the benchmarks, because the instances of the simulator have to process more requests in the same amount of time.

In order to analyze the obtained results, we start with the global execution of the test, and then, with each individual benchmark.

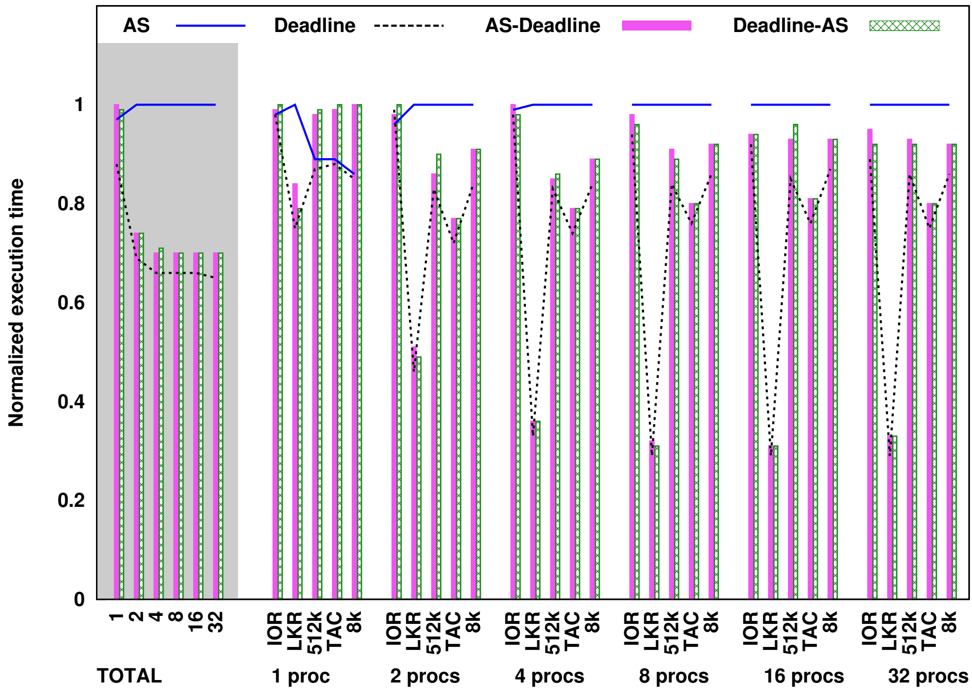
TOTAL

The first histogram in the figures shows the global behavior of DADS; it depicts the results for the overall application time of the test. As we can see, DADS follows the best scheduler, changing the scheduler, if necessary, when the number of processes also changes. Another interesting aspect is that our approach behaves the same for both disks, and achieves a quite similar performance.

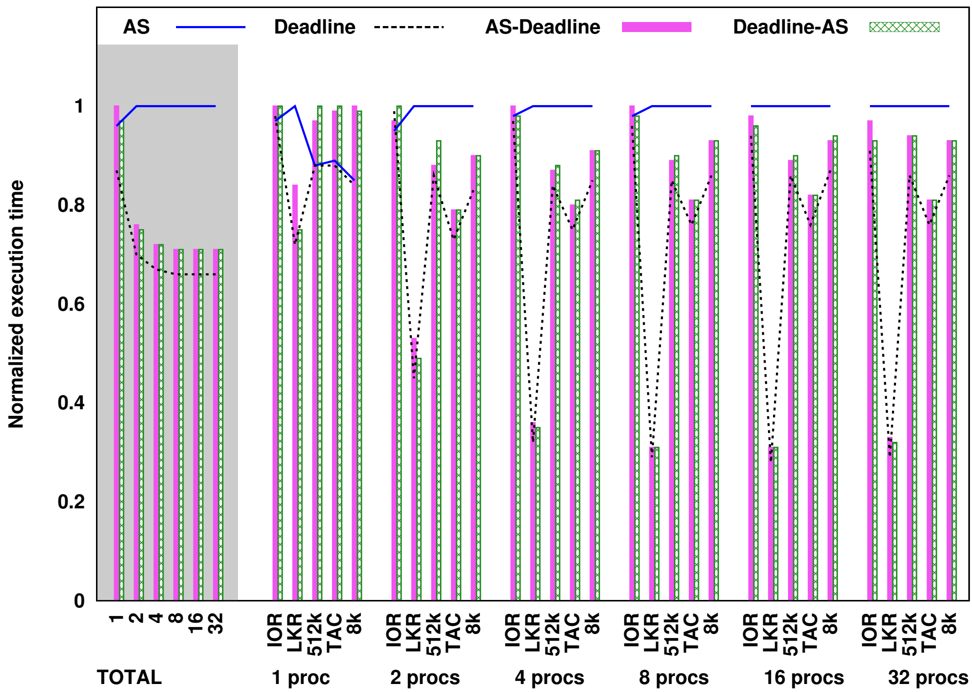
It is worth emphasizing that, for 1 process, although it seems that DADS does not choose the best scheduler, it does. Noop or Deadline are used during all the execution of the test, or after the first check DADS makes. The problem is that, for 1 process, the overhead introduced by our mechanism is more noticeable, due to the small application times achieved by the SSD disks.

IOR Read

For this benchmark, in the original kernel, the application time achieved by the four schedulers is quite similar. The highest difference between CFQ and Noop is less than 5% (between CFQ and Deadline is also 5%), and between AS and Noop is less than 12% (between AS and Deadline is 12.5%). Furthermore application times are quite small. Therefore, sometimes, our technique is not able to decide which scheduler achieves the greatest performance, and spends all the time with the initial scheduler. Note that, the mechanism to avoid a high rate

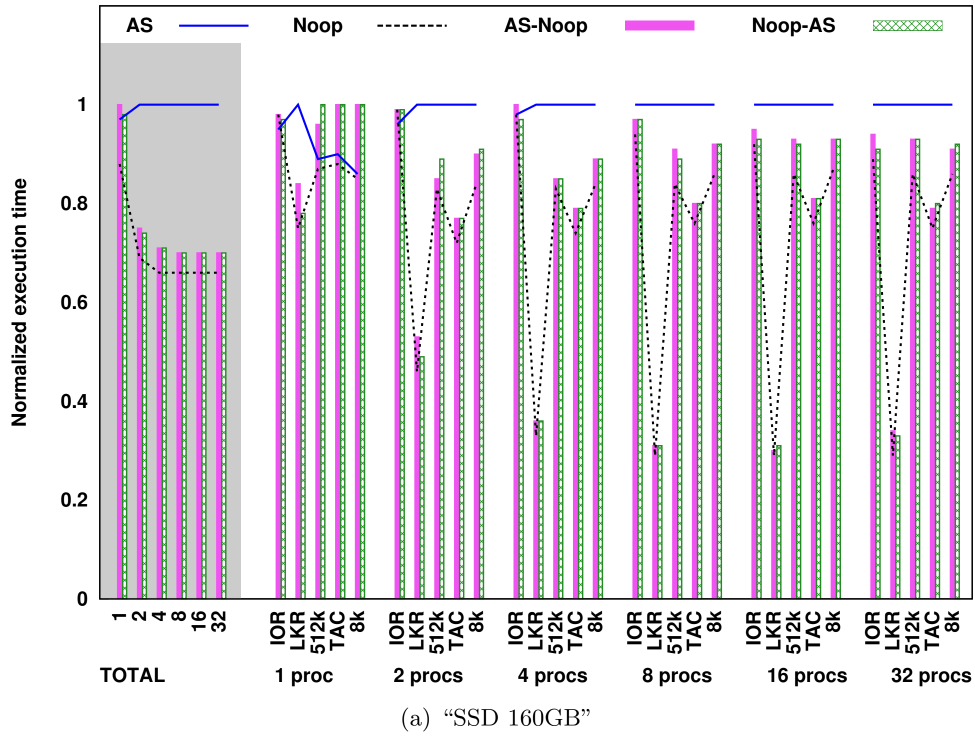


(a) "SSD 160GB"

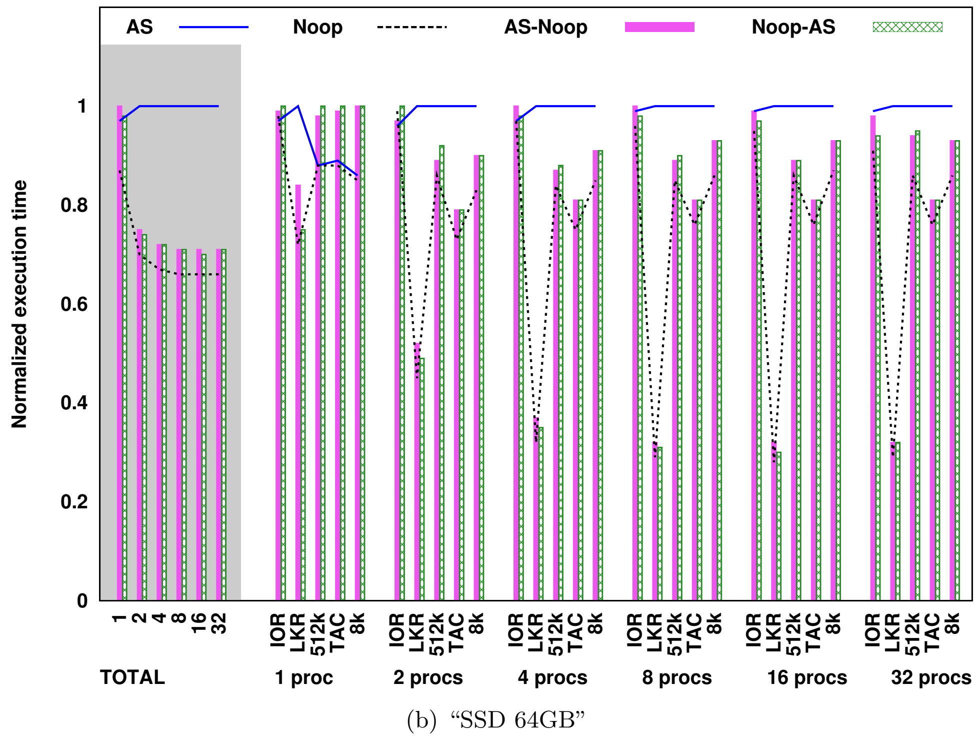


(b) "SSD 64GB"

Figure 4.9: Configurations *AS–Deadline* and *Deadline–AS* for SSD disks.

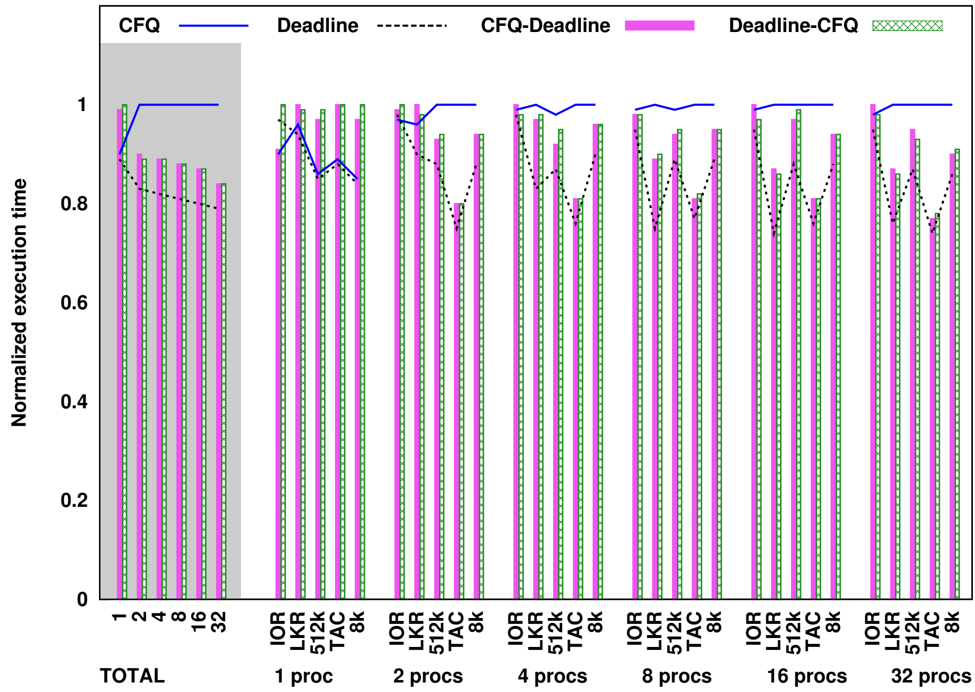


(a) "SSD 160GB"

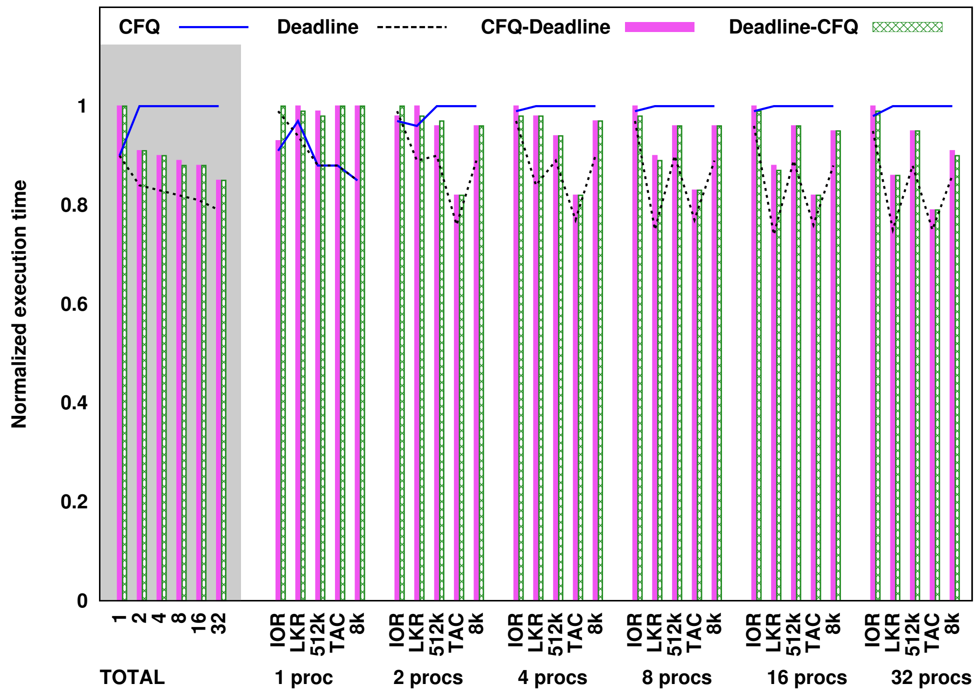


(b) "SSD 64GB"

Figure 4.10: Configurations *AS–Noop* and *Noop–AS* for SSD disks.

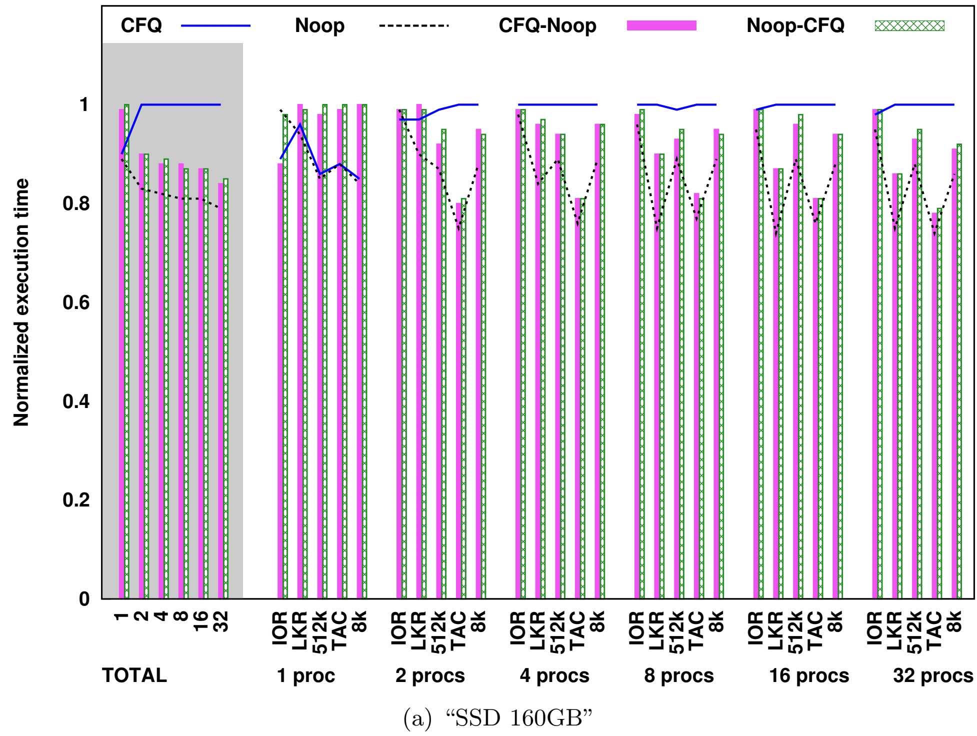


(a) "SSD 160GB"

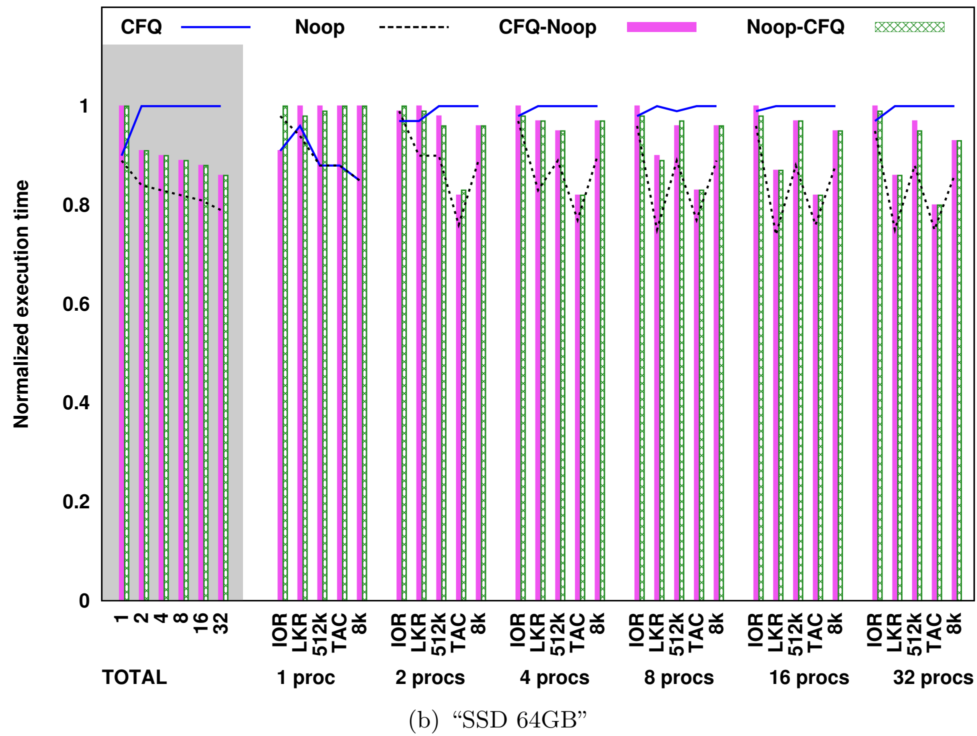


(b) "SSD 64GB"

Figure 4.11: Configurations *CFQ–Deadline* and *Deadline–CFQ* for SSD disks.



(a) "SSD 160GB"



(b) "SSD 64GB"

Figure 4.12: Configurations *CFQ–Noop* and *Noop–CFQ* for SSD disks.

of changes is not activated in this case, since DADS does not perform many changes, and it just keeps one of the schedulers most of the time.

An interesting point is that the overhead introduced by DADS is less noticeable with the IOR benchmark than with the other benchmarks. The reason is due to the fact that the number of requests per second issued by IOR is less than the number issued by the other benchmarks, because the IOR requests are larger and take longer to be served.

Linux Kernel Read

In the *Linux Kernel Read* benchmark, our approach adapts itself to the scheduler that achieves the highest throughput (Noop or Deadline, in each case), and spends all the time with it. Note that, if the best scheduler has not been selected during the previous test (IOR), the change is done in the first check. Therefore, DADS kernel presents, with all the configurations, the same behavior as the best scheduler in the original kernel.

However, the problem with this test is that DADS introduces the highest overhead, compared to the other benchmarks. In this case, the overhead is due not only to the number of requests per second issued, but also to the creation of the large amount of small processes to read the Linux kernel source. For each new process, the control structures that the disk simulator needs have to be created, and, although this creation process is optimized, it implies an increase in the application time. Remember that LKR creates a new process for every matching file.

512 kB Strided Read, TAC, and 8 kB Strided Read

Finally, for the 512k–SR, TAC, and 8k–SR benchmarks, DADS presents a quite similar behavior, and achieves the same performance as the original kernel. Our proposal has selected the best scheduler in the LKR test, and it does not perform any change during the execution of these benchmarks.

Regarding the overhead, in these tests, the number of requests per second is roughly the same, and is one of the highest. Therefore, the overhead is quite similar in each case, and is larger than that introduced during the IOR execution. Realize that, for 512k–SR and 8k–SR, the overhead is more noticeable than for TAC because, in the original kernel, the application time achieved is smaller in the latter than in the formers.

4.7. Related Work

The idea of self–tuning systems is not new, and has been extensively described in the literature. Reiner and Pinkerton [129] present a system–independent method for dynamic modification of operating system control parameters to improve system performance. Cowan *et al.* [130] describe a mechanism to support the concurrent execution and replacement of functions in an operating system to facilitate its reconfiguration. Denys *et al.* [131] present a survey and taxonomy of the existing approaches to achieve customizability in some proposed and tested operating systems.

Seltzer and Small [132] propose a self–monitoring, self–adapting, and extensible operating system (VINO). VINO monitors resource utilization and applications, and captures a significant quantity of data about the performance of the system. It uses *in situ* simulation to

compare the efficacy of policy changes, so it may adapt to changes in the workload. They also present adaptation heuristics to decrease the latency of an application. Regarding the I/O part, system modifications are made to alleviate disk waiting. Unlike our proposal, automatic adaptation has only been explored, but not implemented, in VINO. DADS has been implemented and tested inside the Linux kernel, and although we only consider the I/O disk scheduler, our approach is also both self–monitoring and self–adapting. An initial version of our disk simulator has been successfully used in REDCAP too (see Chapter 3).

Teller and Seelam [133] also present a project which addresses the general problem of providing dynamic operating system adaptations for enhanced performance. They focus on adapting operating system policies, algorithms, and parameters in order to customize the execution environment to achieve the "best possible" performance. However, they do not provide an implementation of such a general system adaptation mechanism, but a small implementation of the concept applied to a specific part of the I/O subsystem and called ADIO [134].

ADIO is an Automatic and Dynamic I/O scheduler selection algorithm that chooses between Deadline and CFQ schedulers. The algorithm selects Deadline when there are two or more requests with expired deadlines. But, when all the request latencies are bellow their deadline bounds, it chooses CFQ to optimize throughput. This approach has several shortcomings. For example, when there are no expired deadlines, CFQ is always used, while Deadline could provide a better performance for the current workload and disk. Moreover, for a large number of processes, Deadline will be always active because most of the requests will exceed their deadline bounds, while CFQ could improve the disk performance in this case. Indeed, as our experimental results show (see Section 4.6), when there are many processes, Deadline can downgrade the I/O performance without guaranteeing the deadline limits. On the other hand, since DADS optimizes service time, it always selects the scheduler that has the best performance. Furthermore, their approach only selects among CFQ or Deadline, and, although it can be modified to use AS or Noop, the comparison always has to be done with Deadline. However, in our method, the system administrator can choose any two I/O schedulers to compare.

Martens and Katchabaw propose IOAZ, a method that automatically determines the best I/O scheduler for the current workload [135]. They collect real–time disk access information, and analyze collected data. This analysis first identifies patterns in disk accesses to determine the dominant characteristics of the workload, and then decides the best disk scheduling algorithm to serve this workload. IOAZ is able to select among the four I/O schedulers available in Linux. Although a workload is defined by a wide variety of characteristics, they only use the number of processes making disk requests, the request sequentiality, and the operation type (read or write). They focus on quite simple access patterns, and, therefore, their technique could not determine the best scheduler for workloads that are neither sequential nor random since IOAZ is not able to identifies the corresponding access pattern. DADS, however, can always choose the scheduler that achieves the highest performance because it does not use access pattern information to make a decision.

Another option regarding I/O scheduling is the Two–layers Learning Scheme, proposed by Zhan and Bhargava [136]. This proposal automates the scheduling policy selection by combining workload–level and request–level learning algorithms and using machine learning techniques. Their system can choose one of five I/O schedulers, the four available in Linux

and its own implementation of the Shortest Seek Time First algorithm. The proposed scheme has three phases: a training phase; a decision phase in which a change scheduler can be performed after a classification of both the workload and incoming requests, and mapping the classification result into the best scheduling policy; and, finally, a feedback phase. Note that their proposal needs a training phase in which, firstly, they train the per–request decision scheduler, and, afterward, they train the scheduling scheme. Although they do not give the duration of the training phase, they affirm that their best results are obtained with both off–line and on–line training, and the on–line training needs to run during 24 hours to produce good estimations. However, the off–line initialization of our time tables only took 100 minutes. Moreover, for a given workload, its mechanism could not choose the best scheduler because it has not learned any information about the access pattern. But, since DADS takes a decision based on the current service times and not in a classification of the workloads or requests, this problem does not arise, and it can always select the scheduler that achieves the best performance.

Several authors present the idea that no one scheduler can provide the best possible I/O performance, and introduce algorithms that manage different policies. For instance, Teorey and Pinkerton [137] propose a two–policy algorithm as a modification of the SCAN scheduler. At low disk utilization, the algorithm uses the LOOK policy; and at a high utilization, the C–LOOK policy is used.

Shenoy and Vin state that different applications, i.e., different workloads, need different disk scheduling algorithms, and present a disk scheduling framework, called Cello, for meeting the diverse service requirements of applications [114]. Cello is a two–level disk scheduling architecture, consisting of a class–independent scheduler and a set of class–specific schedulers, that allocates disk bandwidth at two time–scales: applications and requests. The class–independent scheduler determines when and how many requests from each application class should be inserted into the scheduler queue, whereas the class–specific schedulers sort the requests in accordance with the application requirements. The two levels of the architecture separate application-independent mechanisms from application-specific scheduling policies, and thereby facilitate the co-existence of multiple class-specific schedulers. Lund and Goebel present a similar approach for multimedia databases [138]. However, although both proposals use two schedulers, they establish which one is used for each request, and do not perform dynamic scheduling. Our approach, in contrast to Cello, dynamically selects the scheduler that achieves the highest performance for any request and workload.

Regarding SSD devices, several researchers have proposed specific I/O schedulers for these devices [105, 106, 102], but, to the best of our knowledge, none about dynamic I/O scheduling. Kim *et.al* [105] improve write performance of SSD devices with a new I/O scheduler that bundles several write requests into a single logical block, while schedules reads operations independently. Dunn and Reddy [106] propose an I/O scheduler which orders write requests more efficiently, and always serves requests that are in the same block as the previous request. Finally, Kang *et al.* [102] present a modification of the I/O scheduler introduced by Kim *et.al* [105]. They propose to also categorize requests into selective groups, and to prevent starvation by setting timers for each request. If new I/O schedulers, providing different performances for different workloads, were identified for SSDs, our approach could help to dynamically select the best one.

4.8. Conclusions

In this chapter, we have presented DADS, a framework that automatically and dynamically selects the best Linux I/O scheduler for a given workload by comparing the performance achieved by each available scheduler. The implementation discussed here compares two Linux I/O schedulers, although it can easily be improved to support more schedulers. Our proposal runs, inside the Linux kernel, two instances of a disk simulator to evaluate the expected performance of each scheduler. DADS compares the total service times of each simulation, and decides a change of I/O scheduler when the performance of the real disk is expected to improve.

To implement DADS and to make a fair comparison, an enhanced and much–modified version of our previous in–kernel disk simulator, presented in Chapter 3, has been developed. Among the modifications, the new version: (a) simulates the interarrival times of the requests and the "real" I/O behavior of the processes by considering their thinking times; and (b) simulates disk caches because the disk simulators have different I/O schedulers and the cache hit ratio produced by an I/O scheduler greatly determine disk performance.

The behavior of our proposal has been analyzed by using different workloads, four different hard disks, two SSD disks, both fresh and aged Ext3 file systems, and the four Linux I/O schedulers. Results show that our mechanism selects the best scheduler of the two compared at each moment, and, therefore, improves the I/O performance.

For hard disks, when considering the total time of the test, DADS even outperforms a "normal system" in several cases. Furthermore, the overhead introduced by DADS is rather small, and only in one case it produces a slight increase in the application time with respect to the best scheduler.

For SSD disks, DADS follows the scheduler that presents the best throughput, changing the scheduler when it is necessary. The problem here is that the overhead introduced by the simulation is more noticeable, due to the very high performance offered by these devices, so the application time is slightly increased.

DADS has some important features: i) it can be used in any disk because the in–kernel disk simulator is able to simulate *any disk* [35]; ii) it does not usually interfere with regular I/O requests because simulation and comparison are performed out of the I/O path; and iii) it is able to compare any I/O scheduler and select the best one at any time, being the best the scheduler that achieves the lowest service time.

To sum up, we can claim that, by using DADS, the performance achieved is always close to the best one, and system administrators are exempted from selecting a suboptimal I/O scheduler which can provide a good performance for some workloads, but may downgrade the system throughput when the workloads change.

Chapter 5

Conclusions and Future Directions

Let us finish this thesis by summarizing our findings and providing a brief outlook on further research directions. The main goal of this work has been the improvement of the I/O performance. Our motivation has been that a better I/O performance would usually enhance the overall system performance. The three mayor contributions made to achieve this aim are the following.

REDCAP: The RAM Enhanced Disk Cache Project

Firstly, we have extended the cache hierarchy by introducing a new level, the REDCAP cache, between the page and disk caches. A prefetching technique and an algorithm to control the performance achieved by the new cache round off this proposal. By using a small portion of the RAM memory, REDCAP can significantly reduce the I/O time of disk read requests, and also mitigates the problem of a premature eviction of blocks from the disk cache.

For workloads with some spatial locality, our approach improves the performance up to 80% for hard disk drives, and up to 88% for SSD drives. For workloads where an improvement in the I/O time is hard to obtain (mainly, sequential and random access patterns), it achieves identical performance to that obtained by a normal system.

Although REDCAP is able to obtain its maximum performance with all the file systems tested and any I/O scheduler, improvements achieved slightly depend on the file system and scheduler used. The problem is that the file system and scheduling policy determine the I/O performance of a disk drive to a large extent.

Since REDCAP emulates the behavior of a disk cache (and profits from its read–ahead mechanism), a conclusion that can be extracted from our results is that disk drives should include larger caches. This improvement in size would easily enhance the disk performance.

We have also shown that the prefetching has to be dynamic, and a mechanism to turn the prefetching on/off depending on the improvement achieved is absolutely necessary. Otherwise, for some workloads, aggressive prefetching (with no control) can significantly degrade the I/O performance, and consequently the system performance.

In–kernel disk simulator

Secondly, we have implemented a disk simulator inside the Linux kernel that imitates the behavior of traditional and SSD drives. We model a disk by using a dynamic table of I/O

times addressable by seek distance, request size and operation type (read or write). The dynamic approach allows the disk model to adapt to changes in the workload.

The initial version of our disk simulator controls the request arrival order and the dependencies among requests of the same process or related processes. It also has an I/O scheduler to establish the dispatch order. The second version takes into account thinking times of requests, and simulates a disk cache too.

The accuracy analysis performed states that, for hard disk drives, our model presents a good behavior by matching the real disk in a precise way, and that differences between the real and virtual disks are due to the difficulty in simulating a disk cache. For SSD drives, I/O time estimations provided by our model have a higher precision, and differences between both disks are, on average, smaller than 0.3%.

The proposed simulator can be used for an on–line simulation of the performance obtained by different system mechanisms and algorithms, and for dynamically turning them on and off, or selecting between different configurations or policies, accordingly. The first version of our simulator has been successfully used for improving and simplify the activation–deactivation algorithm of REDCAP. Note that, by using the virtual disk, REDCAP always obtains the maximum possible improvements. The second version has been used in DADS to implement a dynamic scheduling system.

Therefore, we can conclude that, unlike other theoretical approaches, our disk simulator makes a real self–monitoring and self–adapting I/O subsystem come true.

DADS: Dynamic and Automatic Disk Scheduling framework

Our third and last contribution is DADS, a mechanism that makes a real–time comparison between two different Linux I/O schedulers, and dynamically chooses the scheduler that achieves the best performance for the current workload. We simultaneously run an instance of our disk simulator for each scheduler to compare, and the I/O scheduler selected is the one whose simulation provides the lowest service time for the same amount of requested data.

DADS adapts itself to the best scheduler at anytime, and, for hard disks, it can even outperform a “normal system”, because it changes the scheduler to achieve the highest throughput in each moment. For SSD disks, our approach always follows the I/O scheduler that presents the best throughput too. However, due to the very high performance offered by these devices, the overhead introduced by the simulation is noticeable, and, as a consequence, the application time is slightly increased.

Our study confirms that there is no I/O scheduler that always provides the best possible I/O performance, since the result depends on several factors (workloads, disk drives, and so on). Thus, a mechanism like DADS, that is able to choose the best scheduler at a given moment, is necessary. Indeed, by using DADS, system administrators are exempted from selecting a suboptimal I/O scheduler which can provide a good performance for some workloads, but may downgrade the system throughput when the workloads change.

Future Work

The work presented in this dissertation can be extended in many directions since several interesting research paths remain open. The following are just a few.

Two different issues of REDCAP deserve some analysis. Firstly, its activation–deactivation algorithm could be enhanced to control the performance obtained by different parts of a disk, and not by the whole disk. Currently, REDCAP prefetching is globally enabled or disabled for a given disk. However, since several processes can access the disk concurrently, the resultant disk workload may be a blend of different access patterns. Therefore, it is possible that, in some parts of the disk, REDCAP is improving the throughput, whereas, in other parts, it is not. The new algorithm could analyze the performance by groups of disk segments[1]. Given a request, it could decide whether to prefetch data or not depending on the improvement already achieved for the corresponding group and adjacent groups. REDCAP would thereby be active/inactive by parts of the disk.

Secondly, since REDCAP is able to obtain significant improvements for traditional and SSD drives, it is worth investigating the deployment of REDCAP to RAID systems. In this case, a new metric to measure the performance achieved would be needed because a RAID system looks like a single disk, although the controller is internally managing a group of disks.

We have modeled the cache of our disk simulator in a simple way, because we have only considered a fixed number of segments. However, most disk caches have a dynamic behavior, and modify the number of segments to improve the cache–hit rates [49]. We should investigate how our simulator could catch such dynamic behavior, and calculate the number of segments of the cache by analyzing request patterns and request I/O times.

In this work, DADS only compares two by two the Linux I/O schedulers. We also want to extend DADS to compare all the available schedulers simultaneously. Furthermore, an interesting aspect is that all schedulers, except Noop, have several tunable parameters that can be modified to ensure optimal performance. But, tuning schedulers manually to get the best I/O performance is not a straightforward task. Therefore, it would be a good idea for DADS to select not only the best scheduler, but also the best values for the corresponding parameters.

A pretty natural extension of the present work would be the evaluation of our proposals on hybrid hard disk drives. Firstly, we should test whether our in–kernel disk simulator is able to simulate the behavior of these devices. Secondly, we should test the performance that REDCAP could achieve for H–HDDs. Finally, the I/O performance of each scheduler should be evaluated under different workloads, and we should then analyze whether DADS could help to dynamically select the best scheduler.

Finally, another very promising direction of research is the design and implementation of new mechanisms, based on our disk simulator, to improve I/O performance. I/O schedulers are a good candidate. Since our virtual disk also simulates the disk cache of a real disk, it is feasible to implement new I/O schedulers that can take into account the simulated cache's contents to sort requests in the real disk. For instance, the scheduler could serve a request "out–of–order" if the request is to be immediately evicted from the disk cache.

[1] Remember that REDCAP considers the disk as a contiguous sequence of blocks, and splits the disk into segments of the same size as the REDCAP segments.

Bibliography

[1] C. Ruemmler and J. Wilkes, “An Introduction to Disk Drive Modeling,” *Computer*, vol. 27, no. 3, pp. 17–28, 1994.

[2] B. Jacob, S. W. Ng, and D. T. Wang, *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers, September 2007.

[3] J. K. Ousterhout, “Why Aren't Operating Systems Getting Faster As Fast as Hardware?” in *USENIX Summer*. USENIX Association, 1990.

[4] J. Shen and M. Lipasti, *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw–Hill, 2005.

[5] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach. Fifth Edition*. Morgan Kaufmann Publishers, September 2011.

[6] “Seagate.” [Online]. Available: http://www.seagate.com

[7] “Hitachi Deskstar 7K3000 HDS723030ALA640.” [Online]. Available: http://www.hitachigst.com/internal-drives/desktop/deskstar/deskstar-7k3000

[8] “Wester Digital.” [Online]. Available: http://www.wdc.com/en/products/internal/desktop/

[9] “Samsung DRAM Memory.” [Online]. Available: http://www.samsung.com/us/computer/memory-storage/MV-3V4G4/US

[10] “Intel® Solid-State Drive 710 Series.” [Online]. Available: http://www.intel.com/content/www/us/en/solid-state-drives/solid-state-drives-710-series.html

[11] “Samsung SSD.” [Online]. Available: http://www.samsung.com/us/computer/solid-state-drives

[12] “OCZ Vertex 2 SATA II 3.5” SSD.” [Online]. Available: http://www.ocztechnology.com/ocz-vertex-2-sata-ii-3-5-ssd.html

[13] “OCZ Octane SATA III 2.5” SSD.” [Online]. Available: http://www.ocztechnology.com/ocz-octane-sata-iii-2-5-ssd.html

[14] “Seagate Momentus® XT Solid State Hybrid Drives. Model ST750LX003.” [Online]. Available: http://www.seagate.com/www/en-us/products/laptops/laptop-hdd/

[15] “OCZ RevoDrive Hybrid PCI-Express Solid State Drive.” [Online]. Available: http://www.ocztechnology.com/ocz-revodrive-hybrid-pci-express-solid-state-drive.html

[16] Roger Wood and Yimin Hsu and Marilee Schultz, “Perpendicular Magnetic Recording Technology. White paper,” September

2006. [Online]. Available: http://www.hitachigst.com/tech/techlib.nsf/techdocs/F47BF010A4D29DFD8625716C005B7F34/$file/PMR_white_paper_final.pdf

[17] Koji Matsumoto and Akihiro Inomata and Shin–ya Hasegawa, "Thermally Assisted Magnetic Recording." June 2005. [Online]. Available: http://www.fujitsu.com/downloads/MAG/vol42-1/paper18.pdf

[18] Richard Freitas and Joseph Slember and Wayne Sawdon and Lawrence Chiu, "GPFS Scans 10 Billion Files in 43 Minutes," in *IBM Advanced Storage Laborator*. IBM Almaden Research Center. San Jose, CA 95120, 2011.

[19] "Hitachi Global Storage Technologies." [Online]. Available: http://www.hitachigst.com/

[20] E. Frauenheim, "Hitachi to unveil 400GB drive," *CNET News*, March 2004. [Online]. Available: http://news.cnet.com/Hitachi-to-unveil-400GB-drive/2100-1015_3-5171944.html

[21] "Seagate Technology Company Milestones." [Online]. Available: http://www.seagate.com/www/en-us/about/corporate_information/company_milestones

[22] A. L. Shimpi, "The World's First 3TB HDD: Seagate GoFlex Desk 3TB Review," August 2010. [Online]. Available: http://www.anandtech.com/show/3858/the-worlds-first-3tb-hdd-seagate-goflex-desk-3tb-review

[23] "Western Digital, the first to ship an internal 3TB hard drive," October 2010. [Online]. Available: http://www.htlounge.net/art/13919/western-digital-the-first-to-ship-an-internal-3tb-hard-drive.html

[24] A. L. Shimpi, "Seagate Ships World's First 4TB External HDD," July 2011. [Online]. Available: http://www.anandtech.com/show/4738/seagate-ships-worlds-first-4tb-external-hdd

[25] "Seagate FreeAgent® GoFlex™ Desk External Drive." [Online]. Available: http://www.seagate.com/www/en-us/products/external/external-hard-drive/desktop-hard-drive/

[26] "OCZ Z-Drive R2 p88 PCI-Express SSD *EOL." [Online]. Available: http://www.ocztechnology.com/ocz-z-drive-r2-p88-pci-express-ssd.html

[27] M. H. Kryder and C. S. Kim, "After Hard Drives–What Comes Next?" *IEEE TRANSACTIONS ON MAGNETICS*, vol. 45, no. 10, pp. 3406–3413, October 2009.

[28] H. Newman, "Why Solid State Drives Won't Replace Spinning Disk," July 2010. [Online]. Available: http://www.enterprisestorageforum.com/storage-technology/Why-Solid-State-Drives-Wont-Replace-Spinning-Disk.htm-3894671.htm

[29] Laura Grupp, Adrian Caulfield, Joel Coburn, Steven Swanson, Eitan Yaakobi and Paul Siegel, "Characterizing Flash Memory: Anomalies, Observations, and Applications," in *42nd Annual IEEE/ACM International Symposium on Microarchitecture*, December 2009.

[30] N. Mielke and T. Marquart and Ning Wu and J. Kessenich and H. Belgal and E. Schares and F. Trivedi and E. Goodness and L. R. Nevill, "Bit error rate in NAND Flash memories," in *IEEE International Symposium on Reliability Physics (IRPS)*, July 2008.

[31] R. Karedla, J. S. Love, and B. G. Wherry, "Caching strategies to improve disk system performance," *Computer*, vol. 27, no. 3, pp. 38–46, 1994.

[32] W. W. Hsu and A. J. Smith, "The performance impact of I/O optimizations and disk improvements," *IBM Journal of Research and Development*, vol. 48, no. 2, pp. 255–289, 2004.

[33] P. González-Férez, J. Piernas, and T. Cortés, "The RAM Enhanced Disk Cache Project (REDCAP)," in *Proceedings of the IEEE Conference on Massive Storage Systems and Technologies (MSST)*, 2007.

[34] P. González-Férez, J. Piernas, and T. Cortés, "Evaluating the Effectiveness of REDCAP to Recover the Locality Missed by Today's Linux Systems," in *Proceedings Annual Meeting of the IEEE/ACM International Symposium on Modeling, Analysis, and Simulation Computer and Telecommunication Systems (MASCOTS)*, 2008.

[35] P. González-Férez, J. Piernas, and T. Cortés, "Simultaneous evaluation of multiple I/O strategies," in *Proceedings of the 22nd International Symposium on Computer Architecture and High Performance Computing*, 2010.

[36] S. L. Pratt and D. A. Heger, "Workload Dependent Performance Evaluation of the Linux 2.6 I/O Schedulers," in *Linux Symposium*, July 2004.

[37] P. González-Férez, J. Piernas, and T. Cortés, "DADS: Dynamic and Automatic Disk Scheduling," in *Proceeding of the 27th Symposium On Applied Computing*, March 2012.

[38] "Western Digital." [Online]. Available: http://www.wdc.com

[39] S. Bhatia, E. Varki, and A. Merchant, "Sequential prefetch cache sizing for maximal hit rate," in *International Symposium on Modeling, Analysis, and Simulation of Computer Systems*. IEEE Computer Society, 2010.

[40] "Seagate Barracuda XT 6GB/s 2TB Hard Drive ST320005N1A1AS-RK." [Online]. Available: http://www.seagate.com/www/en-us/products/internal-storage/barracuda-xt-kit#tTabContentSpecifications

[41] S. Tweedie, "Journaling the Linux ext2fs Filesystem," in *LinuxExpo'98*, 1998.

[42] R. Love, *Linux Kernel Development. 2nd edition.* Novell, 2005.

[43] "IOR Benchmark." [Online]. Available: http://ior-sio.sourceforge.net

[44] "Coreutils." [Online]. Available: http://www.gnu.org/software/coreutils/

[45] B. L. Worthington, G. R. Ganger, Y. N. Patt, and J. Wilkes, "On-Line Extraction of SCSI DISK Drive Parameters," Hewlett-Packerd Laboratories, Tech. Rep., 1997.

[46] F. I. Popovici, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Robust, Portable I/O Scheduling with the Disk Mimic," in *Proceedings of the USENIX Annual Technical Conference*, 2003.

[47] S. Iyer and P. Druschel, "Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O," in *Symposium on Operating Systems Principles*, 2001.

[48] "Samsung." [Online]. Available: http://www.samsung.com/global/business/hdd/

[49] E. Shriver, "Performance modeling for realistic storage devices," Ph.D. dissertation, May 1997.

[50] J. Schindler and G. R. Ganger, "Automated Disk Drive Characterization," Tech. Rep., December 1999.

[51] "The Linux Kernel Archives." [Online]. Available: http://www.kernel.org/

[52] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel, 3rd edition*. O'Really.

[53] H. Lei and D. Duchamp, "An Analytical Approach to File Prefetching," in *Proceedings of the USENIX Annual Technical Conference*, 1997.

[54] T. M. Wong and J. Wilkes, "My cache or yours? Making storage more exclusive," in *Proceedings of the USENIX Annual Technical Conference*, 2002.

[55] P. González-Férez, J. Piernas, and T. Cortés, "The RAM Enhanced Disk Cache Project (REDCAP)," Dpto. de Ingeniería y Tecnología de Computadores. Universidad de Murcia, Tech. Rep. TR-DITEC-UM-0002-2007, December 2007.

[56] E. Shriver, A. Merchant, and J. Wilkes, "An analytic behavior model for disk drives with readahead caches and request reordering," in *Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, ACM. ACM Press, 1998, pp. 182–191.

[57] C. R. Lumb, J. Schindler, and G. R. Ganger, "Freeblock Scheduling Outside of Disk Firmware," in *Proceeding of the 1st USENIX Conference on File and Storage Technologies (FAST'02)*, USENIX, Ed. USENIX, January 2002.

[58] C. R. Lumb, J. Schindler, G. R. Ganger, and D. F. Nagle, "Towards Higher Disk Head Utilization: Extracting Free Bandwidth From Busy Disk Drives," in *Proccedings of the 4th Symposium on Operating Systems Design and Implementation*, October 2000.

[59] "TPCC-UVa." [Online]. Available: http://www.infor.uva.es/~{}diego/tpcc-uva.html

[60] R. Card, T. Ts'o, and S. Tweedie, "Design and Implementation of the Second Extended Filesystem," *In Proceedings of the First Dutch International Symposium on Linux*, 1994.

[61] "JFS for Linux," http://jfs.sourceforge.net/, 2008. [Online]. Available: http://jfs.sourceforge.net/

[62] "ReiserFS," http://www.namesys.com, 2008. [Online]. Available: http://www.namesys.com

[63] "Linux XFS," http://oss.sgi.com/projects/xfs/, 2008. [Online]. Available: http://oss.sgi.com/projects/xfs/

[64] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the XFS File System," *In Proceedings of the USENIX 1996 Annual Technical Conference.*

[65] M. McKusick, M. Joy, S. Leffler, and R. Fabry, "A fast file system for UNIX," *ACM Transactions on Computer Systems*, vol. 2, no. 3, pp. 181–197, August 1984.

[66] "e2fsprogs." [Online]. Available: http://e2fsprogs.sourceforge.net/

[67] M. V. Wilkes, "Slave Memories and Dynamic Storage Allocation," *IEEE Tran. on Electronic Com.*, pp. 270–271, 1965.

[68] Y. Hu and Q. Yang, "DCD Disk Caching Disk: A New Approach for Boosting I/O Performance," in *Proceeding of the 23rd Annual International Symposium on Computer Architecture, (ISCA'96)*, 1996.

[69] Y. Hu and Q. Yang, "A New Hierarchical Disk Architecture," in *IEEE Micro*, 1998, vol. 18, no. 6, pp. 64–76.

[70] Y. Hu, T. Nightingale, and Q. Yang, "Rapid-cache-a reliable and inexpensive write cache for high performance storage systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 3, pp. 290–307, March 2002.

[71] B. L. Worthington, G. R. Ganger, and Y. N. Patt, "Scheduling Algorithms for Modern Disk Drives," in *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems.*

[72] M. Seltzer, P. Chen, and J. Ousterhout, "Disk Scheduling Revisited," in *Proceedings of the USENIX Winter 1990 Technical Conference.* USENIX Association, 1990.

[73] D. M. Jacobson and J. Wilkes, "Disk Scheduling algorithms based on rotational position," Hewlett-Packerd Laboratories, Tech. Rep., February 1991.

[74] B. L. Worthington, "Aggressive Centralized and Distributed Scheduling of Disk Requests," Ph.D. dissertation, 1996.

[75] G. R. Ganger, "System-Oriented Evaluation of I/O Subsystem Performance," Ph.D. dissertation, 1995.

[76] H.-P. Chang, R.-I. Chang, W.-K. Shih, and R.-C. Chang, "Real-Time Disk Scheduling with On-Disk Cache Conscious," in *Real-Time and Embedded Computing Systems and Applications.* Springer Berlin / Heidelberg, 2003, pp. 88–102.

[77] E. V. Carrera and R. Bianchini, "Improving Disk Throughput in Data-Intensive Servers," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, 2004.

[78] K. S. Grimsrud, J. K. Archibald, and B. E. Nelson, "Multiple Prefetch Adaptive Disk Caching," *IEEE Trans. on Knowl. and Data Eng.*, vol. 5, no. 1, pp. 88–103, February 1993.

[79] Q. Zhu, E. Gelenbe, and Y. Qiao, "Adaptive prefetching algorithm in disk controllers," *Performance Evaluation*, vol. 65, no. 5, pp. 382–395, 2008.

[80] P. Cao, E. W. Felten, A. R. Karlin, and K. Li, "Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling," *ACM Transactions on Computer Systems (TOCS)*, vol. 14, no. 4, pp. 311–343, 1996.

[81] C. Zhifeng, Z. Yuanyuan, and L. Kai, "Eviction Based Cache Placement for Storage Caches," in *USENIX 2003 Annual Technical Conference, General Track*.

[82] G. Yadgar and M. Factor, "Karma: Know-it All Replacement for a Multilevel cAche," in *Proceeding of the 5th USENIX Conference on File and Storage Technologies (FAST'07)*, USENIX, Ed. USENIX, 2007.

[83] V. Soloviev, "Prefetching in Segmented Disk Cache for Multi-Disk Systems," in *Proceedings of the fourth workshop on I/O in parallel and distributed systems: part of the federated computing research conference*, ACM. ACM Press, 1996, pp. 69–82.

[84] Y. Zhu and Y. Hu, "Disk Built-in Caches: Evaluation on System Performance," in *11th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'03)*, 2003.

[85] Seetharami Seelam and I-Hsin Chung and Ding-Yong Hong and Hui-Fang Wen and Hao Yu, "Early experiences in application level I/O tracing on blue gene systems," in *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE Computer Society, April 2008.

[86] Seetharami Seelam and I-Hsin Chung and John Bauer and Hao Yu and Hui-Fang Wen, "Application level I/O caching on Blue Gene/P systems," in *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE Computer Society, May 2009.

[87] Seetharami Seelam and I-Hsin Chung and John Bauer and Hui-Fang Wen, "Masking I/O latency using application level I/O caching and prefetching on Blue Gene systems." IEEE Computer Society, April 2010.

[88] A. J. Smith, "Disk cache–miss ratio analysis and design considerations," *ACM Transactions on Computer Systems (TOCS)*, vol. 3, no. 3, pp. 161–203, 1985.

[89] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman, *Linux Device Drivers, Third Edition*. O'Reilly, 2005.

[90] M. Aboutabl, A. Agrawala, and J.-D. Decotignie, "Temporally Determinate Disk Access: An Experimental Approach," in *Measurement and Modeling of Computer Systems*, 1998, pp. 280–281.

[91] N. Talagala, R. H. Arpaci-Dusseau, and D. Patterson, "Microbenchmark-based Extraction of Local and Global Disk Characteristics," Tech. Rep., 1999.

[92] Z. Dimitrijevic, R. Rangaswami, E. Chang, D. Watson, and A. Acharya, "Diskbench: User-level Disk Feature Extraction Tool," Universidad de California, Tech. Rep., June 2004.

[93] J. hong Kim, D. Jung, J. soo Kim, and J. Huh, "A Methodology for Extracting Performance Parameters in Solid State Disks (SSDs)," in *17th IEEE International Symposium*

on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'09), 2009.

[94] Jose David Prieto Pagán, "Desarrollo de una herramienta para la extracción de caracteríticas de discos duros," Universidad de Murcia, Proyecto fin de carrera, July 2007, (in Spanish).

[95] Francisco Javier Roca Alcaráz and Juan Sánchez Segura, "Simulación de disco y extracción eficiente de caracteríticas," Universidad de Murcia, Proyecto fin de carrera, September 2008, (in Spanish).

[96] J. Axboe, "block: remove the anticipatory IO scheduler," October 2009. [Online]. Available: http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commitdiff;h=492af6350a5ccf087e4964104a276ed358811458

[97] R. Love, *Linux System Programming.* O'Reilly Media, September 2007.

[98] "Linux Kernel Architecture." [Online]. Available: http://www.mercenarylinux.com/kernel-architecture/

[99] G. Rodrigues, "Variations on fair I/O schedulers," December 2008. [Online]. Available: http://lwn.net/Articles/309400/

[100] L.-P. Chang, "A Hybrid Approach to NAND-Flash-Based Solid-State Disks," *IEEE Transactions on Computers*, vol. 59, pp. 1337–1349, 2010.

[101] F. Chen, R. Lee, and X. Zhang, "Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing," in *International Symposium on High-Performance Computer Architecture.* IEEE Computer Society, 2011.

[102] S. Kang, H. Park, and C. Yoo, "Performance enhancement of I/O scheduler for Solid State Devices," in *IEEE International Conference on Consumer Electronics (ICCE)*, 2011.

[103] P. Schmid and A. Roos, "Inside the X25-M Controller: Wear Leveling, Write Amplification Control," 2008. [Online]. Available: http://www.tomshardware.com/reviews/Intel-x25-m-SSD,2012-5.html

[104] "Intel." [Online]. Available: http://www.intel.com

[105] J. Kim, Y. Oh, E. Kim, J. Choi, D. Lee, and S. H. Noh, "Disk Schedulers for Solid State Drivers," in *Proceedings of the 7th ACM international conference on Embedded software*, 2009.

[106] M. Dunn and A. L. N. Reddy, "A new I/O scheduler for solid state devices," Department of Electrical and Computer Engineering Texas A&M University, Tech. Rep., April 2009. [Online]. Available: http://dropzone.tamu.edu/TechReports

[107] Saxena, Mohit and Swift, Michael M., "FlashVM: virtual memory management on flash," in *Proceedings of the 2010 USENIX conference on USENIX Annual Technical Conference.* USENIX Association, 2010, pp. 14–14.

[108] “Findutils.” [Online]. Available: http://www.gnu.org/s/findutils/

[109] C. L. Elford and D. A. Reed, “Technology trends and disk array performance,” *Parallel and Distributed Computing*, vol. 46, no. 2, pp. 136–147, 1997.

[110] R. Y. Wang, T. E. Anderson, and D. A. Patterson, “Virtual log based file systems for a programmable disk,” in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, 1999.

[111] S. Jiang, X. Ding, F. Chen, E. Tan, and X. Zhang, “DULO: An effective buffer cache management scheme to exploit both temporal and spatial locality,” in *Proceeding of the 4th USENIX Conference on File and Storage Technologies (FAST'05)*, USENIX, Ed. USENIX, 2005.

[112] X. Yu, B. Gum, Y. Chen, R. Y. Wang, K. Li, A. Krishnamurthy, and T. E. Anderson, “Trading Capacity for Performance in Disk Array,” in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, 2000.

[113] “The DiskSim Simulation Environment.” [Online]. Available: http://www.pdl.cmu.edu/DiskSim/

[114] P. J. Shenoy and H. M. Vin, “Cello: A Disk Scheduling Framework for Next Generation Operating Systems,” *ACM SIGMETRICS Performance Evaluation Review*, vol. 26, no. 1, pp. 44–55, January 1998.

[115] Z. Dimitrijevic, R. Rangaswami, and E. Chang, “Design and Implementation of Semi-preemtible IO,” in *Proceeding of the 2nd USENIX Conference on File and Storage Technologies (FAST'03)*. USENIX, March 2003.

[116] J. Wilkes, “The Pantheon storage–sytem simulator,” Hewlett-Packard Company, Tech. Rep., May 1996.

[117] “Microsoft Research.” [Online]. Available: http://research.microsoft.com/en-us/

[118] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, “Design tradeoffs for SSD performance,” in *USENIX 2008 Annual Technical Conference on Annual Technical Conference*. USENIX Association, 2008.

[119] J. Lee, E. Byun, H. Park, J. Choi, D. Lee, and S. H. Noh, “CPS-SIM: configurable and accurate clock precision solid state drive simulator,” in *Proceedings of the 2009 ACM Symposium on Applied Computing*. ACM, 2009.

[120] Y. Kim, B. Tauras, A. Gupta, D. Mihai, and N. B. Urgaonkar, “FlashSim: A Simulator for NAND Flash-based Solid-State Drives,” Department of Computer Science and Engineering, The Pennsylvania State University, Tech. Rep., 2009. [Online]. Available: http://csl.cse.psu.edu/?q=node/321

[121] M. Wang, K. Au, A. Ailamaki, A. Brockwell, C. Faloutsos, and G. Ganger, “Storage Device Performance Prediction with CART Models,” in *12th Annual Meeting of the IEEE/ACM International Symposium on Modeling, Analysis, and Simulation Computer and Telecommunication Systems (MASCOTS)*, 2004.

[122] Mesnier, M. P. and Wachs, M. and Sambasivan, R. R. and Zheng, A. X. and Ganger, G., "Modeling the Relative Fitness of Storage," in *Proceedings of the 2007 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 2007.

[123] C. C. Gotlieb and G. H. MacEwen, "Performance of movable-head disk storage devices," *Journal of the ACM*, vol. 20, no. 4, pp. 604–623, 1973.

[124] N. C. Thornock, X. Tu, and J. K. Flanagan, "A stochastic disk I/O simulation technique," in *Proceedings of the 29th conference on Winter Simulation*, 1997.

[125] E. Anderson, "Simple table-based modeling of storage devices," Tech. Rep., 2001.

[126] F. Chang and G. A. Gibson, "Automatic I/O hint generation through speculative execution," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*. USENIX, 1999.

[127] K. Fraser and F. Chang, "Operating System I/O Speculation: How two invocations are faster than one," in *Proceedings of the USENIX Annual Technical Conference*, 2003.

[128] "Samsung SSD MZ-5PA256." [Online]. Available: http://www.samsung.com/es/consumer/pc-peripherals-printer/memory-storage/ssd/\MZ-5PA256/EU/index.idx?pagetype=prd_detail\&tab=specification#s275_TableView

[129] D. Reiner and T. Pinkerton, "A method for adaptive performance improvement of operating systems," in *Proceedings of the 1981 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1981.

[130] C. Cowan, T. Autrey, C. Krasic, C. Pu, and J. Walpole, "Fast concurrent dynamic linking for an adaptive operating system," in *Proceedings of the 3rd International Conference on Configurable Distributed Systems*, 1996.

[131] G. Denys, F. Piessens, and F. Matthijs, "A survey of customizability in operating systems research," *ACM Computing Surveys (CSUR)*, vol. 34, no. 4, pp. 450–468, December 2002.

[132] M. Seltzer and C. Small, "Self–Monitoring and Self–Adapting Operating Systems," in *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*, 1997.

[133] P. J. Teller and S. R. Seelam, "Insights into providing dynamic adaptation of operating system policies," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 2, pp. 83–89, 2006.

[134] S. Seelam, J. S. Babu, and P. Teller, "Automatic I/O Scheduler Selection for Latency and Bandwidth Optimization," in *Proceedings of the Workshop on Operating System Interface on High Performance Applications, in conjunction with the 14th International Conferences on Parallel Architectures and Compilation Techniques (PACT05)*. IEEE y ACM, September 2005.

[135] D. L. Martens and M. J. Katchabaw, "Optimizing System Performance Through Dynamic Disk Scheduling Algorithm Selection," *WSEAS Transactions on Information Science and Applications*, vol. 3, no. 7, pp. 1361–1368, July 2006.

[136] Y. Zhang and B. Bhargava, "Self–learning disk scheduling," *EEE Transactions on Knowledge and Data Engineering*, vol. 21, no. 1, pp. 50–65, January 2009.

[137] T. J. Teorey and T. B. Pinkerton, "A Comparative Analysis of Disk Scheduling Policies," in *Communications of the ACM.* ACM Press, March 1972, vol. 15, no. 3, pp. 177–184.

[138] K. Lund and V. Goebel, "Adaptive disk scheduling in a multimedia DBMS," in *Proceedings of the eleventh ACM international conference on Multimedia*, 2003.