UNIVERSIDAD DE MURCIA

ESCUELA INTERNACIONAL DE DOCTORADO

TESIS DOCTORAL

Microarchitectural Optimizations for an Efficient Utilization of Processor Resources

Optimizaciones Microarquitecturales para un uso Eficiente de los Recursos del Procesador

D. Sawan Singh

2024

UNIVERSIDAD DE MURCIA

ESCUELA INTERNACIONAL DE DOCTORADO

TESIS DOCTORAL

Microarchitectural Optimizations for an Efficient Utilization of Processor Resources

Optimizaciones Microarquitecturales para un uso Eficiente de los Recursos del Procesador

Autor: D. Sawan Singh

Director/es: D. Alberto Ros

D.a Alexandra Jimborean

UNIVERSIDAD DE MURCIA

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD DE LA TESIS PRESENTADA EN MODALIDAD DE COMPENDIO O ARTÍCULOS PARA OBTENER EL TÍTULO DE DOCTOR

Aprobado por la Comisión General de Doctorado el 19-10-2022

D./Dña. Sawan Singh

doctorando del Programa de Doctorado en

Informatica

de la Escuela Internacional de Doctorado de la Universidad Murcia, como autor/a de la tesis presentada para la obtención del título de Doctor y titulada:

Microarchitectural Optimizations for an Efficient Utilization of Processor Resources / Optimizaciones Microarquitecturales para un uso Eficiente de los Recursos del Procesador

y dirigida por,

D./Dña. Alberto Ros

D./Dña. Alexandra Jimborean

D./Dña.

DECLARO QUE:

La tesis es una obra original que no infringe los derechos de propiedad intelectual ni los derechos de propiedad industrial u otros, de acuerdo con el ordenamiento jurídico vigente, en particular, la Ley de Propiedad Intelectual (R.D. legislativo 1/1996, de 12 de abril, por el que se aprueba el texto refundido de la Ley de Propiedad Intelectual, modificado por la Ley 2/2019, de 1 de marzo, regularizando, aclarando y armonizando las disposiciones legales vigentes sobre la materia), en particular, las disposiciones referidas al derecho de cita, cuando se han utilizado sus resultados o publicaciones.

Además, al haber sido autorizada como compendio de publicaciones o, tal y como prevé el artículo 29.8 del reglamento, cuenta con:

- *La aceptación por escrito de los coautores de las publicaciones de que el doctorando las presente como parte de la tesis.*
- *En su caso, la renuncia por escrito de los coautores no doctores de dichos trabajos a presentarlos como parte de otras tesis doctorales en la Universidad de Murcia o en cualquier otra universidad.*

Del mismo modo, asumo ante la Universidad cualquier responsabilidad que pudiera derivarse de la autoría o falta de originalidad del contenido de la tesis presentada, en caso de plagio, de conformidad con el ordenamiento jurídico vigente.

En Murcia, a 17 de September de 2024

Fdo.: Sawan Singh

Información básica sobre protección de sus datos personales aportados	
Responsable:	Universidad de Murcia. Avenida teniente Flomesta, 5. Edificio de la Convalecencia. 30003; Murcia. Delegado de Protección de Datos: dpd@um.es
Legitimación:	La Universidad de Murcia se encuentra legitimada para el tratamiento de sus datos por ser necesario para el cumplimiento de una obligación legal aplicable al responsable del tratamiento. art. 6.1.c) del Reglamento General de Protección de Datos
Finalidad:	Gestionar su declaración de autoría y originalidad
Destinatarios:	No se prevén comunicaciones de datos
Derechos:	Los interesados pueden ejercer sus derechos de acceso, rectificación, cancelación, oposición, limitación del tratamiento, olvido y portabilidad a través del procedimiento establecido a tal efecto en el Registro Electrónico o mediante la presentación de la correspondiente solicitud en las Oficinas de Asistencia en Materia de Registro de la Universidad de Murcia

Abstract

Over time, computer architects have steadily increased hardware complexity to deliver higher performance. Several mechanisms have been introduced to improve processor performance such as increasing the pipeline depth, introducing out-of-order (OoO) execution, and enhancing the memory hierarchy. All mechanisms aim to hide the latency of instructions and data accessing memory, ensuring the processor is always busy executing instructions.

Among the various mechanisms designed to improve processor performance, OoO execution stands out as a critical innovation. While increasing pipeline depth and enhancing the memory hierarchy aim to streamline instruction processing and data access, OoO execution aims to improve processor performance by speculatively executing instructions non-sequentially as soon as their required operands become ready, instead of adhering to the sequence of previous instructions. However, OoO execution requires heavy bookkeeping to ensure correctness and the program order. This relies on structures such as the reorder buffer (ROB), load queue (LQ), store queue (SQ), and store buffer (SB).

Every instruction requires an entry in the ROB, while loads and stores also require entries in the LQ and SQ, respectively. LQ and SQ are used to maintain the memory order and thus both support associative searches. The processor stalls if these structures are full, and the instruction cannot proceed. Therefore, these structures can become bottlenecks if too small, causing processor stalls, or too large, resulting in slow searches and performance degradation.

Additionally, these structures are not utilized efficiently if the processor frontend is not able to deliver useful instructions. The processor fetches instructions either from the L1I cache or the μ-op cache. The μ-op cache which was initially featured as a power-saving mechanism, has been shown to be beneficial in improving performance by saving the fetch and decode latency. However, the μ-op cache is not always able to deliver useful instructions, leading to frontend stalls. This can further impede performance, particularly for applications with large instruction footprints such as server and data center workloads.

In this work we focus on techniques to improve the utilization of ROB, LQ, SB, and μ-op cache that play a crucial role in hiding instructions' latency while ensuring correctness. The thesis makes the following contributions:

Contribution 1: We introduce Regional Out-of-Order Writes in Total Store Order (ROOW), which allows safe out-of-order writes within data-race-free (DRF) regions, reducing processor stalls by 7.11% and improving execution time by 8.13%.

Contribution 2: To reduce unnecessary searches in the LQ we proposed Compiler-Assisted Efficient Load-Load Ordering (CELLO) which smartly filters around 47% of LQ search and improves execution time by 2.8% and reduces the LQ energy expenditure by 33%.

Contribution 3: To reduce the total number of executed instructions and improve the utilization of the processor structures RoB, LQ, and SQ, we propose an efficient mechanism to fuse non-consecutive instructions. Helios considerably reduces the stalls at the level of these structures, achieving 8.2% performance improvements.

Contribution 4: Lastly, to tackle frontend stalls, we introduce μ-op cache prefetching (UCP), targeting server workloads with large instruction footprints. UCP prefetches μ-ops on the alternate path for hard-to-predict branches, providing a 2% performance improvement.

Through our contributions, we enhance processor performance by reducing stalls due to instructions not being able to proceed due to bottlenecks in processor structures and the frontend, demonstrating significant gains in execution efficiency and overall performance, ranging between 2% - 8%.

Resumen

Con el tiempo, la complejidad del hardware en los computadores ha aumentado constantemente, siempre en pos de ofrecer un mayor rendimiento. Varios son los mecanismos introducidos con este objetivo, por ejemplo, aumentar el número de etapas del cauce, la ejecución fuera de orden (OoO, por sus siglas en inglés) u optimizaciones en la jerarquía de memoria. El fin común de los mencionados mecanismos es el de ocultar la latencia de acceso a la información almacenada en memoria, tanto referente a instrucciones como a datos. Como resultado se consigue que el procesador siempre esté realizando trabajo útil y por tanto ofrezca un mayor rendimiento. La ejecución fuera de orden (OoO) destaca entre el resto de mecanismos. Esta principalmente se centra en ejecutar las instrucciones en el cauce del procesador de forma no secuencial. En el momento en el que los operandos de una instrucción que se ha introducido al camino de datos están disponibles estas se ejecutan si existe una unidad funcional disponible para ello. De esta forma, las instrucciones se pueden ejecutar en un orden diferente al especificado por el programa. Sin embargo, estas sí deben cambiar el estado arquitectural del procesador en el mismo orden en el que se emitieron. Con este objetivo y para garantizar la corrección, se requiere hacer un registro exhaustivo de las instrucciones "en vuelo" o ya emitidas.

Dentro del procesador podemos encontrar varias estructuras que permiten este seguimiento, como son el Reorder Buffer (ROB), Load Queue (LQ), Store Queue (SQ) y Store Buffer (SB). Para cada instrucción se requiere una entrada en el ROB, mientras que las lecturas y los almacenamientos de datos también requieren entradas en la LQ y SQ respectivamente. Estas dos últimas estructuras se utilizan para forzar el mantenimiento del orden de los accesos a memoria tal y como fueron emitidos. Por ello, ambas estructuras admiten búsquedas asociativas. Si cualquiera de estas estructuras llegase a llenarse, el procesador no podría continuar emitiendo instrucciones e irremediablemente debe producirse una parada. Es importante optimizar el tamaño de estas en su diseño para mantener un compromiso entre tamaño y tiempo de acceso. Si las estructuras

son demasiado pequeñas, se producirán más paradas, mientras que aumentar su tamaño también aumenta su complejidad a la hora de implementarlas, resultando en búsquedas más lentas y por tanto en una degradación de rendimiento.

Para poder utilizar el mecanismo de ejecución fuera de orden de una forma eficiente, el front-end del procesador, adicionalmente, debe ser capaz de proporcionar las suficientes instrucciones al cauce. Por lo general, las instrucciones pueden obtenerse ya sea de la caché de micro-operaciones (μ-op) o desde el nivel más bajo de la caché de instrucciones (L1I). La caché de micro-operaciones inicialmente fue diseñada como un mecanismo útil para reducir la energía disipada por el procesador. Adicionalmente ha demostrado ser beneficiosa para mejorar el rendimiento al reducir el tiempo que tarda el procesador en recuperarse ante un salto mal predicho. Sin embargo, esta caché no siempre puede proporcionar las suficientes instrucciones, siendo un potencial cuello de botella para el rendimiento, particularmente en aplicaciones en las que el rango de direcciones de memoria que se usa para almacenar las instrucciones es lo suficientemente grande. Un ejemplo claro de este tipo de aplicaciones son las cargas de trabajo de servidores y centros de datos, muy comunes a día de hoy en grandes empresas.

En el desarrollo de este trabajo nos hemos centrado en técnicas que mejoran el grado de utilización del ROB, LQ, SB y la caché de μ-op. Las contribuciones logradas en esta tesis son las siguientes:

Problema de investigación 1: Los modelos de consistencia de memoria más estrictos, como la Consistencia Secuencial, o como es comúnmente conocida, Sequential Consistency (SC), ofrecen una semántica más intuitiva a los programadores, ya que preserva el orden de programa de todos los accesos de memoria. Por otro lado, explotar el paralelismo a nivel de memoria o Memory Level Parallelism (MLP), que trata de reordenar las instrucciones para ocultar la latencia de las operaciones de memoria es igualmente clave para poder obtener cierto rendimiento. El modelo de consistencia Total Store Order (TSO), el cual es soportado por los procesadores Intel y AMD, consigue alcanzar un buen equilibrio entre ofrecer una semántica aceptable al programador y un buen rendimiento al permitir el reordenamiento efectivo de las instrucciones de carga de memoria con respecto a las de almacenamiento. La latencia de las operaciones de almacenamiento se oculta al permitir que estas se realicen fuera de la ruta crítica del procesador, a costa de una notable relajación en la semántico del modelo de consistencia. El modelo TSO preserva el orden load-store y los procesadores que lo usan, utilizan el SB para mover a este las operaciones de almacenamiento pendientes de terminar del cauce. El SB realiza operaciones de escritura a memoria en orden, causando un cuello de botella si la instrucción de la cabeza del SB está esperando para finalizar una escritura (Puede ser debido a un fallo de caché de

alta latencia).

Solución propuesta: Si de alguna forma, el procesador pudiese realizar operaciones de escritura fuera de orden evitando violar la semántica TSO, la entrada que espera en la cabecera del SB no paralizaría las operaciones de escritura que le suceden en el orden secuencial de programa, evitando, como resultado, que el SB se convierta en un cuello de botella en la arquitectura.

Como solución proponemos un rediseño del SB denominado "Regional Out-of-Order Writes in Total Store Order" (ROOW) o "Escrituras regionales fuera de orden en TSO". Nuestro mecanismo propuesto permite al procesador realizar algunas operaciones de escritura fuera de orden siempre y cuando pertenezcan a una región designada como "segura". Las regiones seguras se definen como regiones libres de condiciones de carrera o Data-Race-Free (DRF). Usando el compilador, se delimitan las regiones DRF, en las cuales se garantiza que ningún subproceso o núcleo accede de forma concurrente a la misma ubicación de memoria. Utilizando la información obtenida por el compilador sobre las regiones DRF, el procesador puede reordenar las operaciones de escritura sin romper la semántica de TSO para optimizar el uso del SB. Nuestro estudio muestra que ROOW reduce las paradas del procesador en un 7,11 % de media en todos los benchmarks probado, mejorando así el tiempo de ejecución en un 8,13 %.

Problema de investigación 2: A pesar de que TSO preserva el orden entre lecturas, en hardware estas se reordenan, de forma especulativa entre sí para poder explotar más MLP. Esta ejecución especulativa requiere que toda instrucción de lectura se coloque en orden en la LQ. La LQ es una estructura de contenido direccionable o "content-addressable memory (CAM). En determinados eventos, esta es explorada para evitar exponer cualquier violación del orden secuencial dada por un reordenamiento. Si una violación del orden es detectada, el mecanismo acoplado a la LQ ayuda a la arquitectura a recuperarse de la especulación de reordenamiento errónea. La LQ es una de las estructuras clave más críticas de un procesador, en términos de rendimiento y consumo energético. De ella depende mantener en orden todas las lecturas en vuelo y debe soportar búsquedas con prioridad que, por motivos de rendimiento, se han de realizar asociativamente. Además, esta es accedida cada vez que se realiza un almacenamiento en pos de salvaguardar la semántica secuencial. También, en cualquier evento de reemplazo y en invalidaciones de un dato de caché, esta es accedida para comprobar que el ordenamiento de las lecturas con respecto a otras lecturas (load $\rightarrow$ load) se cumple consistentemente. Además, una alta contención en los puertos de búsqueda (snoop), también puede suponer una fuente de paradas para el procesador ya que el número de puertos es limitado y no poder servir una búsqueda tendrá como resultado retrasar la ejecución de la instrucción que la requería.

La situación de congestión puede verse agravada en caso de que se utilicen mecanismos que permitan la gestión de varios hilos lógicos en el mismo chip físico, como es el caso de Simultaneous Multi-Threading (SMT). En este tipo de procesadores, varios hilos lógicos comparten, tanto estructura arquitectural (soporte OoO) como el estado coherente de las líneas de caché L1D. Por ello, el mecanismo de coherencia de caché, que como se ha mencionado, dispara los eventos de búsqueda por invalidación, ante un almacenamiento realizado por otro hilo, puede ahora no notificar al procesador en estas situaciones. Esto puede ocurrir si el hilo lógico escritor, potencialmente conflictivo, ejecuta sus instrucciones en el mismo chip físico que un lector sobre esa posición de memoria. Sin dichas búsquedas disparadas por la invalidación, las violaciones del orden load $\rightarrow$ load podrían no ser detectadas. Para evitar este tipo de contingencias, en un núcleo SMT, cada almacenamiento que realiza una escritura desde la SQ (o SB) a la L1, activa un evento de búsqueda en la LQ. La búsqueda rastrea lecturas en estado especulativo coincidentes para esa misma dirección de memoria por parte de otros hilos. Esta solución puede, potencialmente, aumentar la latencia de escritura (debido a la congestión de los puertos de búsqueda), por lo que suele realizarse en paralelo con la escritura en caché. Sin embargo, aunque se realice en paralelo con la escritura, la búsqueda generará aún más contención en los puertos de búsqueda de la LQ.

Solución propuesta: Usando la información sobre DRFs que el compilador puede extraer, el procesador puede omitir de forma segura las búsquedas en la LQ dentro de estas regiones. Esta solución es capaz de aliviar la congestión sobre los diferentes puertos de búsqueda. A su vez, las operaciones de escritura en el camino crítico se aceleran al poder ubicar las lecturas en la LQ sin esperas por la contención. Nuestra propuesta sobre esta idea se denomina Ordenamiento Lectura-Lectura Eficiente Asistido por Compilador en Regiones Libres de Condiciones de Carrera, en inglés Compiler-Assisted Efficient Load-Load Ordering in Data-Race-Free Regions (CELLO). Con ella conseguimos demostrar que el procesador puede reducir, en todo el conjunto de benchmarks probados, las búsquedas sobre la LQ en un 47 % de media. Como resultado de la disminución en la congestión, el tiempo de ejecución mejora en un 2,8 %. Derivado de la mejora de rendimiento y del uso de los puertos de la estructura, el gasto energético también se ve reducido en un 33 % de media.

Problema de investigación 3: Muchas microarquitecturas modernas utilizan conjuntos de instrucciones o instruction set architecture (ISA), compuestos por macroinstrucciones. Estas, realmente representan a una o más microoperaciones o μ-ops, que son primitivas más pequeñas y simples. El uso de macroinstrucciones, permite reducir el número de accesos a memoria y optimizar el ancho de banda

utilizado, al permitir, usando el mismo puerto, obtener una sola instrucción que representa un conjunto de varias operaciones. Estas macroinstrucciones son descompuestas e interpretadas por el hardware en un proceso conocido como "cracking". Una vez aplicado el cracking a una instrucción, cada una de las μ-ops que la comprenden puede ser enviada a la etapa de emisión (issue). Estas μ-ops son lo suficientemente simples para que las unidades funcionales del procesador, así como las diferentes estructuras hardware las puedan manejar eficientemente. Por otro lado, la fusión de instrucciones reorganiza varias de estas μ-ops antes de ser emitidas al cauce fuera de orden. Las instrucciones tras la fusión siguen siendo lo suficientemente simples para ser correctamemte gestionadas por el hardware. A la misma vez, estas representan a un número considerable de μ-ops, lo que resulta en una potencial mejora del rendimiento al reducir la congestión del front-end, y, ahorro de recursos del cauce como entradas del ROB, LQ y SQ o recursos del planificador.

En la fusión se unen, en una sola, dos (o más) μ-ops que deben situarse contiguamente en el orden secuencial del programa. Si la restricción de contigüidad fuese relajada, se podría fusionar un número no despreciable de μ-ops no adyacentes adicionales. Al situarse en la etapa de decodificación (decode), el hardware de fusión debe basarse en la información de la que se dispone localmente en esta etapa del camino de datos. Entre la información disponible, se incluye: un registro base que contiene la dirección de memoria, así como los registros de los operandos, aunque no sus contenidos. De esta forma, dos μ-ops podrían ser fusionadas, incluso si sus registros base son diferentes, en base a la dirección de memoria de cada una, si pudiese conocerse esta de antemano.

Solución propuesta: Helios, como se conoce a nuestra propuesta, es capaz de reducir eficazmente las paradas del procesador por saturación de ROB, LQ y SQ. Helios emplea un predictor de fusión para decidir si se deberían fusionar varias instrucciones y cuál es la dirección de memoria de las instrucciones no consecutivas o con diferente registro base. Como resultado, Helios consigue fusionar un 5,5 % adicional de μ-ops dinámicas con respecto a la fusión estándar. Como resultado, incluir Helios en la microarquitectura probada induce una mejora de IPC del 8,2 % con respecto a la versión estándar del hardware de fusión.

Problema de investigación 4: El código ejecutado en los centros de datos, consiste en una cantidad de instrucciones que fácilmente supera las capacidades de la caché L1I en varios órdenes de magnitud. Además, se prevé que la complejidad y el tamaño sigan aumentando a un ritmo del 20 % por año. No solo la caché L1I se ve saturada con este tipo de carga de trabajo, sino que también los BTBs, de gran tamaño, se ven en problemas para monitorizar el destino de todos los saltos.

Los fallos de acceso a la caché paralizan el front-end al tratar de traer instrucciones al procesador. El mecanismo conocido como Decoupled Fetching o Fetch Directed Prefetching (FDP) ayuda a mitigar este problema. En él la generación de la dirección de memoria de la instrucción y la propia obtención de esta del sistema de memoria son desacopladas. Esto permite generar direcciones de memoria de instrucciones antes de que se soliciten siguiendo el camino indicado por el predictor de saltos. Así, cuando el procesador acceda a la L1I, las instrucciones estarán almacenadas en caché preparadas para su uso. Donde antes se producía un fallo en caché que producía una parada, con FDP, muchas veces hay un acierto que permite que la ejecución continúe. Para guiar la prebúsqueda, se utiliza la BTB, que a pesar de su crecimiento constante en tamaño a lo largo de las sucesivas generaciones de procesadores comerciales, encuentra bastantes dificultades para monitorizar todos los saltos del código. Un fallo en la BTB provoca que se busquen instrucciones en la L1I con rutas potencialmente erróneas, las cuales acaban siendo insertadas en el cauce de ejecución. Las rutas erróneas provocan que el cauce tenga que vaciarse parcialmente una vez que las distintas bifurcaciones en el flujo de ejecución han sido identificadas en la decodificación.

Finalmente, los grandes rangos de instrucciones que se manejan en este tipo de código superan la capacidad de la caché de microoperaciones, (μ-op) limitando su utilidad. En muchas microarquitecturas utilizadas en centros de datos se implementa una caché de μ-op. Esta estructura almacena microinstrucciones ya decodificadas en lugar de instrucciones del ISA y cumple con dos propósitos. El primero es mejorar la eficiencia energética ya que acertar consistentemente en la caché μ-op evita accesos a la L1I y evita tener que decodificar las instrucciones. El segundo es mejorar el rendimiento ya que desde la caché μ-op se puede obtener las instrucciones más rápido que desde el proceso de decodificación. Adicionalmente, la longitud del camino de datos se "acorta" cuando se acierta en la caché μ-op y se evita pasar por la etapa de decodificación, haciendo que el impacto de un fallo en la predicción de saltos sea, de media, menor. Desgraciadamente, el tamaño de las cachés μ-op modernas permite un rango de acción mucho menor que el de la caché de instrucciones.

Solución propuesta: Para poder hacer frente a la saturación que sufre el front-end del procesador en centros de datos, proponemos la prebúsqueda en la caché μ-op o también conocido por su nombre en inglés μ-op cache prefetching (UCP). El estudio que hemos llevado acabo muestra que la FTQ es incapaz de ocultar las latencias por fallos en la L1I tras especular erróneamente un salto. El principal objetivo de UCP es el de hacer prebúsqueda de μ-ops en el camino no tomado, o alternativo, de un salto en pos de almacenar estas en la caché μ-op. De esta forma, el procesador es capaz de recuperarse de una forma mucho más efectiva de un fallo en la predicción de saltos, ya que las μ-ops que necesita ya están

almacenadas en la caché μ-op. Adicionalmente, para evitar la saturación de la caché, UCP sólo empieza una prebúsqueda en la ruta no tomada para aquellos saltos que son considerados difíciles de predecir.

Las pruebas llevadas a cabo con UCP muestran que con una sobrecarga mínima de hardware, UCP puede proporcionar una mejora del IPC del 2 % de media. Con este mecanismo se consigue mejorar el rendimiento del procesador al reducir las paradas causadas por cuellos de botella en el sistema de memoria y en el front-end del procesador. Esta mejora se traduce en mejoras significativas del rendimiento general y la eficiencia del sistema que se ubican en el rango del 2 % al 8 %.

To my parents who always inspired me to learn!

Acknowledgement

This thesis has been a journey filled with learning and growth. First, I would like to express my deepest gratitude to Alexandra and Alberto for their guidance and support throughout this process. I still remember our first meeting in Uppsala in 2018—I was both nervous and excited, but you both made me feel comfortable and welcomed. I am very thankful to you for giving me the opportunity to enter the world of computer architecture.

After Uppsala, I had the privilege of moving to Murcia to work with you again for my master's thesis. When Alberto offered me a PhD position in the group after finishing my thesis, I was thrilled to accept. I am grateful to both of you for your ongoing guidance and support during my PhD. I have learned so much from you—Alberto, you taught me not only the technical aspects but also how to identify a problem, take the first steps, and bring a project to a conclusion. You always encouraged me to aim for the best and to believe in my research. Alexandra, your insightful ideas during the development phase greatly improved my work, and both of you have helped me become a better writer. I still remember my first attempt at drafting a paper, and when I look back, I see how much I have improved thanks to your feedback (though I still have plenty of room for growth).

Alexandra and Alberto, thank you again for everything. Without your belief in me and the opportunities you provided, my dream of doing research might have never been accomplished. Thank you for your late-night replies and help, specially during deadlines. In Indian culture, advisors are like parents, nurturing and guiding their students, and I feel blessed to have had you both in that role.

Just as I am grateful to Alexandra and Alberto, I would also like to thank Arthur. Arthur, you have always been there to help me with technical details. I learned so much from you, and visiting you in Grenoble was a fantastic experience. I still remember the lunch you organized to help new members of the group get to know each other.

Manuel and Josue, I am so glad we had the chance to work together. Manuel,

you have not only supported my research but also taken care of everyone in the CAPS group. You helped me navigate the paperwork for my PhD, and I will always cherish our small hallway conversations. Josue, thank you for your assistance with SMT details. I see you as my brother who was always there to help whenever I needed it. I still remember Alberto saying we should all learn to debug from you, and I hope to one day be as skilled as you are.

To Alexandra, Alberto, Arthur, Manuel, and Josue, thank you for all the meetings, discussions, and late-night paper reviews. This PhD is only possible because of your support. I have always admired your work and ethics, and I hope to one day live up to the standards you have set.

Thank you so much, Biswa, for all your technical help, great discussions, and always being there when I needed some help. I have learned a lot from you and hope to keep learning in the future. I also appreciate the times we have shared over Indian food in Murcia. Even if I am not around, I hope you will keep visiting Murica.

This PhD does not belong solely to me; it belongs to my parents, who have supported me in every decision I have made. Even though they were sad about me moving to a different country, they never stopped me and always tried to hide their feelings so it would not affect me. You both have sacrificed so much for me, and I hope to make you proud.

I also want to mention my sister and Hyeji, who have always supported me and never doubted me, no matter the situation. My sister has been protecting me since childhood—no matter how much taller I am now; she still sees me as her little brother. Hyeji, thank you for waiting for me to finish my PhD. I know that mentioning you here is not enough, but I hope I can make it up to you.

My stay in Murcia would have been impossible without my extended family, and the CAPS group. From morning coffee discussions to heated AMD vs. Intel lunch debates, you have all been there for me. Victor, you have always helped me in every situation. I will never forget our bike rides to work in the morning, our late-night technical discussions at home, and all the other moments we shared (some of which I know you remember too). Sebas, thank you for the legendary lunch barbecues and burgers. Spending time with you in Murcia was amazing—it was always fun to talk to you about anything. Recently, we started working on a project together, and I learned so much from you. Agus, thank you for being there during my ChampSim frustrations and helping me with the code. You always helped whenever I asked. Ashkan, you and I started at the same time and went through all the processes together, like applying for a work permit. Those days are still fresh in my mind. You have always taken care of so many things without anyone asking, like reserving tables for lunch or bringing me something to eat when I was on tight deadlines. Thank you for everything. Nicolas, thank

you for the bike ride in Italy—you gave me a great story to tell about how we survived Italian roads. Thank you for being such a great friend and office mate. Eduardo, thank you for taking care of Echo and always being available to help. Every time I have asked you for something, you have always done your best to solve the problem.

I would also like to thank the new members of the CAPS group who have joined recently—Ravi, Shreya, Emily, and Maitri. Thank you for the interesting discussions during lunch and coffee breaks. Pascual and Emilio, thank you for many discussions related to the processor frontend, I always learned something from our conversations.

I would also like to thank two former members of the CAPS group, Paco and Pablo. Paco, you were my first contact in Murcia and have always helped me in every situation. Even after graduating, you assisted me with the PhD paperwork. Pablo, we missed you during our lunch breaks, and I am so glad that we might be in the same city soon.

While I was in Grenoble, Chandana and Harsha welcomed me and helped me settle down. Thank you for all the delicious food you both cooked for me. Grenoble became a home away from home because of you both.

My stay in Murcia would have been stressful if I did not have an administrator like Paco. Thank you, Paco, for taking care of all the paperwork. I will miss coming to Pacha for my morning coffee and saying, "Buenos días, Paco."

Finally, I would like to thank my friends in Murcia, who have become like family—David and Joha. You two are another achievement I am taking from Murcia. You have been there for me in every situation, from my first few months in Murcia to my last few days. Thank you again for taking care of me, helping me move, picking me up from the airport during COVID-19, and so much more.

To everyone who has been part of my PhD directly or indirectly, *thank you!*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The increasing demand for faster processors has led to a significant increase in the complexity of processors, which is forcing manufacturers to pack more transistors in a single die. This has led to the development of complex processors with multiple cores, multiple threads, and multiple execution units. In 1965 Moore predicted that the number of transistors in a single die would double every two years [51]. Till now the speculation holds well as shown in Figure 1.1, but the architecture community and industry have been debating that Moore's Law may end soon. Intel's CEO, Pat Gelsinger, has stated that Moore's Law is *"alive and well"* emphasizing the company's commitment to continue advancing chip manufacturing technology [6]. On the other hand, Nvidia's CEO, Jensen Huang, has declared that Moore's Law has ended, pointing out that the advancements in transistor density are no longer keeping pace with Moore's original prediction [6]. Experts from MIT have also weighed in, suggesting that while the miniaturization of transistors is still occurring, the pace has slowed down significantly compared to the standard set by Moore's Law. They argue that the law has effectively been over since at least 2016, as the expected doubling of components every two years is no longer happening [7].

In summary, while there is no consensus on the death of Moore's Law, it's clear that the rate of progress in transistor density is not as predictable as it once was. The industry is exploring alternative ways to drive computer performance, including improvements in software, algorithms, and hardware architecture.

Another trick in the book to increase the performance of the processor is to increase the clock speed of the processor. Figure 1.2 shows the increase in processor clock speed over the years. The clock speed of a processor is the

number of cycles it can perform per second. The higher the clock speed, the faster the processor can perform calculations. The first microprocessor, Intel 4004 [81], was introduced in 1971. It had a clock speed of 740 kHz and could perform 60,000 operations per second. The latest processors, like Intel Core i9-10900K [64], have a clock speed of 5.30 GHz and can perform 21.2 billion operations per second. Although the clock speed has increased significantly over the years, the increase in clock speed has slowed down in recent years. This is due to the power consumption and heat dissipation issues that arise with the increase in clock speed [78].

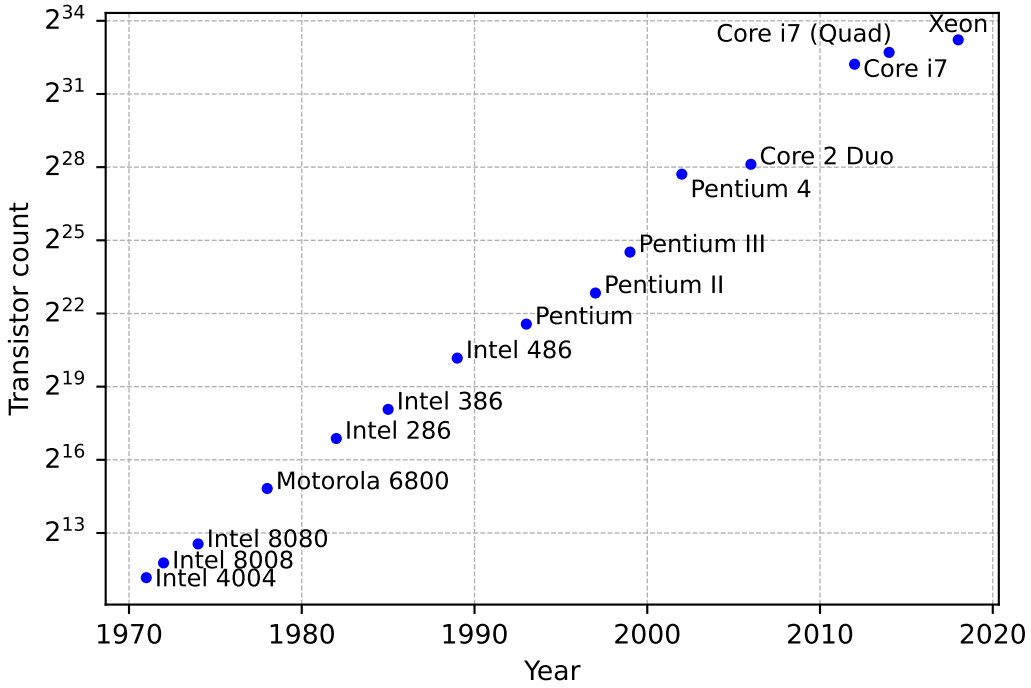


Figure 1.1: Moore's Law (source [1]).

Despite advances in clock speed, simply increasing the frequency at which a processor operates is not the only way to improve performance. Today's processors include a variety of designs and architectures tailored to different applications and market segments. By leveraging diverse types of processors, such as those optimized for high performance, low power, or specific computing tasks, manufacturers can more effectively meet diverse computing needs. The next section examines some of the more common types of processors based on

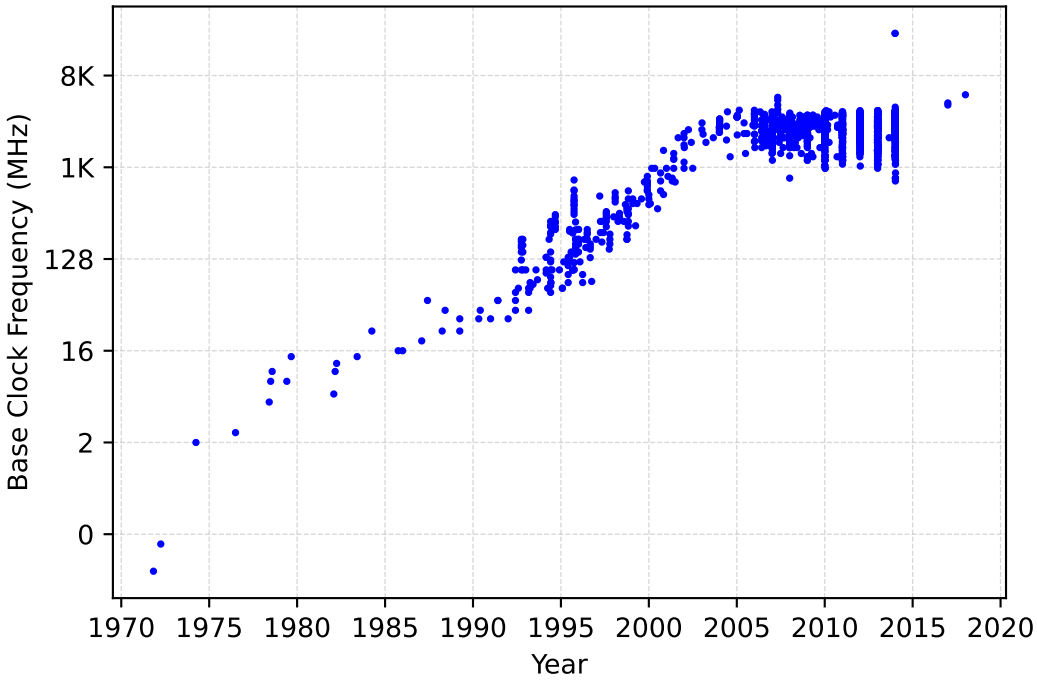


Figure 1.2: Increase in processor clock speed over the years (source [1]).

their design and intended market segments. Some common types of processors based on design are as follows:

- **In-order and Out-of-order (OoO)**, In-order CPUs execute instructions in the order. In contrast, OoO CPUs can execute instructions out of order. This allows OoO CPUs to execute instructions that are not dependent on each other in parallel. OoO processors are more complex than in-order processors but provide better performance while in-order processors are simpler and therefore more energy efficient but deliver lower performance. OoO CPUs hide latencies when instructions are not ready to execute by speculatively executing ready instructions, predicted to be independent. This increases performance at the cost of higher hardware complexity, which in turn leads to higher energy expenditure. In contrast, in-order processors block when instructions are not ready to execute. This keeps them simple, thus more energy efficient, but less performant than OoO. Each type is suitable for a different market segment for example in-order processors are suitable for embedded systems where power consumption is critical

while OoO processors are suitable for high-performance computing where performance is critical.

- **Scalar and superscalar**, a scalar processor cannot execute more than one instruction at a time and thus cannot achieve a throughput of more than 1. In contrast, a superscalar processor can execute multiple instructions in various pipeline stages at a time, thus having the potential to achieve a higher throughput.

- **Vector processors** are designed to exploit the data parallelism in the program. They can execute the same instruction on multiple data elements in parallel. This is useful for scientific and engineering applications where the same operation is performed repeatedly on multiple pieces of data. Intel AVX extension [80] is an example of a vector processor that can perform 8 double precision floating point operations in parallel.

- **Multicore processors** consist of more than one core on a single die. Each core can execute instructions independently and the communication between the cores is handled by an interconnection network. This allows the processor to execute multiple threads in parallel. Multicore processors are useful for applications that can be parallelized.

- **Multithreaded processors** allow the execution of different threads on the same core. All the threads share the same resources of the core. This allows the processor to switch between threads when needed.

Among the several types of processors, superscalar out-of-order (OoO) processors stand out for their ability to significantly improve performance by executing instructions out of order. This architectural approach allows the processor to use its resources more efficiently, reducing idle time and increasing overall throughput. Due to its relevance and widespread application in modern computing, this thesis will focus on OoO processors.

Processors use out-of-order (OoO) execution to improve performance by executing instructions as soon as their operands are available, rather than waiting for the previous instruction to complete. Even if the processor executes the instruction out-of-order, the instructions still need to be retired in program order to ensure that the program behaves as if it was executed in-order. To achieve this the processor must keep track of the in-flight instruction. This is done by using a structure called a reorder buffer (ROB). For memory instructions, the processor must keep track of the memory order to ensure that the memory operations are done in program order. This is done by using load queue (LQ) and store queue

(SQ). All the loads allocate an entry in the LQ while stores allocate an entry in the SQ. The processor uses LQ and SQ to ensure that the memory operations are done in program order. Since store operations access the memory after the store instruction is retired, the processor moves the entry from the SQ to another structure called store buffer (SB). Thus, SB contains the retired store operations which then access the cache in program order. In current processors, SQ and SB are implemented as a single structure and the division between the two structures is done logically.

The size of these structures is critical for the optimal performance of an OoO processor. If ROB, LQ, SQ, and SB are too small, the processor will have to stall until the structures have space to allocate new entries. During the OoO execution processor snoops the LQ and SQ for memory order violations. When the loads execute, they access the cache while in parallel also searching the SQ to find any recent producer. If the load finds a recent producer in the SQ, the value is forwarded from the store to the load. LQ is searched when the stores resolve their addresses to find any matching entry that can break sequential semantics. If an entry is found the processor triggers a squash and the instructions after the store are re-executed. Additionally, in a simultaneous multithreading processor (SMT), the processor snoops the LQ when the store performs to track any load-load reordering. This is required as in SMT processors multiple threads can execute on the same core, which means there will be no cache invalidation to ensure the load-load ordering. Thus, LQ and SQ are complex CAM structures that keep both FIFO order and allow searches. This creates a scaling issue as if LQ and SQ are too large, snooping through these structures will be slow and can lead to performance degradation.

Another problem arises when the processor's frontend fails to deliver enough instructions to feed the backend. This can happen due to the frontend being too slow as the processor will have to stall until the frontend delivers the instructions. Several applications such as server workloads have large instruction footprints. These applications can have numerous instructions in-flight at any given time and thus suffer from high L1I or μ-op cache misses. μ-op caches especially can lead to performance degradation as the processor as the processor takes one cycle to switch from fetching μ-op cache to L1I cache.

Thesis Contributions

This thesis focuses on improving the performance of OoO processors. The main research question we address is how to improve the utilization of the various queues and the processor frontend to reduce processor stalls. Below we summa-

rize various contributions of this thesis:

Research Problem 1: Strong memory consistency models such as Sequential Consistency (SC) [46] offer intuitive semantics to programmers by preserving the program order of all memory accesses. On the other hand, exploiting memory level parallelism (MLP), which relies on reordering instructions to hide long-latency memory operations, is key for performance. The Total Store Order (TSO) memory consistency model [67], supported by Intel and AMD processors, achieves a good balance between programmability and performance by allowing load instructions to be effectively reordered with respect to store instructions. This way, the latency of store operations is hidden by allowing them to perform out of the processor's critical path, at the cost of relaxing the consistency model semantics. In a TSO model, the store-store order is preserved, and processors using TSO semantics use SB to move the stores from the processor pipeline and keep them in SB. From SB the store performs the memory operations in order which creates a bottleneck if the head is waiting to write in the cache (e.g., on a long latency miss).

Proposed Solution: SB bottleneck can be solved if the processor can perform write operations out-of-order without breaking the TSO semantics, such that the entry waiting at the SB head does not stall the other stores. We propose a new store buffer design called *Regional Out-of-Order Writes in Total Store Order* (ROOW) which allows the processor to perform some write operations out-of-order that belong to a safe region. The safe regions are defined as the data-race-free (DRF) regions. The compiler delineates the DRF regions and offers the guarantee that different threads or cores executing concurrently will not access the same memory location. Thus, the processor using the DRF information smartly allows the stores to perform the memory operations out-of-order only in the DRF region without breaking the TSO semantics. Our study shows that ROOW can reduce the processor stalls by 7.11% on average across all the benchmarks and improves the execution time by 8.13%.

Research Problem 2: Despite TSO preserving the load-load order, in hardware, loads are speculatively reordered with respect to each other to improve MLP [30]. This speculative execution requires all loads to be placed in order in the LQ, a content-addressable memory (CAM) structure that is searched on certain events to prevent exposing speculative ordering violations and is coupled with a mechanism to recover from misspeculation when detected. The LQ is one of the most critical structures in a processor, in terms of performance and energy [29]. It needs to keep all in-flight loads in order and support priority searches which,

for performance reasons, are done associatively. Furthermore, it is searched frequently: each time a store executes, to safeguard sequential semantics [52], and on any invalidation or cache eviction, to enforce the load→load order [30]. In addition to the high contention on its search (i.e. snoop) ports, the LQ also stalls OoO processors when it becomes full. As discussed earlier, a speculative reordering that violates the load→load order in a thread could be exposed by stores performed by a different thread. In an SMT processor, both threads can run in the same SMT core, sharing the coherent state of the cache lines in the L1D. This means that a thread will not receive a coherence invalidation from a store performed by a co-running thread (in the same SMT core). Without such an invalidation, no LQ search is triggered to check for potential load→load order violations. To address this issue, in an SMT core, each store that writes from the SQ to the L1 triggers a search in the core's LQ looking for matching speculative loads from the co-running threads. To avoid increasing the writing latency, this LQ search is typically performed in parallel with the cache write. Nevertheless, this solution exacerbates contention at the LQ search ports.

Proposed Solution: Using the DRF information the processor can skip the LQ searches in the DRF region. This allows the processor to remove the congestion at the LQ search port. Less congestion at the LQ search port allows the stores on the critical path to perform the associative search in the LQ without waiting. We proposed *CELLO: Compiler-Assisted Efficient Load-Load Ordering in Data-Race-Free Regions* that shows that the processor can reduce the LQ searches by 47% on average across all the benchmarks and improve the execution time by 2.8%. Filtering the LQ searches also allows the processor to reduce the energy expenditure of LQ by 33% on average across all the benchmarks.

Research Problem 3: After retrieving instructions from instruction memory, many modern general-purpose microarchitectures will translate architectural instructions into one or more microarchitectural operations –a.k.a. "µ-ops"– through a hardware process called *instruction cracking*. After *cracking*, all operations in flight in the pipeline are µ-ops. *Cracking* therefore re-arrange one complex architectural instruction into multiple µ-ops that are simple enough for the hardware to handle efficiently, while its dual, instruction *fusion*, re-arranges multiple µ-ops into one that is just complex enough for the hardware to handle efficiently. Fusion has the potential to improve performance by decreasing latency as well as saving pipeline resources such as ROB, Scheduler, and LQ/SQ entries. Fusion is usually thought of as a technique that will fuse two (or more) µ-ops that are consecutive in the dynamic instruction stream. However, we find that if this constraint were to be relaxed, a non-negligible number of additional µ-ops could be fused. Fusion

hardware relies on static information available at decode to decide whether to fuse two memory µ-ops or not. The static information includes a base register that holds the memory address, while at decode the registers are known, but the values are not. Thus, two memory operations with different base registers can also be fused successfully if the memory address is known.

Proposed Solution: Our proposal *Helios* effectively reduces the processor stalls in the RoB, LQ, and SQ by fusing the non-consecutive instructions and instructions with different base registers. *Helios* employs a fusion predictor which allows the processor to fuse with a non-consecutive instruction. Overall *Helios* manages to fuse an additional 5.5% of dynamic μ-ops over the baseline fusion. This results in an IPC improvement of 8.2% over the baseline fusion.

Research Problem 4: Datacenter-class workloads run deep stacks, and their code footprint can exceed current L1I capacities by two orders of magnitude [15]. Furthermore, it is predicted that their code footprint will keep increasing at the rate of 20% per year [15]. Not only does the code not fit in the L1I cache, but the large BTBs also struggle to provide enough reach to track all branches [14–16, 27, 34, 35, 40, 42]. On one hand, L1I misses contribute to performance degradation by stalling the frontend while an instruction is being retrieved from the memory system. This is mitigated by Decoupled Fetching (or Fetch Directed Prefetching, FDP) [59,60], in which fetch address generation and instruction retrieval from the memory system are decoupled. This allows fetch address generation to run ahead during L1I misses, enabling the overlap of instruction misses and performing instruction prefetching based on branch direction and target predictions. On the other hand, FDP relies on the BTB to guide instruction fetch, that is, the burden of caching information about the whole code footprint is shifted from the L1I to the BTB, which, despite steady growth across commercial processor generations, often struggles to capture the whole code footprint. BTB misses causing potentially wrong path instructions to be fetched from the L1I and inserted in the pipeline, causing additional pipeline re-steers once the taken branches are identified in decode. Finally, large code footprints exceed the microarchitectural operation (µ-op) cache capacity, limiting its usefulness. A µ-op cache is currently implemented in many processor designs used in data centers [21, 64]. This structure caches decoded instructions (µ-ops) instead of architectural instructions and serves two purposes. The first is power efficiency, as consistently hitting in the µ-op cache avoids accessing the L1I and bypasses the decoders. The second is performance, as the throughput of the µ-op cache is higher than the one of the "slow path" decoders. This, combined with a shortened pipeline length when hitting in the µ-op cache, can reduce the average

cost of branch mispredictions. However, modern µ-op caches have a smaller reach than instruction caches.

Proposed Solution: To target the frontend stalls in datacenter workloads, we proposed *µ-op cache prefetching* (UCP). UCP focuses on server workloads as these workloads have large instruction footprints and suffer from high frontend stalls. Our study shows that FTQ is unable to hide the L1I miss latency after a branch misprediction. Thus, UCP is designed specially to prefetch the μ-ops on the not-predicted path (often called an alternate path) in the μ-op cache. UCP only begins prefetching the alternate path for hard-to-predict (H2P) branches as these branches are the ones that are often mispredicted. When the branch is mispredicted, the processor has to flush the pipeline and the FTQ must start from nothing thus the L1I miss latency is not hidden anymore. But thanks to UCP, the μ-ops on the alternate path are already prefetched in the μ-op cache. This allows the processor to quickly fetch the μ-ops from the μ-op cache after a branch misprediction. Our study shows that with small hardware overhead UCP manages to provide an IPC improvement of 2% on average.

This thesis is based on the following research outputs. The bullet shows where each paper contributes.

- *Sawan Singh*, Alexandra Jimborean, and Alberto Ros. **Regional Out-of-Order Writes in Total Store Order**. In Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques (PACT). Association for Computing Machinery, New York, NY, USA, September 2020, pp. 205-216. https://doi.org/10.1145/3410463.3414645.

- *Sawan Singh*, Josue Feliu, Manuel E. Acacio, Alexandra Jimborean, and Alberto Ros. **CELLO: Compiler-Assisted Efficient Load-Load Ordering in Data-Race-Free Regions**. 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT), Vienna, Austria, October 2023, pp. 1-13. https://doi.org/10.1109/PACT58117.2023.00009.

- *Sawan Singh*, Arthur Perais, Alexandra Jimborean, and Alberto Ros. **Exploring Instruction Fusion Opportunities in General Purpose Processors**. 55th IEEE/ACM International Symposium on Microarchitecture (MICRO), Chicago, IL, USA, October 2022, pp. 199-212. https://doi.org/10.1109/MICRO56248.2022.00026.

- *Sawan Singh*, Arthur Perais, Alexandra Jimborean, and Alberto Ros. **Alternate Path µ-op Cache Prefetching**. 51st Annual International Symposium

on Computer Architecture (ISCA), Buenos Aires, Argentina, June 2024, pp. 1230-1245. https://doi.org/10.1109/ISCA59077.2024.00092.

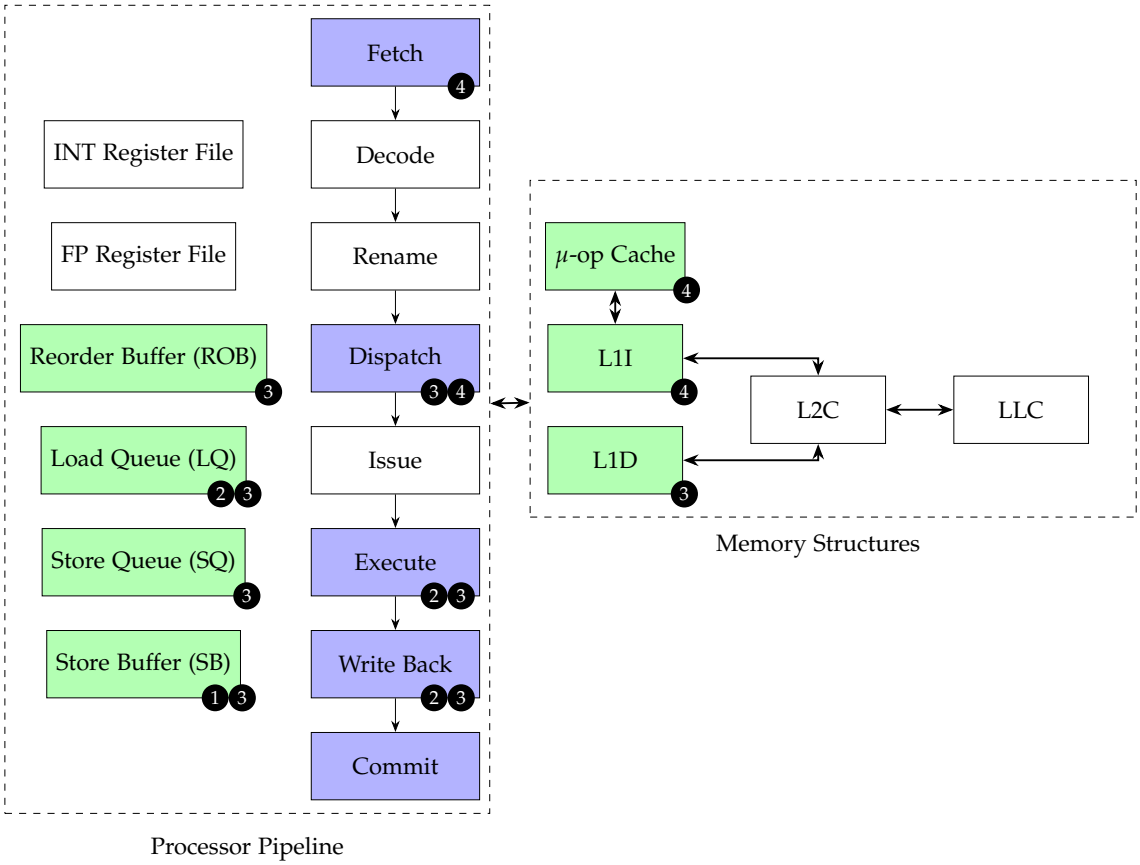


Figure 1.3: Overview of OoO processor pipeline. The structures in green are optimized in this thesis. While the optimized pipeline stages are shown in blue.

Chapter 2

Background

In this chapter, we will discuss how the modern processor is designed and how it executes the instructions. We will discuss the Instruction Set Architecture (ISA) and the Out-of-Order (OoO) processor. Later we discuss the frontend and backend of the processor and how the instructions are executed out of order. In the end we discuss compiler overview and how it is used in this thesis.

2.1 Instruction Set Architecture (ISA)

An Instruction Set Architecture (ISA) is a fundamental component of the abstract model of a processor. It defines how the processor is controlled by the software. The ISA serves as an intermediary between the hardware and the software, specifying both the capabilities of the processor and how it is utilized. The ISA is the only way through which a user can interact with the hardware. The ISA can be regarded as a manual for the programmers, as it is the portion of the machine that is visible to the assembly language programmer, the compiler writer, and the application programmer. The code written by the programmers is converted to machine code by compilers (GCC or clang for example) and then it is ready to execute on the processor. Figure 2.1 shows a comparison between a simple C++ code and its converted assembly code for RISC-V ISA.

The ISA defines the supported data types, registers, memory management, key features (such as virtual memory), and the input/output model of a microprocessor. It specifies the instructions that a microprocessor can execute and the format of those instructions. The ISA also determines how the microprocessor interacts with the main memory and other peripherals. The ISA serves as a bridge between

the hardware and the software. It provides a standardized interface for software developers, allowing them to write programs that can run on different microprocessors without needing to know the specific details of each microprocessor's implementation. The ISA can be extended in two ways. First, new instructions or capabilities can be added to the existing instruction set, providing additional functionality to the microprocessor. Second, the ISA can be expanded to support larger addresses and data values, allowing the microprocessor to work with larger amounts of memory and handle larger data types.

Overall, the ISA plays a crucial role in determining the capabilities and behavior of a microprocessor. It defines the fundamental building blocks that software developers use to create programs, making it a key consideration in computer architecture design and software development.

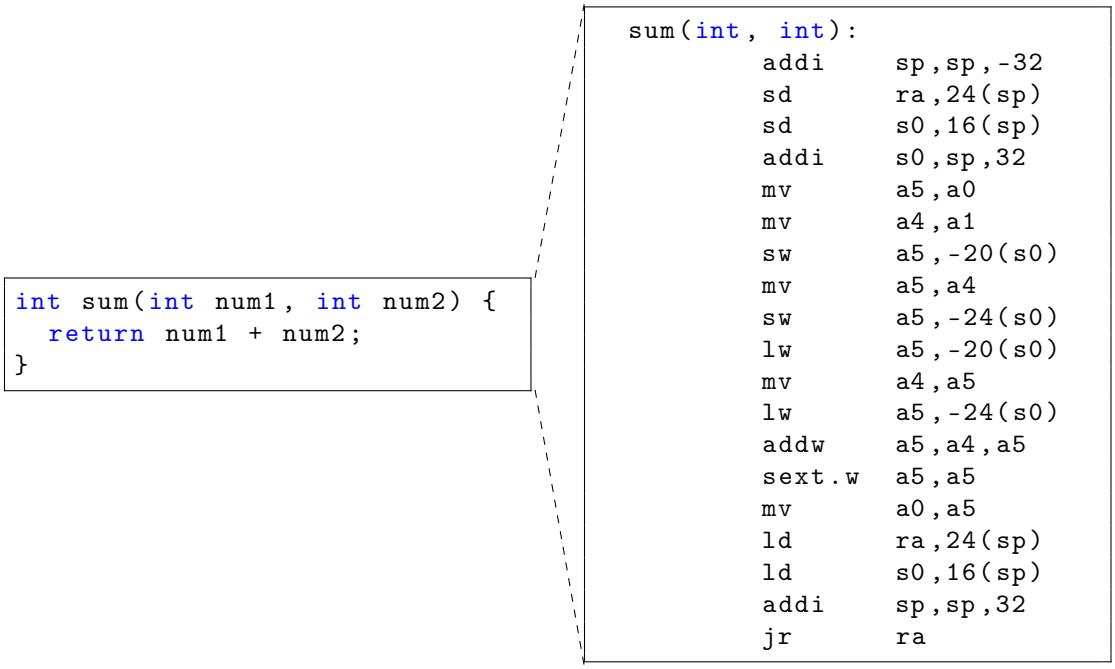


Figure 2.1: Comparison between C++ code and its converted assembly code.

2.1.1 Popular ISAs

2.1.1.1 Complex Instruction Set Computers (CISC)

The Complex Instruction Set Computer (CISC) architecture is characterized by a large and diverse set of instructions that can perform multiple operations in a single instruction. The CISC architecture was developed in the 1960s and 1970s to simplify programming and reduce the number of instructions required to perform a task. The CISC architecture is designed to support high-level programming languages and provide a rich set of instructions that can perform complex operations. The x86 architecture, developed by Intel, is a prominent example of a CISC architecture. The x86 architecture is widely used in personal computers, servers, and embedded systems. Many modern-day processors are based on x86 architecture such as Intel Core i9 [64] and AMD EPYC [8]. The x86 architecture supports many instructions, including arithmetic, logic, control flow, and string manipulation instructions. The x86 architecture also includes support for SIMD (Single Instruction, Multiple Data) instructions, which can perform parallel operations on multiple data elements. The x86 architecture has evolved over the years to include new features and instructions, such as virtualization support, security features, and advanced vector extensions. x86 architecture is used in Chapter 4 to demonstrate the proposed optimizations.

2.1.1.2 Reduced Instruction Set Computers (RISC)

The Reduced Instruction Set Computer (RISC) architecture is characterized by a small and simple set of instructions that can be executed in a single cycle. The RISC architecture was developed in the 1980s as a response to the complexity and inefficiency of CISC architectures. The RISC architecture advocates for simple instructions that can be executed quickly, allowing for high performance and energy efficiency. The RISC architecture is designed to optimize the performance of the processor by reducing the complexity of the instructions and focusing on executing them efficiently. The ARM architecture [3], developed by ARM Holdings, is a prominent example of RISC architecture. The ARM architecture is widely used in mobile devices, embedded systems, and IoT devices. ARM architecture is known for its simplicity, modularity, and energy efficiency. The ARM architecture supports a small set of instructions, including arithmetic, logical, and data movement instructions. The ARM architecture also includes support for SIMD instructions, which can perform parallel operations on multiple data elements. The ARM architecture has evolved over the years to include new features and instructions, such as virtualization support, security features, and

advanced vector extensions. Traces used in Chapter 6 are generated from an ARM machine. Other examples of RISC architectures include RISC-V [82], an open-source RISC ISA known for its simplicity, modularity, and extensibility. This is why the RISC-V ISA is used in Chapter 5. In conclusion, the CISC architecture advocates for complex instructions that can perform multiple operations in a single instruction, while the RISC architecture advocates for simple instructions that can be executed in a single cycle.

x86 and ARM are the most popular ISAs found in recent processors. Intel and AMD both use x86 ISA in their processors, while ARM designs processors based on a RISC ISA. Having said this, current processor designers are increasingly adopting RISC architectures due to their performance, energy efficiency, and ease of hardware design. The RISC-V architecture has gained popularity as an open-source RISC ISA that offers simplicity, modularity, and extensibility.

As ISA is the interface between the software and the hardware, it plays a crucial role in Chapter 4 where, software information is passed to the hardware by extending the ISA to include a new instruction. The new instruction is then later used in the processor to make several optimizations discussed in chapter 4.

2.2 Out-of-Order (OoO) Processors

Out-of-order (OoO) processors are a type of processor architecture designed to improve instruction-level parallelism and overall performance by executing instructions in an order that maximizes resource utilization. In OoO processors, instructions are dynamically reordered at runtime to execute those that are independent of each other concurrently. This allows the processor to keep its execution units busy even when some instructions are stalled due to dependencies or memory accesses. Figure 1.3 shows a high-level view of the pipeline of an OoO processor. The processor pipeline consists of multiple stages, each responsible for a specific task, such as instruction fetch, decode, execute, and write-back. The pipeline stages are connected by registers that hold the intermediate results of each stage. The pipeline stages operate concurrently, allowing multiple instructions to be processed simultaneously. The processor also uses various structures such as the RoB, LQ, SQ, and SB to manage the out-of-order execution of instructions by keeping a precise state. The processor interacts with the memory hierarchy to fetch instructions to execute. Modern-day processors also consist of a μ-op cache which stores the decoded instructions to reduce the latency of the frontend. Data caches on the other hand are used to store the data that is being operated on by the instructions. Deep cache hierarchy is used to reduce the latency of the memory accesses, as the memory access is the slowest

operation in the processor. If the L1 cache misses the processor fetches the data from the L2 cache and if the L2 cache misses the processor fetches the data from the last level cache (LLC). If the data is not found in the LLC cache the processor fetches data from the main memory.

We will further discuss in detail various relevant components of an OoO processor in the following sections.

2.2.1 Front End

The front end of an OoO processor is responsible for fetching instructions from memory and decoding them into internal μ-ops. Figures 2.2 show a typical frontend of a high-performance processor. The current L1I cache has a fetch latency of around 4 cycles. Thus, if the processor waits for the instruction to be fetched to determine if the branch is taken or not, it would severally limit the frontend. To avoid this current processor implements a decoupled front end where the next address to fetch is separated from the actual fetch of the instruction. This allows the processor to run ahead and generate the address to fetch in parallel with the actual fetch of the instruction. This is known as Fetch Directed Prefetching (FDP) [59].

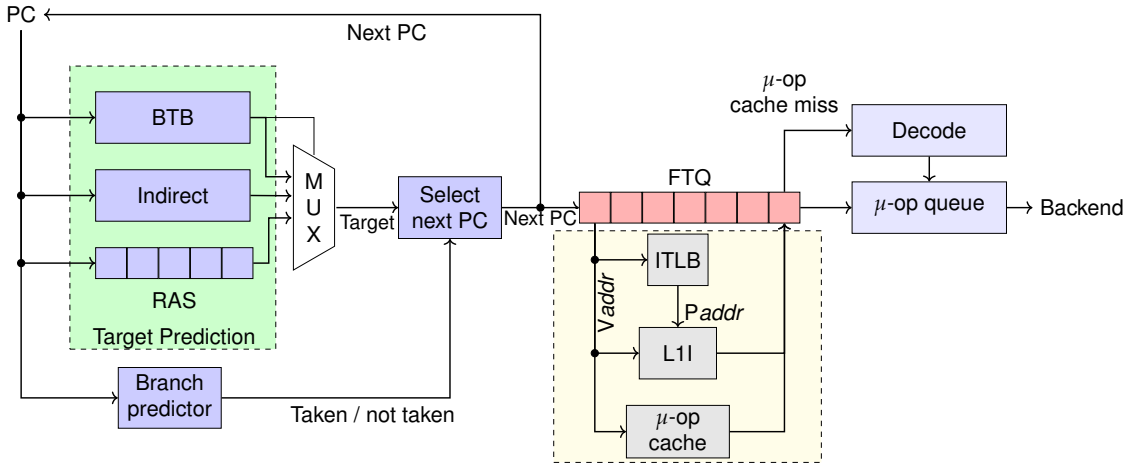


Figure 2.2: frontend of an OoO processor.

2.2.2 Next address prediction

The branch prediction unit (BPU) has the responsibility of predicting the next address to fetch. The BPU consists of target prediction and direction prediction. Target prediction predicts the target address of the branch instruction, while whether the branch is taken or not is predicted by the direction predictor.

Target prediction: To accurately predict the target processor uses a branch target buffer (BTB), an indirect branch target predictor (Indirect Predictor), and the return address stack (RAS). BTB is a cache-like structure that stores the recent target addresses of the branches. Other than branch target info, BTB also holds various information such as the next sequential instruction addresses, if those instructions are a branch or not, and if a branch then its type. An indirect predictor is used to predict the target address of indirect branches. Indirect branches are control flow instructions whose target address depends on the value of the register, which is why the target address is not known until the branch is executed. The RAS is used to predict the return address of a function call. The RAS is a stack that stores the return address of the function call. When a function call is made the return address is pushed onto the RAS. When the function returns the return address is popped from the RAS and used as the target address. When the processor asks for a target prediction all the components are searched in parallel. If the entry is not found in the BTB (BTB miss) no information about the branch is available and thus the branch is predicted as not taken and the next address is the next sequential address. If the entry is found in the BTB the target address is chosen either from BTB, indirect, or from RAS depending on the type of the branch. If the branch type is indirect the target predicted by the indirect predictor is chosen. If the branch type is a return the target predicted by the RAS is chosen. If the branch type is a direct branch the target predicted by the BTB is chosen. As shown in figure 2.2, all generated target addresses are sent to the multiplexer, and the target address is chosen based on the branch type provided by the BTB.

Direction prediction: Apart from the target the processor also needs to predict if the branch should be taken or not. The branch predictor provides the prediction. If the branch is predicted as not taken or the predicted branch target is not available, the processor follows the next sequential address. If the branch is predicted as taken the processor follows the target address.

Re-steering fetch: The current processors can have several KBs of BTB, indirect predictors, and branch predictors. Getting a prediction from such a large structure can take several processor cycles. To overcome this processor complements these

large structures with small but less accurate predictors such that the processor can get the prediction in a single cycle. In this multi-level setting, the second-level predictions are trusted over the first-level predictions. If the prediction from the first level predictor is correct the processor continues to fetch from the predicted target address and predicted direction. If the prediction is mismatched with the prediction from the second-level predictor the processor re-steers the fetch to the target address and direction predicted by the second-level predictor.

The branch prediction is validated at the decode and execute stage, depending on the type of branch. Once a branch miss is detected the processor initiates a pipeline flush and re-steers the fetch to the correct target address. When this happens the FTQ is unable to hide the L1I miss latency as the FTQ is also flushed. We identify that this can severally limit the frontend performance, detailed in Chapter 6.

2.2.2.1 μ-op Cache

The μ-op cache serves as a crucial component in modern-day high-performance processors, facilitating efficient processing by holding recently generated μ-ops, which are micro-operations resulting from the decoding of architectural instructions fetched from the L1I cache. Addresses in the FTQ are used to index either or both the L1I cache and the μ-op cache, depending on the current frontend operating mode. The L1I cache contains recently used encoded architectural instructions, while the μ-op cache holds μ-ops recently generated by decoding instructions fetched from L1I.

The frontend operates in two modes [76]. In *stream* mode, the FTQ only queries the μ-op cache. On a hit, μ-ops are directly sent to the μ-op queue. This represents the fast path and saves power as the L1I, and decoders are bypassed. All the entries from the μ-op queue move to the dispatch queue to be allocated and issued in the processor backend. On a μ-op cache miss, the mode switches to *build* mode. The L1I is then queried to provide instructions that will flow through the decoders to generate μ-ops, before being inserted in the μ-op queue. During this mode, a hardware block builds μ-op cache entries following specific rules that dictate the termination of a μ-op cache entry [45]: (1) A predicted taken branch (2) Crossing an L1I line boundary (3) Exceeding a statically defined number of (a) μ-ops (b) immediate or displacement fields (c) micro-coded μ-ops. The frontend queries both the L1I and the μ-op cache in parallel until encountering consecutive hits in the μ-op cache, prompting a switch back to *stream* mode to save power. L1I hits therefore represent the slow path, as architectural instructions need to be decoded. Furthermore, continuously alternating between the two modes

introduces latency overhead [4, 61].

The μ-op cache was primarily designed for power savings [76], by holding the μ-ops of frequently executed instructions. However, in modern x86 processors, its role goes beyond that. Since decoding multiple x86 instructions in parallel is challenging, decode width remains limited to 4-5 architectural instructions even in aggressive designs. However, the μ-op cache width can exceed this limit at minimal cost, by caching more μ-ops per entry. For instance, AMD Zen4 can provide up to 9 *macro-ops*[1] per cycle from the μ-op cache, while it is limited to decoding 4 architectural instructions per cycle, which yield fewer than 9 *macro-ops* [21]. Therefore, from a performance standpoint, the larger width combined with the shortened frontend length stemming from bypassing decoders makes the μ-op cache an efficient pipeline (re)fill accelerator, as long as the requested μ-ops are found in the μ-op cache. We emphasize that caching μ-ops is not limited to microarchitectures implementing complex instruction sets. For instance, the ARM Neoverse V2 microarchitecture features a 1.5K-entry decoded cache [36].

2.2.2.2 Instruction Fetch

In the event of a μ-op cache miss, the frontend switches to *build* mode, initiating instruction fetching if ports are available in the L1I and Instruction Translation Lookaside Buffer (ITLB). The ITLB stores recent translations of virtual-to-physical memory, facilitating faster access to instruction addresses. Fetching instructions from the L1I, typically virtually indexed but physically tagged, initiates the flow through decoders for μ-op generation before queuing. Despite the parallel querying of L1I and μ-op cache during *build* mode, consecutive μ-op cache hits trigger a return to *stream* mode, emphasizing L1I hits as the slow path requiring architectural instruction decoding and introducing latency overhead during mode alternation [4, 61].

2.2.2.3 Instruction to μ-op Conversion & Macro-op Fusion

Upon fetching, architectural instructions undergo decoding, transforming them into μ-ops before transmission to the backend. For complex architectures like x86, this process entails sophisticated decoding mechanisms to handle the architecture's complexity, variable instruction lengths, and diverse addressing modes [38]. In contrast, RISC architectures like RISC-V feature simpler, fixed-length instructions, reducing decoding complexity and facilitating faster processing. RISC-V

[1] AMD translates x86 instructions to one or more *macro-ops*. *Macro-ops* are therefore decoded instructions.

instructions are of fixed length, typically 4 bytes (2 bytes when compressed ISA is used) [79], which simplifies the decoding process as the decoder can easily identify instruction boundaries without ambiguity. RISC-V also employs a smaller number of addressing modes and simpler instruction formats, which contributes to a more straightforward and faster decoding process. This design philosophy in RISC-V prioritizes efficiency and performance, reducing the need for complex decoding mechanisms and micro-operations. As the μ-op cache already holds decoded μ-ops, the decoding process is not needed in case of a μ-op cache hit. μ-op cache misses, on the other hand, require the instruction to be decoded.

2.2.2.4 μ-op Fusion

Once the instruction is decoded either by the decoder or fetched from the μ-op cache, the μ-ops are sent to the μ-op queue. The μ-op queue is a small queue that holds the μ-ops before they are sent to the backend. The processor uses the μ-op queue to do μ-op fusion, which is the process of combining multiple μ-ops into a single μ-op to reduce the number of instructions that need to be executed. The fused entry then occupies only one entry in RoB, LQ, and SQ. The current processor can fuse several combinations of μ-ops such as load-load, store-store, load-ALU, ALU-ALU, etc. A list of previously proposed fusion pairs is listed in Table 2.1 and an example of a load-load pair is shown in Figure 2.3.

Table 2.1: Several RISC-V fusion idioms envisioned in [19].

add rd, rs1, rs2 *ld rd, 0(rd)*	*lui rd, imm[31:12]* *addi rd, rd, imm[11:0]*
ld rd, imm(rs1) *add rs1, rs1, 8*	*auipc t, imm20* *jalr ra, imm12(t)*
slli rd, rs1, {1,2,3} *add rd, rd, rs2*	*mulh[[S]U] rdh, rs1, rs2* *mul rdl, rs1, rs2*
slli rd, rs1, 32 *srli rd, rd, 29/30/31/32*	*div[U] rdq, rs1, rs2* *rem[U] rdr, rs1, rs*
lui rd, imm[31:12] *ld rd, imm[11:0](rd)*	*auipc rd, symbol[31:12]* *ld rd, symbol[11:0](rd)*
ld rd1, imm(rs1) ***ld rd2, imm+8(rs1)***	***st rs2, imm(rs1)*** ***st rs3, imm+8(rs1)***

The previously proposed fusions have the following rules:-

- Both μ-ops should be consecutive in the instruction stream. This guarantees that both the possible pairs will be next to each other in the μ-op queue.

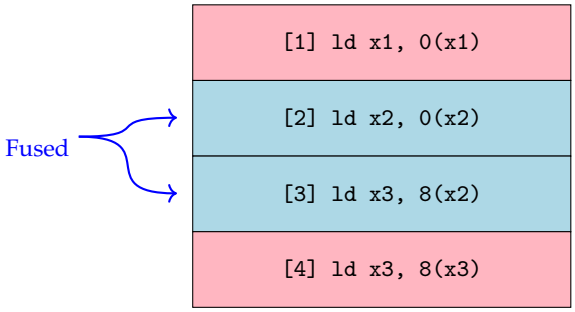


Figure 2.3: Example of μ-op fusion in RISC-V.

This is important as searching the whole μ-op queue for a possible fusion pair would be expensive.

- The memory pairs should access the consecutive memory locations. Due to limited information at the decode stage, the processor can only fuse the memory pairs that are accessing the consecutive memory locations.
- If memory pairs are fused the processor should ensure that both the μ-ops have the same base register.

These conditions severely limit the processor's ability to fuse the μ-ops. The impact of these limitations is discussed in Chapter 5, where we propose a new fusion mechanism that can fuse non-consecutive the μ-ops and eradicate all the 3 conditions mentioned above.

2.2.3 Back End

The back end of an OoO processor is responsible for executing instructions out of order and committing the results in order. Figure 2.4 shows a typical backend of a high-performance processor. The back end consists of the following key components.

2.2.4 Register Renaming

Dynamically scheduled processors exploit instruction-level parallelism by executing instructions out of order. This requires the processor to maintain a mapping between architectural registers (visible to the programmer) and physical registers (used internally by the processor). Register renaming is the process of assigning physical registers to the operands of instructions to resolve register dependencies

and enable out-of-order execution. The register renaming unit maintains a pool of physical registers that are mapped to architectural registers. This mapping allows multiple instructions to use the same architectural register without causing data hazards. The physical registers are allocated and deallocated dynamically as instructions are dispatched and retired. The processor has a separate pool of physical registers for integer and floating-point operations. The decoded μ-ops are sent to the register renaming unit, which assigns physical registers to the operands of the instructions. Register renaming is essential for resolving register dependencies and enabling out-of-order execution.

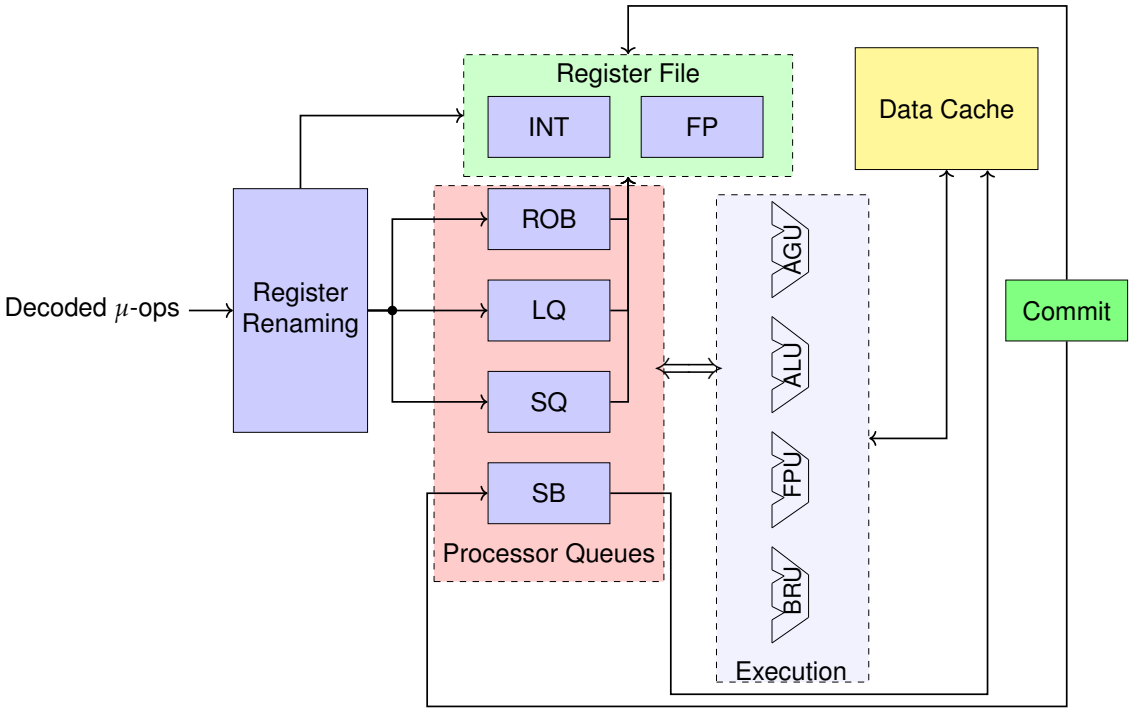


Figure 2.4: Backend of an OoO processor.

2.2.5 Processor Queues

OoO processor requires several queues to manage the execution of instructions. Some are used to maintain the program order while others are used to maintain the memory ordering.

Reorder Buffer (ROB): The ROB is a circular buffer that tracks the speculative execution of instructions and their results. It maintains the program order of instructions and ensures that instructions are committed in order. The ROB contains entries for each instruction in flight, including the instruction's type, source and destination registers, and execution status. The ROB is used to handle exceptions, mispredictions, and memory ordering. When an instruction completes execution, its result is written back to the physical register file, and the corresponding entry in the ROB is marked as ready. Instructions are committed in order from the ROB to ensure precise exceptions and maintain program correctness. The ROB also helps in handling branch mispredictions by allowing the processor to squash instructions following a mispredicted branch and resume execution from the correct path. The ROB is a critical component of the processor's out-of-order execution engine and plays a key role in maintaining program order and correctness, thus it can easily become the bottleneck of the processor as no instruction will be able to execute if the RoB is full.

Load Queue (LQ): The LQ contains all the in-flight load operations. Once a load instruction is issued it is added to the LQ. If the LQ is not free the instruction cannot be issued. LQ is used to ensure the correctness of the loads executing, as loads can be executed out of order. The LQ is required to maintain the correctness in:-

- **Intra-thread:** loads in the processor should read the latest value produced by the store operation to the same address. Waiting for all the stores to finish before executing the load would severely limit the performance of the processor. Thus, processors allow the loads to execute speculatively in the presence of an unresolved store. When the store resolves it searches LQ to find any load to the same address as the store. If the load is found the load is squashed and re-executed. The loads that execute in the presence of an older unresolved store are called data-speculative (D-spec, following the terminology of Duan *et al.* [24]) loads (Figure 2.5).

- **Consistency:** several memory models force that the load-load ordering should be maintained such as in Total Store Order (TSO). Waiting for the previous load to complete before executing the next load would limit the performance of the processor. Thus, the processor allows the loads to execute over older loads. The loads are committed in order, but the load-load ordering can be visible if a remote core writes to the same memory location as the speculative load. Thus, on cache invalidation and evictions, the processor searches the LQ, and if the load is found the load and the subsequent instructions are squashed and re-executed. The loads that

execute over older loads are called memory speculative (M-spec, following the terminology of Duan *et al.* [24]) loads (Figure 2.5 shows the M-spec (InvEv) which requires an LQ search due to cache invalidation or evictions).

- **Coherence:** While the load-load ordering is defined by the programming model, load-load ordering to the same address is universal for any coherent system. When a load is reordered over a load with the same address the same condition as the consistency model is applied as these loads are also M-spec loads. LQ is searched on cache invalidation and evictions. If the load is found the load and the subsequent instructions are squashed and re-executed.

Loads are generally on the critical path of the processor as they appear at the top of the dependency chain. This makes LQ a critical structure in the processor and if the LQ is full it can stall all the dependent instructions.

Store Queue (SQ) & Store Buffer (SB): Many high-performance processors relax the store-load to improve performance. To support this the processors, require SQ and SB. In actual implementations, the SQ and SB are a single physical structure and the division between them is a pointer that separates the two structures. Stores are allocated in the SQ at the issue stage and after committing the entry *moves* to the SB. From SB the store writes to the cache in order. The SB allows the processor to hide the long latency store operations by allowing the processor to perform the store operations out of the processor's critical path. This causes load instructions to be effectively reordered concerning store instructions, thus relaxing the consistency model semantics. Thus, when a load executes it searches the SQ/SB to find the latest store to the same address. If the store is found the value is forwarded to the load. While the loads are executed speculatively the TSO semantics preserve the order of store operations with respect to other stores and of load operations with respect to other loads. However, to achieve memory-level parallelism, in practice, loads are speculatively reordered with respect to other loads. Stores, on the other hand, perform in-order based on the observation that stores are not on the processor's critical path. Unfortunately, when a store operation misses in the cache, all subsequent stores block until the miss is resolved and the store performs. This creates severe bottlenecks in current OoO processors.

2.2.5.1 LQ searches in SMT Processors

In SMT (Simultaneous Multithreading) core each hardware thread only has visibility of its logical LQs and SBs. This configuration is typical in the current

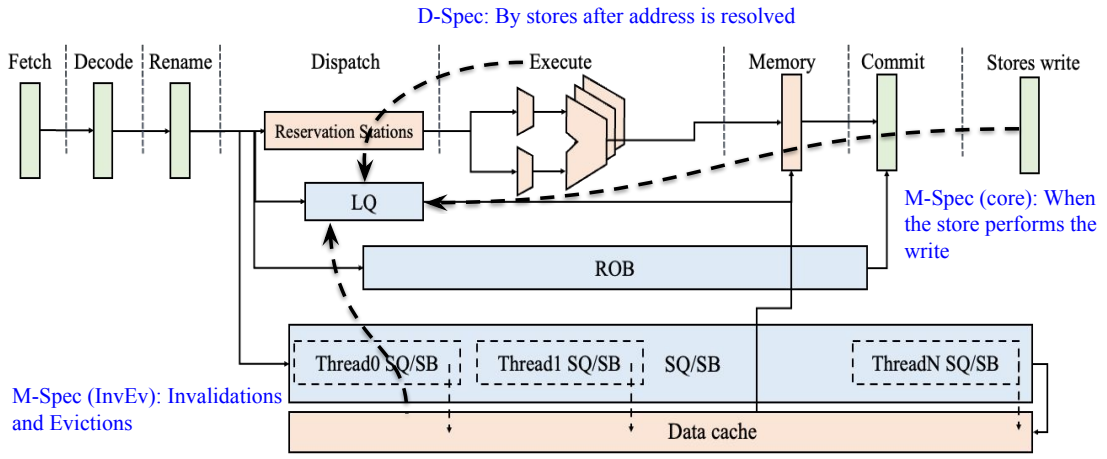


Figure 2.5: LQ Searches

processor which employs SMT to improve performance. In this case, speculative reordering that violates load-load ordering within one thread could be influenced by stores executed by another hardware thread within the same core [25]. However, by default, coherence invalidations from the second thread's stores are not applied to the first thread's LQ, since both threads share the same coherent state of cache lines in the L1D cache. Thus, in SMT implementations, the processor needs a solution to trigger an LQ search in each thread whenever a store writes to the cache from the SB. The LQ search is an additional LQ search required in SMT processors and is performed in parallel with the store writing to the cache. If a match is found in the LQ the load and the subsequent instructions are squashed. As these loads can expose the load-load ordering it is also considered as M-spec loads (Figure 2.5).

2.3 Parallel Programming Models and Synchronization Constructs

Parallelism is a key aspect of modern processors. Current programming models such as OpenMP (Open Multi-Processing) [54], MPI (Message Passing Interface) [50], and CUDA [53] provide constructs for expressing parallelism in programs. Compilers analyze the program code to identify parallel regions and generate code that can be executed in parallel on multi-core processors or accelerators. For example, OpenMP provides support for creating multi-threaded applications. Using OpenMP the programmer can create various threads that can run in parallel and perform different tasks. MPI on the other hand is used

to create distributed memory parallel applications. MPI allows the programmer to create multiple processes that can run on different nodes and communicate with each other using messages. CUDA is a parallel computing platform and programming model developed by NVIDIA for general-purpose computing on GPUs. CUDA allows the programmer to write code that can be executed on the GPU to accelerate computation. All parallel programming languages and models also provide various synchronization constructs to manage the interaction between parallel threads or processes. These constructs ensure that data is shared correctly between threads and that the program executes correctly in parallel.

When two threads access the same memory location concurrently and at least one of the threads is modifying the memory location, a memory conflict or a data race can occur. To prevent such scenarios, synchronization constructs such as locks are used. A lock is a synchronization primitive that allows only one thread to access a shared resource at a time. When a thread acquires a lock, it gains exclusive access to the code (a critical section), and other threads are blocked from executing the same code section until the lock is released. As long as all the memory accesses are protected by locks, no conflict or data race can occur. Another synchronization construct is the barrier, which is used to synchronize the execution of multiple threads. A barrier is a synchronization point that forces all threads to wait until all threads have reached the barrier before proceeding. Barriers are used to ensure that all threads have completed a certain phase of execution before moving on to the next phase. Similarly, signal-wait constructs are used when threads are executing in a producer-consumer pattern. The producer thread signals the consumer thread when data is available, and the consumer thread waits for the signal before consuming the data. Both barriers and signal-wait are synchronization points, used to coordinate the execution of multiple threads and provide a happens-before relationship between the threads.

2.4 Compilers

To effectively translate the parallel constructs into executable code, compilers play a crucial role in the development of parallel applications. Code written by programmers and developers is in high-level languages such as C, C++, Java, etc. These high-level languages are designed to be human-readable and easy to write, but they need to be translated into machine code for the processor to execute. This translation is done by a compiler, which converts the high-level code into a low-level representation that the processor can understand. The translation depends on the ISA of the processor. The compiler generates machine code that adheres to the ISA of the target processor, ensuring that the program executes

correctly and efficiently on the processor. The compiler's task is not only to translate the code but also to optimize it for better performance and efficiency. To achieve this the compiler performs various optimizations at various stages of the compilation process. Modern day compilers work across several stages such as frontend, intermediate representation, optimization, and backend to generate efficient machine code.

LLVM [47] is a popular compiler infrastructure that provides a set of reusable libraries and tools for building compilers. The LLVM infrastructure has been widely adopted in industry and academia, with many compilers and tools built on top of it. LLVM is designed to be modular and extensible, allowing developers to build custom compilers for different languages and target architectures. A high level of compilation pipeline of LLVM is shown in Figure 2.6. LLVM uses an intermediate representation (IR) called LLVM IR, which is a low-level representation of the program that is independent of the source language. The LLVM IR is used for optimization and code generation before being translated into machine code for the target processor. LLVM provides a wide range of optimization passes that can be applied to the IR to improve the performance and efficiency of the generated code. These optimizations include instruction scheduling, loop optimization, data flow analysis, inlining, register allocation, etc. LLVM also supports Just-In-Time (JIT) compilation, allowing programs to be compiled at runtime for improved performance. The optimizations are applied to the IR through a series of passes. The passes are divided into three categories: analysis, transformation, and utility. The analysis pass gathers information about the program, such as control flow, data flow, and dependencies. The transformation passes modify the program to improve performance, reduce code size, or fix errors. The utility passes provide support for other passes, such as debugging and profiling. The passes are executed in a specific order to ensure correct and efficient optimization of the program. Once the optimizations are applied, the IR is translated into machine code for the target processor by the backend of the compiler.

2.5 Construction of DRF regions using LLVM

Understanding how to effectively manage synchronization in parallel programs is crucial for achieving high performance and correctness. One of the foundational concepts in parallel computing is the idea of memory consistency models, which define how memory operations appear to execute different threads. Sequential consistency (SC) is the most intuitive memory model, providing a straightforward approach where operations appear to execute in a strict sequential order.

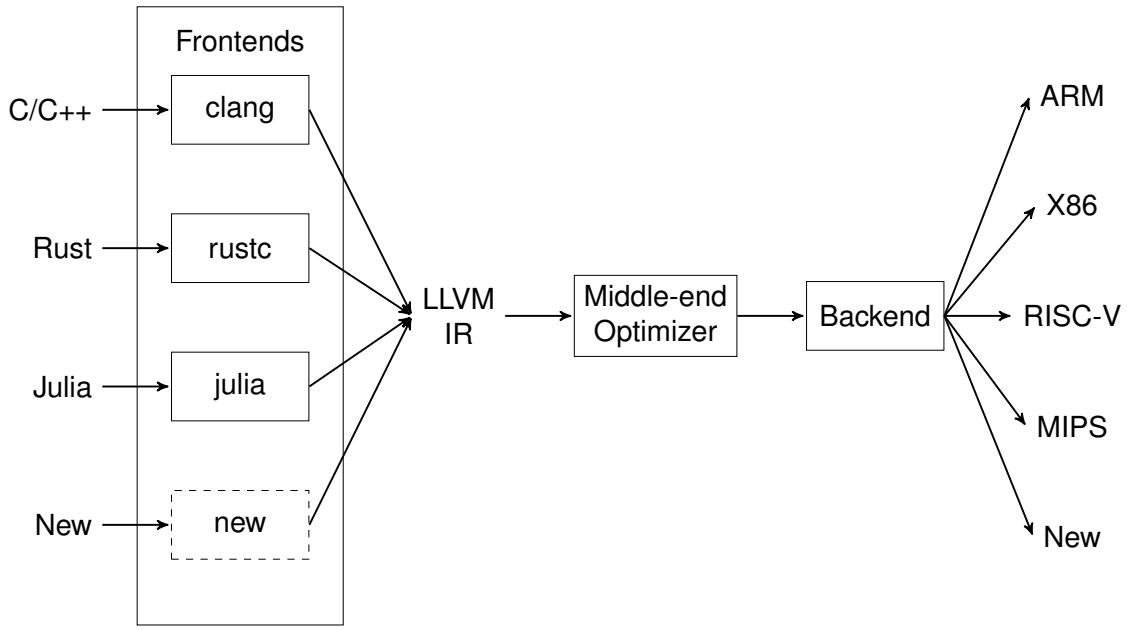


Figure 2.6: High-level overview of LLVM compilation pipeline

However, SC is also restrictive, often leading to performance penalties. To mitigate this, many modern systems adopt weaker memory models that allow for more flexibility and higher performance at the cost of increased programming complexity.

To balance performance and ease of programming, the SC-for-DRF (Data Race Free) model has been proposed [12,28]. This model guarantees sequential consistency only for programs that are data race free, allowing optimizations in both software and hardware. This approach is particularly relevant for widely-used programming languages like Java, C, and C++, which adhere to the SC-for-DRF model, requiring that data race-free programs behave as though they are sequentially consistent.

Within this context, LLVM plays a pivotal role in the construction and identification of DRF regions within programs. A DRF region, or synchronization-free region, is a block of code where data races are guaranteed not to occur, either due to the access patterns (different threads accessing different memory locations) or synchronization mechanisms ensuring exclusive access. Conversely, code outside these regions, including synchronization points, constitutes non-DRF (nDRF or sync) regions. Figure 2.7 illustrates an example of a parallel program with clearly marked DRF and sync regions. Stores A, C, F, D, and E are within DRF regions, allowing them to run concurrently without data races, while store B is in a sync

region, constrained by synchronization to ensure correct execution.

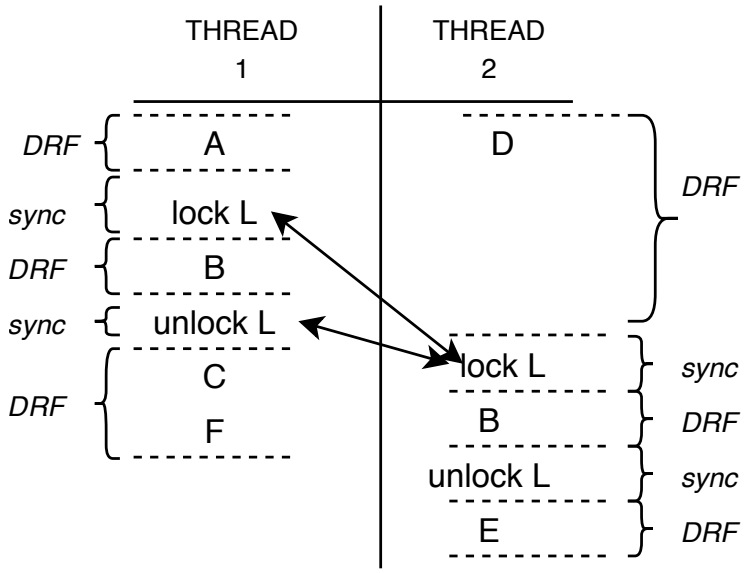


Figure 2.7: Example code showing DRF regions.

The construction of DRF regions is implemented as a compiler pass in LLVM, utilizing a state-of-the-art pointer analysis tool [77] to improve the accuracy of alias analysis within nDRF regions. The process begins by identifying nDRF regions, such as critical sections, barriers, and signal-wait constructs, and building a control-flow graph (Sync-CFG) among them. The first reachable nDRF region in each thread function is marked as an entry nDRF region using a depth-first search (DFS) of the Sync-CFG. Upon completion of the analysis, the compiler inserts a dedicated instruction, designated as *setDRF val*. The *val* is set to 1 to indicate the commencement of the DRF regions and 0 to indicate the end of the DRF regions.

In chapter 4, the DRF pass is employed to identify DRF regions in parallel programs with the objective of optimizing the LQ and SB of the processor. This is achieved by utilizing the DRF pass, which is a technique for identifying regions of interest in a given program.

Chapter 3

Methodology

In this chapter, we outline the methodology employed in this thesis. First, we provide an overview of the simulators utilized in the study. Next, describe the microarchitectural models used in the analysis. Following that, we discuss the benchmarks and traces used for the experiments. Lastly, we present relevant metrics used to evaluate the performance of the proposed techniques.

3.1 Simulators Used

This thesis uses two different simulators to evaluate the proposed techniques. The first simulator is an in-house simulator that models detailed pipeline stages and processor backend. Due to this reason, this simulator was used in Chapter 4 and 5. The second simulator is the ChampSim [5] which is a very popular simulator in the computer architecture community specifically for prefetching, thus this was used for Chapter 6. Below we describe both simulators in detail.

gems4proc: gems4proc models a seven-stage pipeline as described by Gonzalez et al. [32]. The simulator can be configured to utilize Sniper [18] for x86 and Spike [62] for RISC-V, thereby providing the processor model with the instructions to be executed. Additionally, GEMS [49] is employed to model the memory hierarchy and cache coherence, utilizing a standard invalidation-based directory protocol. Furthermore, GARNET [13] is used to model the interconnect. Besides, the simulator can conduct simulations of multi-core processors and SMT cores. The simulator can model a variety of memory and consistency models. In the context of SMT modeling, the simulator can be configured to utilize a range of scheduling algorithms, including round-robin and fair-share.

ChampSim: ChampSim is an open-source, trace-based simulator that is widely employed in the field of computer architecture research, particularly for the purpose of studying various prefetching techniques. It is developed over the simulation environment used for the Second Data Prefetching Competitions [2]. ChampSim is capable of modeling a variety of modern processors. Most of the processor configurations such as CPU core, RoB size, LQ size, etc can be configured using a configuration file. ChampSim also contains a DRAM model that is configurable with different banks and bus contention. As the traces only contain virtual addresses ChampSim simulates a page table and a TLB to convert the virtual address to a physical address. Prefetchers in ChampSim can be configured for each cache level and can be turned on or off and as most of the recently proposed prefetchers are implemented in ChampSim, it is widely used for prefetching research.

3.2 Simulation Methods

This section provides details of various simulation parameters used for each chapter including the μ-architectural models and various benchmarks used.

Chapter 4: Chapter 4 of this thesis combines two proposals: one aimed at optimizing the SB and the other focused on optimizing the LQ. Both optimizations are made possible by the DRF compiler, as explained in Chapter 2. The baseline and the proposed optimizations are implemented in gems4proc simulator.

To evaluate our proposal to optimize SB, we mimic an Intel Skylake micro-architecture employing macro-op and μ-op fusion. The model employs a single circular queue for both the store queue and store buffer that utilizes better the resources as done in Intel architectures. We run both parallel applications from the Splash-3 [66] benchmark suite, which is a data-race-free version of the original Splash2 benchmark suite released before the pthreads memory model was updated to enforce SC-for-DRF, and the PARSEC 3.0 [17] benchmark suite, which is a popular modern suit that complies with the C++ standard and therefore enforces SC-for-DRF. We use the simmedium inputs for barnes, blackscholes, cholesky, dedup, fft, fluidanimate, lu_cb, and lu_ncb and the simsmall inputs for fmm, ocean_cp, oceanncp, radiosity, radix, raytrace, streamcluster, swaptions, volrend, water_nsquared, and water_spatial.

To assess the effectiveness of our proposal for the LQ optimization, we model detailed LQ and SQ, including the searches for the speculative support for memory ordering. We simulate a multi-core processor, with eight 2-way SMT cores, providing a TSO consistency model. The processor parameters, shown

Table 3.1: Parameters for Chapter 4 (SB)

Processor	
Processor Model	Intel Skylake [23]
Fetch Width	5 instructions
Issue Width	8 ports
Allocation Queue	97 entries
Reorder Buffer	224 entries
Load Queue	72 entries
Store Queue + Store Buffer	56 entries
Memory	
Private L1 I&D caches	32KB, 8 ways, 4 hit cycles, pipelined
L1 prefetcher	Stride, degree 3
Private L2 cache	256KB, 8 ways, 12 hit cycles
Shared L3 cache	1MB per bank, 8 ways, 35 hit cycles
Directory	8 ways, 200% coverage of L2
Memory access time	160 cycles
Network Topology	2D Mesh

in Table 3.2, are chosen after modern processors and resemble the Intel Alder Lake micro-architecture. Following Intel's SMT implementations [22], the ROB, LQ, and SQ are statically partitioned among the threads, assigning a fraction of the structures to each thread, while the execution units are dynamically shared. We evaluate parallel workloads from the Splash-3 [66] and PARSEC 3.0 [17] benchmark suites, as well as six fine grain synchronization-intensive benchmarks [31, 44]. All the applications comply with the C/C++ standard, and thus, enforce SC-for-DRF. Results correspond to the parallel regions of the applications.

Chapter 5: For the evaluation of our proposal on instruction fusion, we use gems4proc simulator which models 8-wide Fetch and Decode stages to ensure that the Allocation Queue gets filled even in high IPC workloads. Our model implements a TSO consistency model, thereby being compliant with the RISC-V TSO extension (*Ztso*). The high-level characteristics of the system used in our simulations are displayed in Table 3.3. We evaluate our proposal with the SPEC CPU 2017 and MiBench benchmark suites, which are widely used desktop and embedded workloads, respectively. We skip the Linux kernel boot and setup for all the applications. Then, SPEC applications skip an additional 10B instructions

Table 3.2: Parameters for Chapter 4 (LQ)

Processor	
Processor Model	Intel Alder Lake [64]
Cores and threads	8-core 2-way SMT
Fetch Width	6 instructions
Issue/Commit Width	12 instructions
Reorder Buffer	512 entries
Load queue	192 entries, 3 write, 2 search ports
Store queue	128 entries, 2 write, 3 search ports
Branch predictor	TAGE-SC-L [72]
Mem. dep. predictor	StoreSet [20]
Memory	
Private L1I cache	32KB, 8 ways, 4 hit cycles, pipelined
Private L1D cache	48KB, 12 ways, 5 hit cycles, pipelined, IP-stride prefetcher
Private L2 cache	1MB, 8 ways, 12 hit cycles
Shared L3 cache	4MB per bank, 16 ways, 35 hit cycles
Directory	8 ways, 200% coverage of L2
Memory access time	160 cycles
Network Topology	2D Mesh

and report results for the next 500M instructions. MiBench applications run until completion. SPEC workloads run using reference inputs while MiBench workloads use the large input set. The binaries were compiled with GCC 10.2.0 targeting the RV64G ISA with flags -O3 -static.

Chapter 6: We used *develop* branch of ChampSim [5][1] to evaluate our proposal related to processor front-end. The ChampSim version used includes a detailed frontend model implementing fetch-directed prefetching (FDP) [60], a branch target Buffer (BTB), indirect target predictor, return address stack (RAS), and conditional branch predictor. L1I prefetch requests issued through FDP are demand accesses and, therefore, we do not consider them as prefetch requests. That is, we assume a given address in FTQ checks the L1I tags a single time and fetches the instruction bytes, as opposed to checking it once for prefetching and a second time when it reaches the head of the FTQ as a demand request. We extend ChampSim's standard µ-op cache design to reflect the frontend

[1]Commit `c8eff1dafdb398fcb9a40c95994cb202d831d678`

Table 3.3: Parameters for Chapter 5

Processor	
Model	Intel Icelake [55]
Predictors	L-TAGE [69], Store-set [20]
Stages	Fetch/Decode/Rename/Allocation /Issue/Execution/Memory/Commit
Frontend Stages	8-wide Fetch/Decode, 5-wide Rename
Allocation Queue	140 entries
Backend Stages	5-wide Alloc., 10x Exec. Ports, 2x loads 2x stores, 4x AGU 4x ALU, 1x DIV, 2x FP Add/Sub 1x SQRT, 20-wide Commit
ROB/IQ/LQ/SB	352/160/128/72 entries
Memory	
L1I	32KB, 8 ways, 4-cycle hit lat., pipelined
L1D	48KB 12 ways, 4-cycle hit lat., pipelined
L1D prefetcher	Stride, degree 3
L2	256KB, 8 ways, 12 hit cycles
LLC	8MB, 8 ways, 35 hit cycles
RAM	160-cycle latency

described in Section 2. Specifically, our frontend works either in *stream* mode or in *build* mode, paying a 1-cycle penalty when switching modes [61]. The µ-op cache entries follow all termination conditions discussed in Section 2.2.2.1. We modify the ChampSim's frontend to support the generation of addresses on the alternate path in parallel with the predicted path. The L1I is even/odd interleaved so that basic blocks spanning two cache lines can be retrieved in a single cycle. Interleaving also enables sharing the L1I tag lookup bandwidth between the predicted path and alternate path at no extra cost over the baseline. The baseline µ-op is dual ported, and its tag arrays are even/odd interleaved in UCP. We accurately model the port contention of BTB, L1I, and µ-op cache, by generating the instruction addresses on the wrong path. The processor and memory hierarchy are configured following the specifications of Intel's latest Alder Lake performance core. The primary parameters are listed in Table 3.4.

Table 3.4: Parameters for Chapter 6

Out-of-order	
Processor Model	Intel Alder Lake [64]
Branch prediction	64K-entry 16-bank instruction BTB [58] LRU, 64KB ITTAGE [70], 64-entry RAS, 64KB TAGE-SC-L [72]
µ-op cache	4Kops, 64 sets, 8 ways, 8 µ-op/entry, 1-cycle hit, LRU [43,45], 2 ports
Frontend Stages	Up to 16 sequential addresses predicted per cycle, 16-wide fetch, 6-wide Decode, 6-wide Dispatch, 192-entry FTQ, 32-entry decode buffer, 32-entry dispatch buffer
Backend Stages	10-wide Execute, 3x load, 2x stores, 10-wide Commit, 512-entry ROB, 192-entry LQ, 114-entry SB
Memory	
ITLB	256 entries, 8 ways, 1-cycle hit, 8-entry MSHR
DTLB	96 entries, 6 ways, 1-cycle hit, 8-entry MSHR
STLB	2048 entries, 16 ways, 8-cycle hit, 16-entry MSHR
L1I	32KB, 8 ways, 4-cycle hit, 32-entry PQ, 16-entry MSHR, LRU, 2 banks
L1D	48KB, 12 ways, 5-cycle hit, 16-entry MSHR, IP-stride prefetcher, 8-entry PQ, LRU
L2	1.25MB, 20 ways, 10-cycle hit, 32-entry MSHR, LRU
LLC	30MB, 12 ways, 40-cycle hit, 64-entry MSHR, LRU
DRAM	2-channel, 8-bank, t_{RP}: 12.5ns, t_{RCD}: 12.5ns, t_{CAS}: 12.5ns

3.3 Metrics

This section describes the metrics used to evaluate the performance of the proposed techniques in this thesis.

Execution Time: Execution time is the time taken by the processor to execute a given workload. Execution time is generally preferred when evaluating multi-thread or multi-core as the total execution time includes not only the processing time on each core but also the overhead of managing parallel tasks, inter-core communication, and synchronization. This provides a holistic view of system performance. Improvements in the execution time is used in Chapter 4 as it uses parallel workloads.

IPC: Instructions per cycle (IPC) is a metric used to evaluate the performance

of a processor. It is calculated as the ratio of the total number of instructions executed to the total number of cycles taken to execute those instructions. IPC is calculated as follows:

$$\text{IPC} = \frac{\text{Total Instructions}}{\text{Total Cycles}} \tag{3.1}$$

Higher IPC means that the processor can execute more instructions in a given cycle, thus improving the overall performance of the processor. IPC is mostly used with single-core and single-thread processors as IPC is only meaningful when the total work to be done is constant. IPC is used as a performance metric in Chapter 5 and Chapter 6 as they use single-thread and single-core workloads.

Predictor Accuracy & Coverage: Accuracy and coverage are two important metrics used to evaluate the performance of any predictors. Accuracy is the percentage of correct predictions made by the predictorCoverage is the percentage of events for which the predictor makes a prediction. The accuracy and coverage of a predictor are calculated as follows:

$$\text{Accuracy} = \frac{\text{Correct Predictions}}{\text{Total Predictions}} \tag{3.2}$$

$$\text{Coverage} = \frac{\text{Total Predictions}}{\text{Total Events}} \tag{3.3}$$

Accuracy symbolizes the effectiveness of the predictor; higher accuracy means that the predictor can make correct predictions most of the time it predicts. Similarly, coverage symbolizes that the predictor can cover most of the events where a prediction is required. Coverage includes both correct and wrong predictions.

Cache Hit Rate: The cache hit rate is the percentage of cache accesses that result in a hit. It is calculated as follows:

$$\text{Cache Hit Rate} = \frac{\text{Cache Hits}}{\text{Total Cache Accesses}} \tag{3.4}$$

The cache hit rate is an important metric to evaluate the performance of the cache. A higher cache hit rate means that the cache can serve most of the requests from the cache itself, thus reducing the latency involved in fetching data from the main memory.

Miss per Kilo Instructions (MPKI): MPKI is a metric used to evaluate the performance of a cache. It is calculated as the ratio of the total number of misses to the total number of instructions executed, scaled by a factor of 1000. MPKI is

calculated as follows:

$$\text{MPKI} = \frac{\text{Total Misses}}{\text{Total Instructions}} \times 1000 \tag{3.5}$$

MPKI is used to evaluate the efficiency of the cache. A lower MPKI means that the cache is efficient in serving most of the requests from the cache itself, thus reducing the latency involved in fetching data from the main memory.

Geomean vs Average: Geometric mean (geomean) is a metric used to calculate the central tendency of a set of numbers. It is calculated as the nth root of the product of n numbers. Geomean is preferred over the arithmetic mean (average) when the numbers are in different orders of magnitude. Geomean is calculated as follows:

$$\text{Geomean} = \sqrt[n]{x_1 \times x_2 \times \ldots \times x_n} \tag{3.6}$$

Average is the sum of all the numbers in a set divided by the total number of elements in the set. The average is calculated as follows:

$$\text{Average} = \frac{x_1 + x_2 + \ldots + x_n}{n} \tag{3.7}$$

In the thesis, performance results such as IPC and execution time are reported as geomean whereas the accuracy, coverage, MPKI, and cache hit rate are reported as average.

3.4 Energy estimation tool

For this thesis we used CACTI (Computer-Aided Design of Interconnects and Caches Tool) [48] which is a widely used tool for modeling the power, area, and timing of cache memories and other processor structures in microprocessors. Originally developed by HP Labs, CACTI has become a standard tool in computer architecture research for estimating energy consumption and performance of cache designs. It is particularly useful for architects looking to explore different memory configurations and understand their impact on system-level energy efficiency. CACTI allows users to model various types of memory components, including caches, main memories, and interconnects, at various levels of detail. The tool simulates both dynamic and static power consumption, considering several factors such as cache size, associativity, line size, and technology node. By modeling these parameters, CACTI provides detailed estimates of the energy expenditure required for different memory operations, such as reads, writes, and idle states. This capability is crucial for architects who need to balance perfor-

mance with power efficiency, especially in energy-constrained environments like mobile and embedded systems.

In this thesis the energy stats presented in Chapter 4, we use a 22nm technology node for our simulations. The LQ and SQ are modeled as a CAM and use the high-performance (hp) model. To estimate the overall energy consumption, we first calculate the energy required per access which includes reading, writing, and searching. After this, the per access energy is multiplied by the total number of accesses to get the total energy consumption.

Chapter 4

Store Buffer and Load Queue Optimization

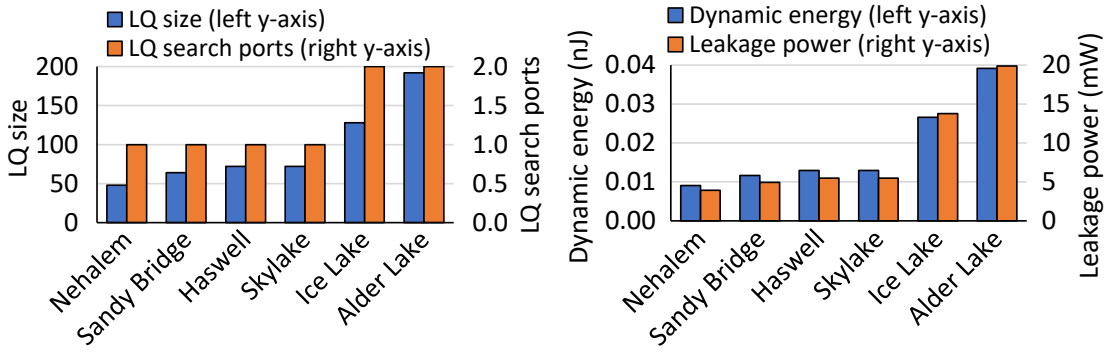
In this chapter, we propose optimizing both the store buffer and the load queue. We identify that the compiler can mark regions where the store-store ordering restriction can be relaxed. Additionally, the search required in the load queue can be avoided by using the same compiler information. The compiler identifies code regions that are safe to be relaxed, and this information is then passed to the processor. Once the processor has this information, it can relax the store-store ordering restriction and skip the load queue search when it is safe to do so.

4.1 *Research Problem 1:* Store ordering in TSO

TSO semantics preserve the order of store operations with respect to other stores and of load operations with respect to other loads. However, to achieve memory level parallelism, in practice, loads are speculatively reordered with respect to other loads [30]. Stores, on the other hand, perform in-order based on the observation that stores are not on the processor's critical path. Unfortunately, when a store operation misses in the cache, all subsequent stores block until the miss is resolved and the store performs. This creates severe bottlenecks in current out-of-order (OoO) processors.

4.2 *Research Problem 2:* LQ searches in SMT

As discussed in Section 2.2.5.1, the LQ is one of the most critical structures in a processor in terms of performance and energy [29]. It needs to keep all in-flight loads in order and support priority searches which, for performance reasons, are done associatively. Furthermore, it is searched frequently: each time a store executes, in order to safeguard sequential semantics [52], and on any invalidation or cache eviction, to enforce the load→load order [30]. In addition to the high contention on its search (i.e. snoop) ports, the LQ also stalls out-of-order (OoO) processors when it becomes full. Consequently, as depicted in Figure 4.1a, both LQ size and search ports have increased over the last few years. For example, Intel processors have increased the LQ size from 48 entries in the Nehalem microarchitecture [39] to 192 in the Alder Lake [65]. The number of search ports also grew in Ice Lake [56], adding a second search port to the LQ.



(a) LQ size and search ports

(b) Dynamic energy per search and leakage power

Figure 4.1: Evolution of LQ characteristics across different generations of Intel processors.

However, the increase in LQ size and search ports comes with a high energy cost. The LQ is a power-hungry and latency-sensitive structure as it needs to support fast associative searches on addresses [29]. Figure 4.1b shows the rise in energy consumption per search and leakage power as the LQ size and search ports increase. The energy consumption of the LQ is dominated by searches and almost quadrupled from Skylake to Alder Lake. Similarly, leakage power quadrupled across the same microarchitectures. The simultaneous multithreading (SMT) paradigm, nowadays adopted by AMD and IBM in their high-performance processors, further exacerbates the criticality of the LQ. SMT enables a core to

execute multiple threads simultaneously, while sharing most of the pipeline resources, including the LQ. Moreover, SMT threads also share the coherent state of the cache lines in the L1D, requiring additional LQ searches to prevent an SMT thread from exposing a speculative load→load reordering to another thread co-running in the same core [26]. Namely, on each store, a thread must search the LQs of the co-running threads in the SMT core when writing from the store queue (SQ) to the L1. This almost doubles the energy consumption of the LQ and the search ports contention since, in an SMT, every store needs to search the LQ twice: when it executes and when it writes to the cache.

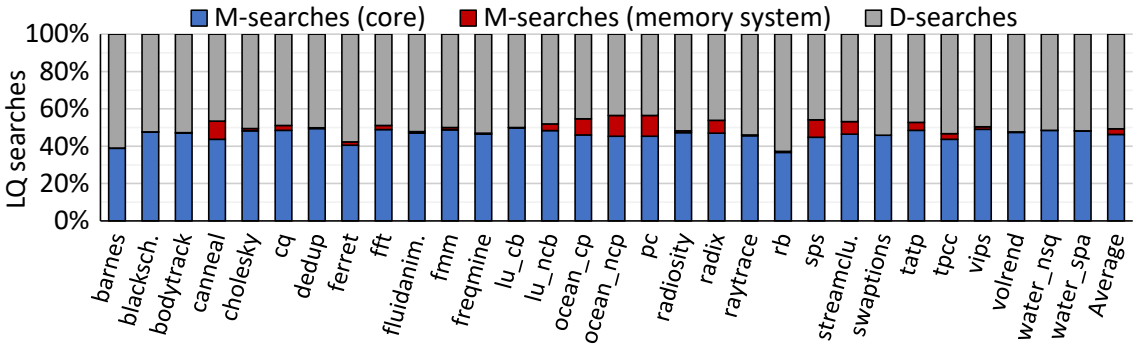


Figure 4.2: Percentage of LQ searches due to memory consistency events and memory dependencies.

Figure 4.2 shows the percentage of LQ searches that are required either to maintain the correct load-load ordering (labeled as *M-searches*, as they target M-speculative loads) or to enforce memory dependencies (labeled as *D-searches*, as they target D-speculative loads). M-searches are further divided depending on their source: *M-searches (Memory system)* are triggered after invalidations or evictions that reach the private caches, while *M-searches (Core)* are triggered on the core's LQ when stores write from the SQ to the L1. M-searches range from 37% (*rb*) to 57% (*ocean_ncp*) depending on the application and, on average, represent 49% of the LQ searches. Since i) most stores are DRF and ii) frequently the LQ contains just DRF loads, an overwhelming majority of the LQ M-searches are superfluous and can be avoided by conveying the DRF information to the processor. Alleviating these searches turns to lower LQ energy consumption and search port contention.

4.3 *Insight 1:* Taking advantage of compiler information

In SC-for-DRF, potentially racy accesses must be guarded by synchronization, entailing that they will execute sequentially. The regions of code delimited by synchronization operations, denoted in this thesis as *DRF regions*, offer the guarantee that writes do not target the same memory location as other concurrent memory accesses (see Fig. 4.3). Consequently, the memory operations performed by a thread during a DRF region remain "invisible" to the other threads until the end of the DRF region. This guarantee allows performing memory accesses in any order during DRF regions as long as sequential semantics are respected. DRF region boundaries ensure that memory accesses perform and become visible to the other threads at the end of the region. Thus, boundaries include not only synchronization operations but also fences, which prevent the reordering of memory operations across them.

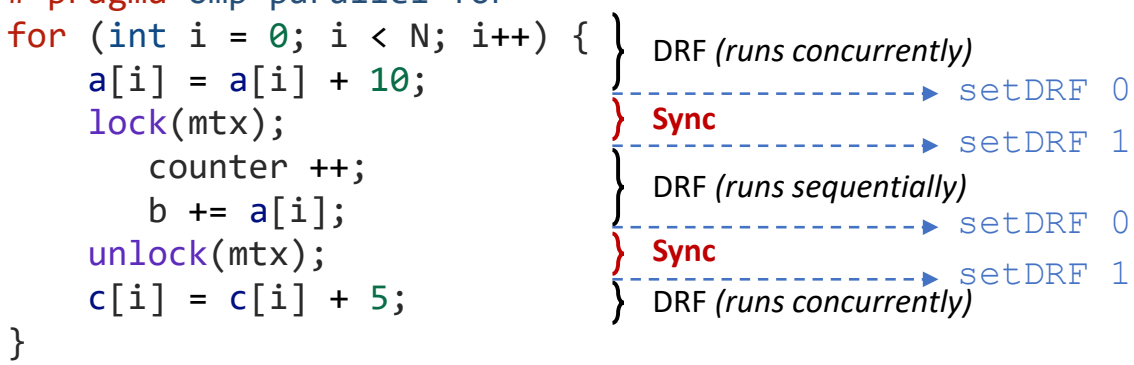


Figure 4.3: Example of code showing a parallel SC-for-DRF program and the delineated DRF regions and sync operations.

4.4 *Insight 2:* Stores can be virtually re-ordered in DRF regions

If processors had information about the nature of the stores residing in the store buffer (stemming from either DRF or sync regions), they could orchestrate the in-order versus out-of-order execution of store operations, without relaxing the consistency guarantees. Figure 4.4 walks the reader through an example illustrating the benefits of performing stores out-of-order (as we propose in this

thesis – ROOW) in contrast to enforcing their order across the entire store buffer (as implemented in standard TSO). Each row represents the content of the store buffer and the status of each store. All stores *A*, *B*, *C*, and *D* belong to the same DRF region, hence can be performed out-of-order as they have exclusive access to the target memory location. In ROOW, they all start as soon as they enter the store buffer (on the right), which means *B* can complete before *A* (which encounters a cache miss) and similarly *D* can complete before *C*. This parallelism hides the miss latency as the processor does not wait for the miss to be resolved, as in the standard TSO store buffer implementation. At the end of the DRF region the other cores observe that all stores completed, but not the order in which they have been performed.

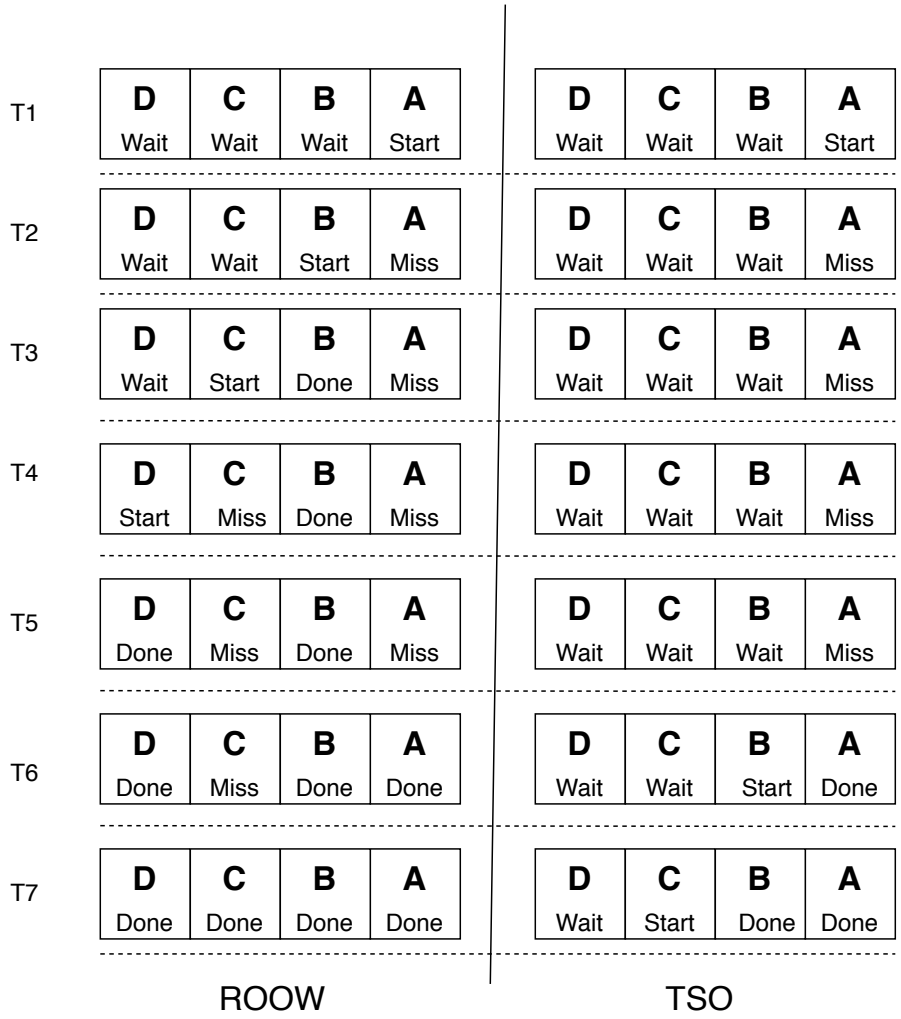


Figure 4.4: Benefits of out-of-order execution of store operation

4.5 *Insight 3:* Pressure on the LQ can be alleviated using compiler information

Since DRF stores are guaranteed to be executed in the absence of concurrent loads to the same address in other threads, they cannot expose consistency violations (e.g., load→load). Therefore, it is not necessary to perform any search in the same core LQ (to squash M-spec loads from other SMT threads) when a DRF store writes. Store writes can also cause invalidations that reach other cores private caches. An invalidation request caused by the writing of a DRF store cannot expose a consistency violation either. Likewise, DRF loads are also guaranteed to be executed in the absence of concurrent stores to the same address in other threads and thus, they cannot expose consistency violations while being executed out of order. Therefore, no LQ search is required on store writes, cache invalidations, and cache evictions when the LQ contains only DRF loads, as they execute non-M-speculatively by definition. On the contrary, consistency violations are possible for racy loads, which may execute M-speculatively and, therefore, based solely on the DRF status of loads, LQ searches are required to prevent exposing any consistency violation. Thus, the LQ search can be avoided when i) it is caused by the writing of a DRF store from a thread running in the same SMT core, or ii) there are no sync loads in the LQ. Since DRF memory operations are far more frequent than non-DRF ones, most LQ searches can be safely avoided, reducing energy consumption in the LQ and contention in its search ports.

Along with the observations above, the loads can be removed from the LQ as soon as they do not need to be searched. Removing loads from the LQ helps reduce LQ occupancy, which either i) eliminates LQ-induced stalls and thus improves performance or ii) allows shrinking the LQ and thus reduces its energy consumption and area.

4.6 *Proposal 1:* Conveying compiler information to the processor

We propose to extend the compiler to mark the delineated regions with a dedicated instruction (`setDRF val`), as shown in Figure 4.3. The operand `val` is a single bit that indicates the beginning of a DRF region (`val` = 1) or a sync region (`val` = 0) for the thread executing the instruction. The compiler identifies synchronization operations that correspond to the standard mechanisms supported in widely used libraries. The memory operations residing in the sync

regions must be executed in-order to ensure correctness and to preserve the TSO guarantees. During the execution of DRF regions, in contrast, memory operations can be performed out of order while inherently preserving the TSO guarantees since no other threads can concurrently perform loads or stores to the same address if at least one thread performs a write (by the DRF definition). Thus, thanks to the SC-for-DRF guarantees, both private and shared variables (i.e. local and global accesses) are treated equally, which simplifies the static analysis and increases its accuracy.

The dynamic percentage of load and store operations that are part of synchronization code (non-DRF) for the applications evaluated in this thesis is shown in Figure 4.5. Generally, the percentage of sync stores is virtually 0. Only in some applications, such as *fluidanimate* and *radiosity* and the synchronization-intensive *pc*, *sps*, and *tatp*, more than 2% of the stores belong to sync regions. On average, only 1.2% of the stores belong to sync regions. Given the immense percentage of sync store operations, performing them in-order is an extremely expensive luxury in TSO processors. Similarly, all LQ searches when stores write becomes unnecessary, thanks to DRF information. Similarly, the percentage of sync loads is small in most applications. Only three synchronization-intensive workloads (*cq*, *rb*, and *tatp*) show a much larger percentage because threads frequently spin on locks trying to acquire them. Despite these three workloads, the percentage of loads that belong to sync regions is 10.2% (and only 3.9% without these three workloads).

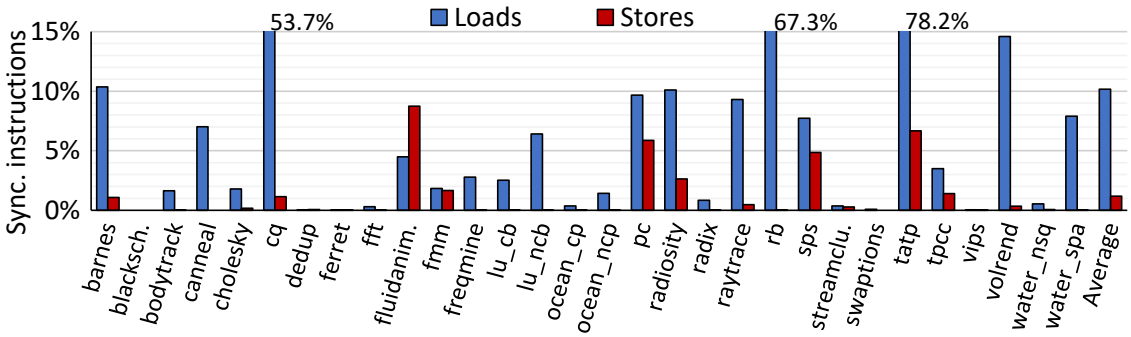


Figure 4.5: Percentage of load and store operations found in synchronization code (sync) at runtime.

4.7 *Proposal 2:* ROOW: Regional Out-of-Order Writes in Total Store Order

ROOW transmits the compiler information to the hardware through a new dedicated instruction as discussed in 4.6. To increase memory level parallelism, our store buffer implementation allow stores to perform out-of-order when they suffer a cache miss. Next sections detail the behavior of ROOW.

Conveying static information to the store buffer: The processor requires minor modifications to execute the `setDRF` instruction. The instruction can be basically considered as a *nop* operation except at commit time. When the `setDRF` instruction commits, a dedicated processor flag called *region flag* (1 bit) changes its mode according to the operand value of the `setDRF` instruction. The update of the flag is performed at commit time since all instructions commit in-order. A value of the flag equal to 0 indicates that the processor is committing sync stores from that point on. A value of 1 indicates that the next stores are DRF. The region flag is set by default to 0. This way, applications in which the DRF regions have not been delineated (e.g. legacy code) still preserve the TSO semantics. When a store commits, it reads the current value of the region flag to detect whether it belongs to a DRF region or a sync region. Stores keep this information in a per-entry bit added to the store buffer, called the *mode* bit. This bit is required because both in-order and out-of-order stores can co-exist in the store buffer, so they must indicate their nature on an individual basis. Figure 4.6 and 4.7 shows an example of how the compiler information is conveyed to the store buffer.

Dual-mode store buffer: ROOW implements a dual-mode store buffer. That is, a store buffer than can simultaneously perform stores in-order or out-of-order depending to the type of store (mode bit): DRF or sync. Store buffers are content-addressable memories (CAM) that implement a circular buffer with a head and a tail pointer. Stores are inserted through the tail and removed from the head. Sync stores behave as in a standard TSO store buffer. They are initiated in-order (from head to tail) and they are inserted in-order in the cache pipeline.[1] If the store hits in the cache, the write will be performed in-order. However, in case of a cache miss, subsequent stores in the cache pipeline would be reordered if they hit in the cache or have a sorter miss latency. To prevent this behaviour, when a store encounters a cache miss, the subsequent stores in the cache pipeline are squashed and they will have to be initiated again, in-order, once the missing store completes. On the other hand, DRF stores may perform out-of-order. DRF stores

[1]Without loss of generality, this thesis assumes a four-stage cache pipeline and a single cache port for stores, so on each cycle a single store is initiated.

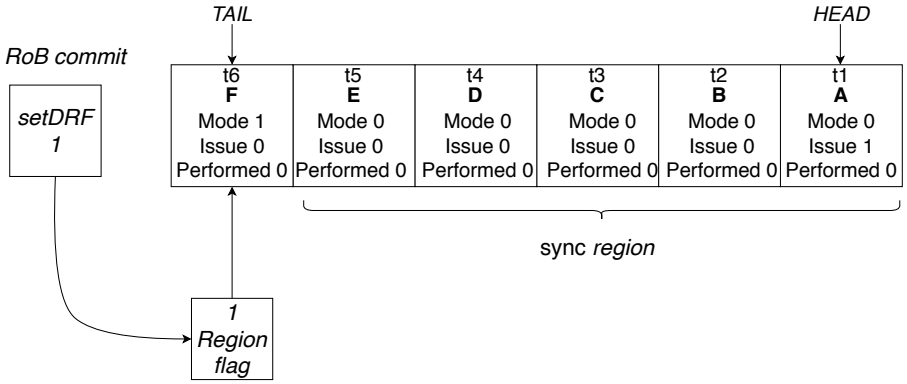


Figure 4.6: Stores A, B, C, D, and E copy the region flag 0 and thus belong to a sync region (Mode bit 0). Once a `setDRF 1` operation commits, the processor sets the region flag and inserts a logical store buffer fence, marking the beginning of a DRF region. Store F that enters after `setDRF 1` copies in its Mode bit the region flag's value, which is now 1.

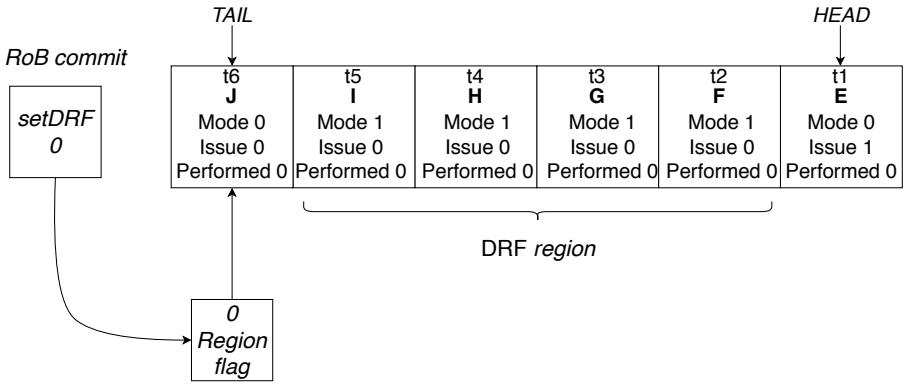


Figure 4.7: Operation `setDRF 0` marks the end of a DRF region: it resets the region flag and triggers the insertion of an store buffer fence. As seen before, store J copies the current value of the region flag in its Mode bit. (Stores F, G, H and I copied the value of the region flag at the moment the stores entered the store buffer, marking them as DRF.)

are also initiated in-order and inserted in the cache pipeline. However, DRF stores are never flushed from the cache pipeline and do not need to be re-initiated. On a cache miss of a DRF store, no action is taken regarding subsequent stores, thus increasing memory level parallelism as more store misses will be in flight. DRF stores may therefore perform out-of-order and are initiated one after the other until the first sync store is encountered.

Note that on a cache miss for a sync store, DRF stores in the cache pipeline do

not need to be flushed. Figure 4.8 and 4.9 shows an example of this scenario. Assume that the state of the store buffer is the one showed in Figure 4.8. Stores A, B, C and D are issued in the cache pipeline to perform the write operation. Stores D, E, and F are DRF stores. Once store A, the store at the head, suffers a cache miss, the processor needs to re-issue all the issued sync stores except itself, as shown in Figure 4.9. These stores, B and C, are flushed from the cache pipeline. However, store D can safely continue, being DRF, and therefore is not re-issued.

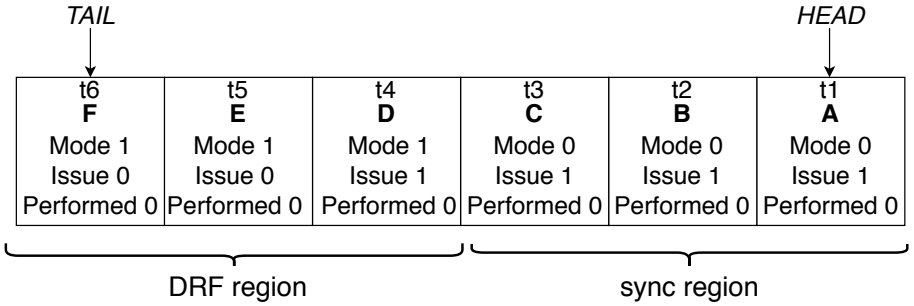


Figure 4.8: Before Miss

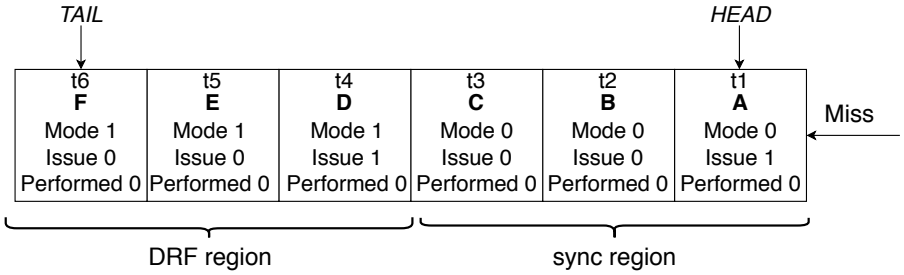


Figure 4.9: After Miss

Store operations are retired from the store buffer in-order once they complete the write, regardless if the store is DRF or sync. That is, the head pointer of the store buffer simply moves to the next store. Removing DRF stores out-of-order, would improve store buffer utilization but in contrast would add complexity to the store buffer and make sequential semantics harder to fulfill, as we explain in the next section.

Results: ROOW performs store operations out-of-order within the code regions indicated as safe by the compiler. ROOW leads to a speed up of 8.13% on average (+1% when performing alias analysis to remove fences, Figure 4.10) and reduces processor stalls by 7.11% (Figure 4.11) compared to a mainstream store buffer.

Our design also allows the use of the store buffer as a cache for free (without adding or changing any hardware) which gives us 18.54% loads-forwarded-from-stores compared to 7.24% in the baseline. We have also conducted a sensitivity analysis which shows that we can reduce the size of the store buffer from 56 entries to 16, while still improving performance by 5.64% (Figure 4.12) compared to the baseline configuration.

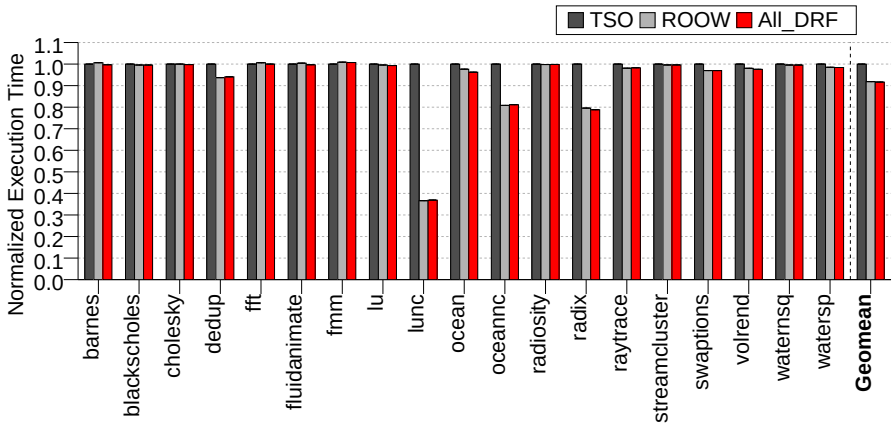


Figure 4.10: Normalized execution time with respect to an store buffer with 56 entries that implements TSO

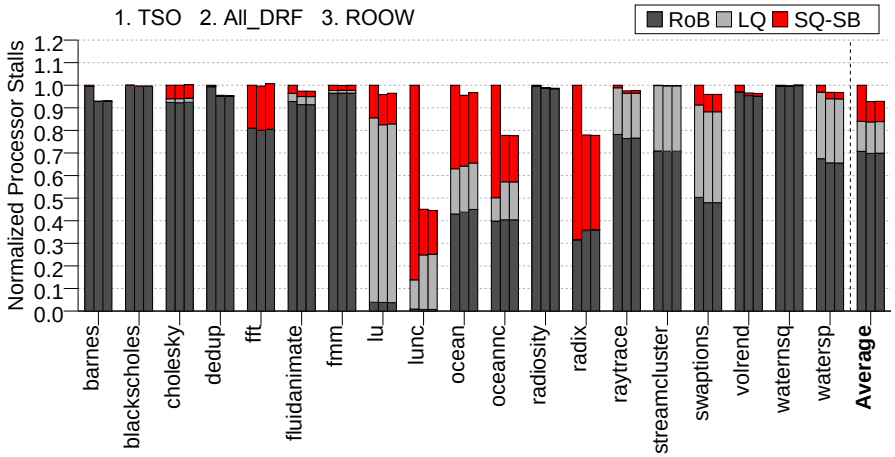


Figure 4.11: Processor stalls

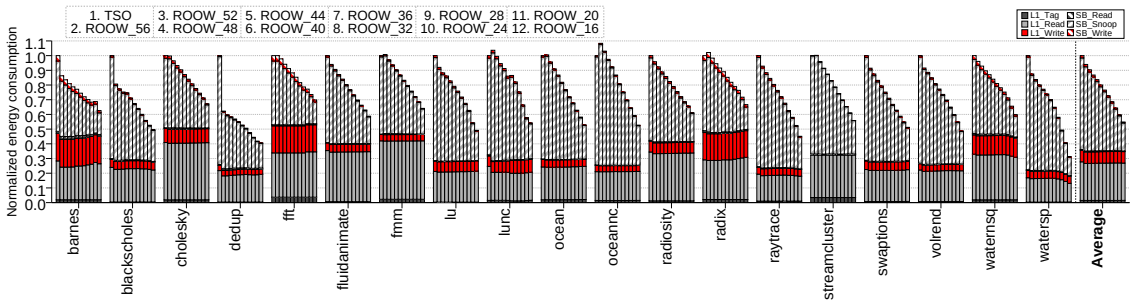


Figure 4.12: Energy consumption of the store buffer and L1 cache

4.8 *Proposal 3:* CELLO: Compiler-Assisted Efficient Load-Load Ordering in DRF Regions

Besides the compiler, CELLO also requires (small) changes within the processor core. Figure 4.13 shows an overview of the CELLO hardware on top of an SMT core, with the additional structures and flags marked in yellow. Each hardware thread has a *region flag* and a *num-sync* counter. The *region flag* is set by the dedicated `setDRF` instruction.Memory instructions read the flag to keep it in the *mode* (*M*) bit augmented to the SQ and LQ entries, and loads increase the *num-sync* counter of its thread if the mode is *sync*. When committing sync loads, the counter is decreased. Store writes, and cache invalidations and evictions check the *store-DRF* and *load-DRF* filters, respectively, and only search the LQ when it is essential to guarantee that no load→load ordering violation is exposed. Overall, in a 2-way SMT core with a 192-entry LQ and a 128-entry SQ, CELLO hardware overhead amounts to 40 bytes (320 mode bits and two 1-bit *region flags*), two 8-bit counters, and simple additional logic per core.

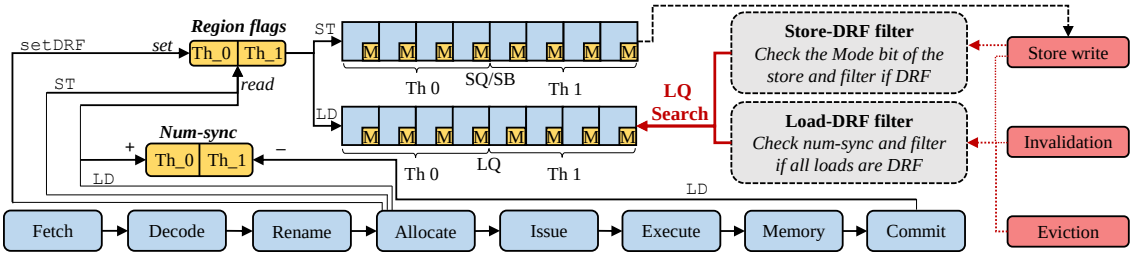


Figure 4.13: Pipeline overview for CELLO on top of an SMT core. The structures / flags in yellow and the DRF-based filters are the additions required to support CELLO.

Reducing LQ searches: Since DRF stores are guaranteed to be executed in the absence of concurrent loads to the same address in other threads, they cannot expose consistency violations (e.g., load→load). Therefore, it is not necessary to perform any search in the same core LQ (to squash M-spec loads from other SMT threads) when a DRF store writes. Store writes can also cause invalidations that reach other cores private caches. An invalidation request caused by the write of a DRF store cannot expose a consistency violation either. Thus, at a first glance, we could skip its corresponding LQ search. However, after the cacheline is invalidated, the core would not detect forthcoming store writes to the same cacheline, potentially from non-DRF stores, which could expose a load→load reordering. Therefore, despite only sync stores can expose consistency violations in other threads, we can only skip safely the LQ searches associated to DRF store writes when they are performed by a thread running in the same SMT core.

Likewise, DRF loads are also guaranteed to be executed in the absence of concurrent stores to the same address in other threads and thus, they cannot expose consistency violations while being executed out of order. Therefore, no LQ search is required on store writes, cache invalidations, and cache evictions when the LQ contains only DRF loads, as they execute non-M-speculatively by definition. On the contrary, consistency violations are possible for racy loads, which may execute M-speculatively and, therefore, based solely on the DRF status of loads, LQ searches are required to prevent exposing any consistency violation. Summing up, CELLO proposes to avoid the LQ search when i) it is caused by the write of a DRF store from a thread running in the same SMT core, or ii) there are no sync loads in the LQ. Since DRF memory operations are far more frequent than non-DRF ones, most LQ searches can be safely avoided, reducing energy consumption in the LQ and contention in its search ports. The implementation for reducing LQ searches is fairly simple. We implement two DRF-based filters: *Store-DRF filter* and *load-DRF filter*.

The *store-DRF filter* is based on the DRF status of stores. When the core performs a write, it checks the *Mode* bit on the corresponding SQ entry. A *Mode* bit set to 1 implies that the store belongs to a DRF region. Therefore, it cannot conflict with concurrent same-address loads in other threads and, consequently, it does not require any LQ search in the same core.

The *load-DRF filter* is based on the DRF status of the loads. We use a counter per SMT thread in a core, called *num-sync*, that holds the number of sync loads for a particular thread in the LQ. The counter is set to 0 by default indicating that there are no sync loads in the LQ, is incremented by 1 once a sync load from the thread enters the LQ, and is decremented by 1 when a sync load from the thread leaves the LQ. When performing a cache invalidation or eviction, the memory

system first checks the *num-sync* counters of the core. If all the counters are 0, the LQ search does not take place, as the pipeline contains no M-speculative loads. When one of the counters is not 0, it indicates the presence of a sync load in the LQ, and thus, the memory system performs the LQ search. Similarly, when the core performs a write, the load-DRF filter checks the *num-sync* counters of the other threads co-running in the core. (It does not need to check the *num-sync* counter of the thread performing the write since this LQ search targets M-spec loads from the other threads running in the SMT core.) If all these counters are 0, the LQ search does not take place.

Early removal of loads from the LQ: Having an unresolved load at the head of the LQ is common, as long latency loads are the main culprits for processor stalls. While loads are not resolved, they are the source of M-speculation for subsequent loads [63]. In a standard TSO implementation, loads remain at the LQ head until they commit and are squashed if there is a match in an LQ search.

In CELLO, we aim to remove the loads from the LQ as soon as they do not need to be searched. Removing loads from the LQ helps reduce LQ occupancy, which either i) eliminates LQ-induced stalls and thus improves performance or ii) allows shrinking the LQ and thus reduces its energy consumption and area. Three conditions need to be satisfied in order to remove a load from the LQ:

1. *The load is at the head of the LQ.* Since the LQ is a circular buffer, occupancy reduction is only effective when removing the load at the head.

2. *The load is DRF.* The compiler guarantees that DRF loads do not conflict with stores from other threads and therefore they are non M-speculative by default.

3. *All stores older than the DRF load at the head have already resolved their address and searched the LQ.* The load becomes, then, non D-speculative.

The first two conditions are checked using the head pointer in the LQ and the *Mode* bit attached to each load in the LQ. For the D-speculative condition, CELLO leverages a bit per entry in the SQ that indicates if the store has executed (commonly found in current implementations). This bit tracks resolved/unresolved store addresses. The default value is 0, indicating that the store address is still unresolved. When the store executes, the bit is set. The load simply performs a bitwise *OR* operation between the *Execute* bits and a bitmask that identifies the stores older than the load in the SQ with the corresponding bit set to 0. A bitmask can be generated with a range decoder, which sets to 0 and 1 the bit corresponding to each store depending on whether it is older or younger than

the load, respectively. An *AND* operation is performed on all the resulting bits. If the result is 1, it means that no unresolved older store addresses exist, and the load can be considered non-D-speculative. Note that since this operation only acts on a single bit per entry in the SQ, it is simpler than an SQ search, which requires a priority decoder and full address comparison.

Results: CELLO filters LQ searches, triggered to detect potential consistency violations, for regions that are indicated as safe by the compiler. Since these regions are predominant, CELLO skips 47% of the LQ searches (Figure 4.15), reducing the total LQ energy expenditure by 33% on average (up to 53%, Figure 4.16) compared to a baseline SMT system. At the same time, CELLO improves the baseline performance by 2.8% on average (up to 18.6%, Figure 4.14), a noticeable benefit given CELLO's minimal hardware overhead: 40 bytes (a 1-bit flag in the LQ and SQ entries), two 8-bit counters, and simple logic to check the flags and counters. Furthermore, by allowing the non-speculative loads to exit the LQ early, CELLO enables shrinking the LQ size from 192 to 80, achieving energy and area savings of 69% and 56%, respectively, without performance penalties compared to the 192-entry LQ baseline system.

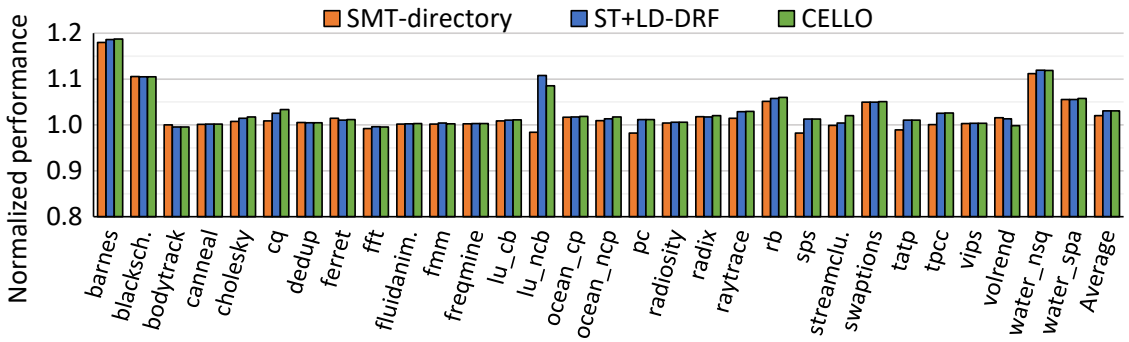


Figure 4.14: Normalized performance for the SMT-directory, ST+LD–DRF filtering, and CELLO setups compared to the baseline system.

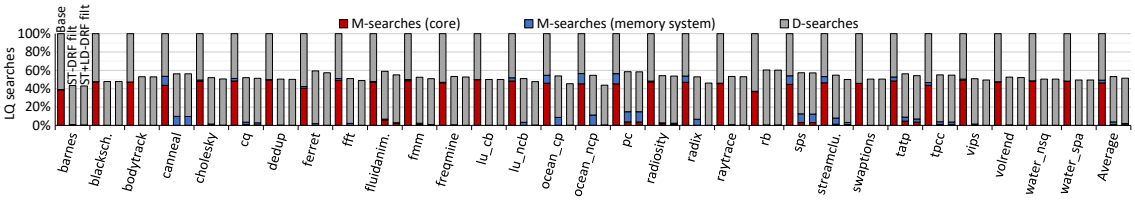


Figure 4.15: Percentage of LQ searches performed with the baseline, ST–DRF filtering, and the ST+LD–DRF filtering configurations compared to the baseline.

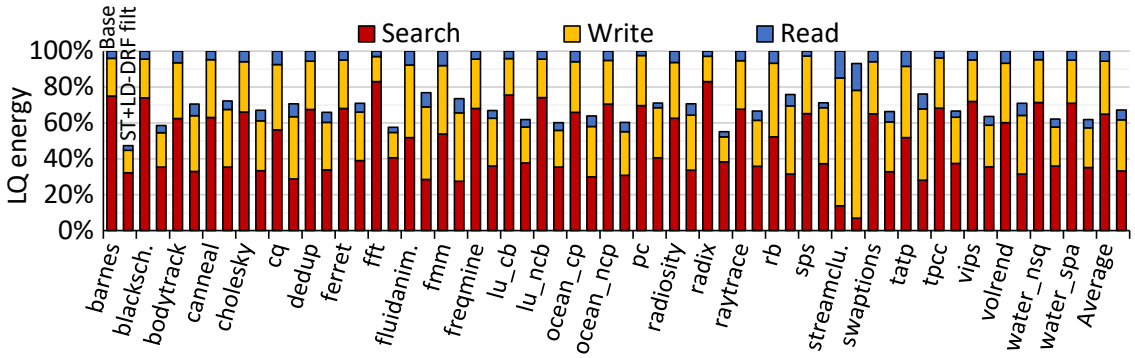


Figure 4.16: LQ energy expenditure of the baseline and ST+LD–DRF filtering configurations normalized to the baseline.

Chapter 5

Exploring Instruction Fusion Opportunities

In this chapter, we introduce Helios, an innovative instruction fusion technique capable of merging non-consecutive instructions. We begin by discussing the motivation behind Helios, followed by the insights that inspired its development. Next, we detail the design and implementation of Helios, and conclude with its evaluation.

5.1 *Research Problem:* Limitation in instruction fusion

Instruction *fusion* is a well-known microarchitectural technique used in many commercially available processors [9–11, 37]. Fusion is used to better exploit available hardware resources by leveraging the fact that instructions do not always require all the resources the internal instruction format allows them to claim (e.g. physical destination registers). Fusion is usually thought of as a technique that will fuse two (or more) µ-ops that are consecutive in the dynamic instruction stream. However, we find that if this constraint were to be relaxed, a non-negligible number of additional µ-ops could be fused. Another limitation of fusion is that it relies on static information to decide whether to fuse two memory µ-ops or not. In the context of RISC-V, memory µ-ops may be fused if: i) They are either all loads or all stores ii) They share the same *physical* base register iii) They access data that resides within a *cacheline_size* region. However, not all fuseable

µ-op pairs meeting condition iii) also validate condition ii). Indeed, we find there exist pairs of both consecutive and non-consecutive memory µ-ops that do not share a *physical* base register and yet access the same cacheline sized memory region, meaning that they could be fused. Yet, it is not easy or even possible to identify those pairs using only static information, as effective addresses are needed to confirm fuseability. Therefore, fusion based on static information is leaving potential on the table.

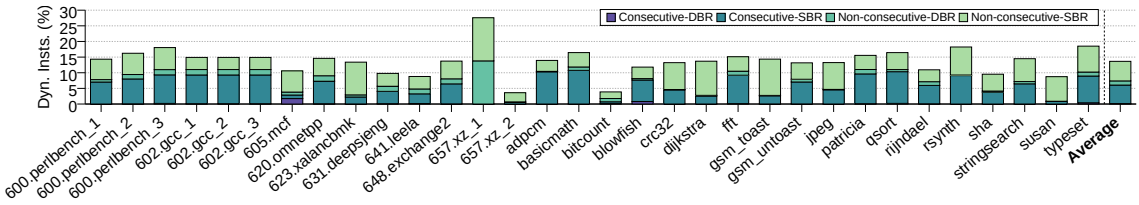


Figure 5.1: Paired memory µ-ops with consecutive, non-consecutive, and different-base register relative to total dynamic µ-ops.

5.2 *Insight:* Dynamic information is requested to identify additional fusion opportunities

Figure 5.1 reports the additional fusion potential brought by non-consecutive fusion (NCSF) and fusion with different base register (DBR) for memory µ-ops. Note that in NCSF, the number of asymetric accesses is quite high, at 12.1% of the NCSF pairs. Conversely, the vast majority of CSF pairs are symmetric accesses. Whereas DBR pairs amount to 1.5% of dynamic µ-ops on average, including CSF and NCSF. This indicates that a significant potential for instruction fusion remains untapped due to insufficient information available at the decode stage to fuse the instruction pairs.

5.3 *Proposal:* Helios

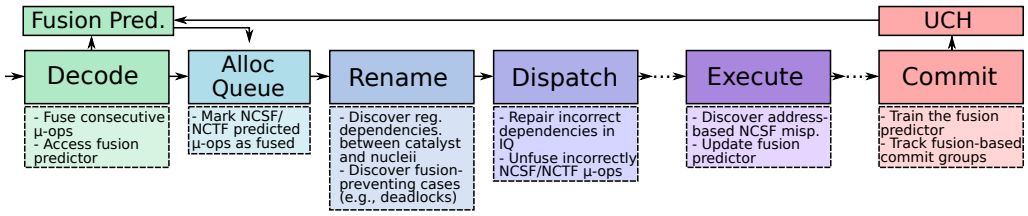


Figure 5.2: Overview of fusion-related responsibilities in Helios.

To support non-consecutive (NCS) and non-contiguous (NCT) fusion of memory µ-ops that potentially use a Different Base Register (DBR), *Helios* relies on multiple changes along the pipeline, which are summarized in Figure 5.2. At a high level, *Helios* relies on three key mechanisms. First, one way to perform NCSF is to have each µ-op inspect all older µ-ops in the Allocation Queue, which sits between Decode and Rename. Such an exhaustive approach to NCSF is costly. Therefore, *Helios* uses a predictive approach to identify not only NCS but also NCT and DBR candidates. Second, NCS fusion may suffer from the presence of dependencies between *nucleii* and their *catalyst*. *Helios* identifies dependencies and, when possible, addresses them so that fusion can still proceed. Third and last, *Helios* handles incorrectly fused instructions as well as other mispredictions (e.g., branch) within a *catalyst*.

A Unified Predictor: *Helios* implements a unified hardware predictor for NCSF, NCTF, and DBR that, given a µ-op PC,[1] provides the *distance*, in µ-ops, to the *head nucleus* to fuse with. The predictor infrastructure consist of 2 structures: the *Unfused Committed History* (UCH) and the *Fusion Predictor* (FP). UCH lives in Commit stage. It is used to find potential fusion pairs, i.e., memory operations that access the same cache line, to train FP. FP is placed in the frontend (Decode), and predicts which µ-ops should be speculatively fused.

Unfused Committed History (UCH): UCH keeps the cache lines accessed by the last committed memory µ-ops eligible for fusion, i.e., the non-already fused memory µ-ops. It is organized as a cache where each entry contains a valid bit, a 32-bit tag (partial cache line address), and a 7-bit commit number (CN), for a total of 5 bytes per entry. Each entry may also feature replacement information depending on the actual design (LRU is done through the CN in this thesis). Distinct histories are implemented for loads and for stores. For loads, the UCH is organized as a fully-associative cache. In our experiments, we find that the *head nucleus* can generally be found a few loads ahead, and therefore we implement a 6-entry UCH for loads with LRU replacement. For stores, a single entry holding the last unfused committed store is kept in the UCH as in Helios, stores cannot be fused across other stores to prevent memory consistency issues. At Commit, loads search the UCH for loads (Ld-UCH) and stores search the UCH for stores (St-UCH). Overall, the UCH structure requires just 280 bits. When a retiring µ-op matches a tag in the UCH, a potential fuseable pair has been found. The distance between the two µ-ops is computed by subtracting the CN of the committing µ-op to the matching entry's CN, and the matching entry is invalidated, as µ-ops

[1] In this paper, RISC-V memory instructions always translate to a single µ-op, hence the PC to µ-op equivalence for memory µ-ops.

can only fuse with a single other µ-op. The maximum distance that we allow for fusion is 64 µ-ops, so the CN field requires 7 bits (the last bit controls the CN overflow). The FP is then updated using the computed distance as explained in the next Section.

On a miss in the UCH, the µ-op is inserted into the UCH. Invalid entries resulting from a previous match are preferred victims for replacement, followed by the LRU entry. It should be noted that this process is out of the execution critical path and can be done post-commit. That is, a post-commit decoupling queue in which at most n committing loads (resp. store) µ-ops are inserted each cycle can be implemented. If the queue is full, µ-ops are simply dropped and will get a chance to train at a later time. The queue is drained at a rate of m µ-op each cycle, with m the number of UCH ports. In our experiments, on average 0.23 loads update and 0.28 loads search the UCH per cycle at commit (0.13 and 0.16 per cycle for stores). Experiments further suggest that implementing an 8-entry queue in front of the load UCH and allowing a single UCH search and update per cycle has no impact on the performance of Helios.

Fusion Predictor (FP): FP contains information about potential *tail nucleii*. FP is organized similarly to a cache, each entry containing an 8-bit tag, a 6-bit µ-op distance to the head *nucleus* to fuse with, a 2-bit saturating counter, and a pseudo-LRU bit. Each entry therefore requires 17 bits. The processor attempts to allocate an FP entry for a committing µ-op when a match is found with an older UCH entry. If a match is found, FP is searched and if the tag is already present in FP and the distance matches, the confidence counter of the entry is increased. If the distance does not match, the new distance is inserted and the confidence is set to 1. If the tag is not found, an entry is selected for eviction following a pseudo-LRU replacement policy.

In this thesis, we chose a tournament predictor [41], which selects from a "local" PC-based predictor and a "global" *gshare*-like predictor, to implement FP. It includes a 512-set, 4-way structure indexed by the PC and another 512-set, 4-way structure indexed by a XOR of the PC and the global branch direction history. Each structure therefore features 2048 entries, amounting to 34Kbits each. A 2048-set direct-mapped and untagged selection table containing 2-bit saturating counters (4Kbits) is used to select which prediction is used. The total predictor bitcount is therefore 72Kbits (9KB). Alternatively, other predictors, such as TAGE-based [75] or local history based [83], can be employed. In the context of RISC-V, which features aligned instructions, the predictor structures may be implemented as multiple single-ported banks interleaved on PC. In practice, a number of banks greater than the decode width is preferable to handle cases where µ-ops belonging to different basic blocks are at Decode. A high number of

banks also permits to perform both predictions and updates in the same cycle if they go to different banks, as described by Seznec et al. [74]. Once a *distance* is retrieved from the FP at Decode, fusion is attempted in the Allocation Queue, and is successful only if the following conditions are met:

1. The saturating counter has the maximum value (3).
2. The two µ-ops form a valid fusion idiom, that is:
   - Both µ-ops are loads or both are stores.
   - The *head nucleus* is not already a fused µ-op.
3. The *head nucleus* still resides in the Allocation Queue or is in the same Decode Group as the *tail nucleus*.

The 2-bit saturating counter is updated when the fused µ-op executes by computing the target addresses, and a misprediction is uncovered. On a correct prediction, the entry is not updated since the confidence counter has already saturated from the UCH-based training process. Updates are achieved through a dedicated structure that contains relevant prediction information (e.g., index of tables used for prediction, predicted distance, confidence) for µ-ops that flow down the pipeline, similarly to how branch or value prediction update may be handled [68]. While its exact size depends on implementation details (e.g. how many entries are sufficient to prevent stalling), each entry requires 29-bit of storage given the predictor we consider (assuming selector and PC-based set indexes can be regenerated from the PC at update time). In our experiments, we consider an unlimited queue. The confidence counter is reset to 0 on a fusion misprediction.

FP can be integrated in a microarchitecture featuring a µ-op cache by having FP and the µ-uop cache searched in parallel. Further integration of FP in the µ-op cache appears wasteful because not all µ-ops are eligible for non-consecutive fusion. However, directly caching consecutively fused µ-ops in µ-op cache entries is a possibility, as long as consecutively fused µ-ops contain enough information to be unfused at the output of the cache if a branch jumps to the *tail-nucleus*. Caching NCSF'd µ-ops appears less synergistic because NCS fusion is inherently dynamic. For instance, depending on control flow, a load may fuse with younger load A or younger load B (e.g. if A is on the taken path and B is on the fallthrough of the same conditional branch). Statically caching one of the two possible NCSF'd µ-ops in the µ-op cache would be unable to capture this behavior. It may however be adapted to constrained NCS fusion schemes that do not allow any control-flow change within the *catalyst*.

Results: Helios relies on a predictive scheme to fuse distant µ-ops and tackles numerous challenges to guarantee correct execution. Figure 5.3 shows the normalized IPC of Helios compared to a baseline configuration with no fusion and Figure 5.4 shows the number of CSF and NCSF pairs in Helios and OracleFusion. Helios achieves a significant performance uplift over microarchitectures supporting various flavours of fusion, notably 14.2% over no fusion and 8.2% over consecutive and contiguous only memory fusion.

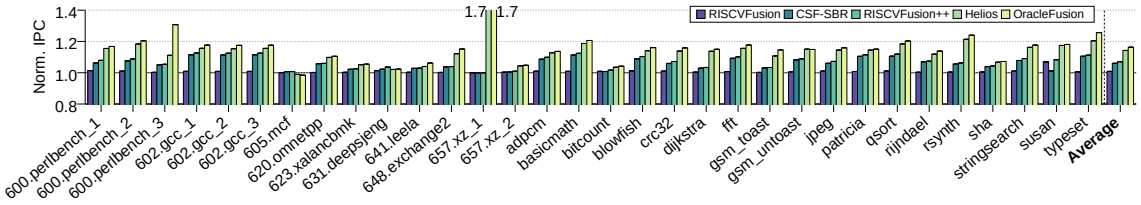


Figure 5.3: Normalized IPC with respect to baseline configuration with no instruction fusion.

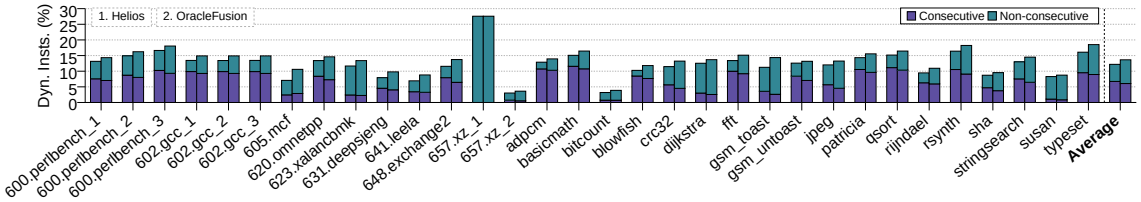


Figure 5.4: Number of CSF and NCSF pairs in Helios and OracleFusion.

Chapter 6

Alternate Path µ-op Cache Prefetching

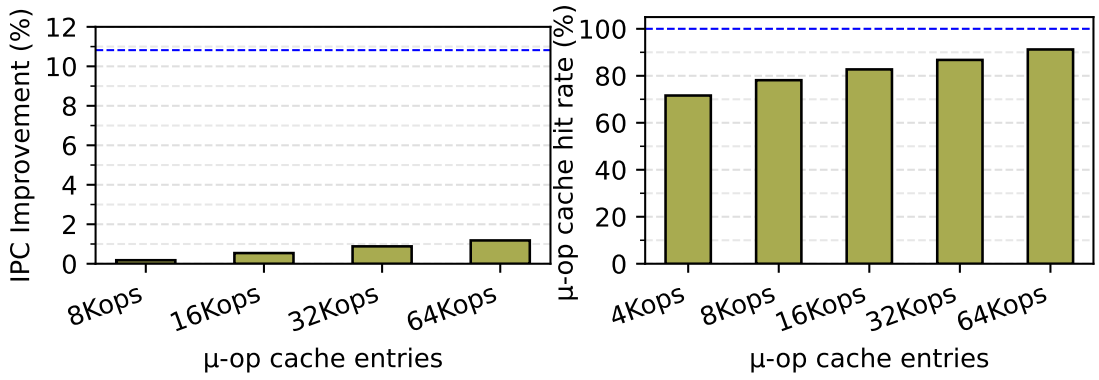
In this chapter, we discuss the behavior of μ-op caches in modern processors when running server workloads. We show that the current μ-op cache design is not able to capture the critical instructions in server workloads and propose prefetching instructions from the alternate path into the μ-op cache. and propose a new μ-op cache design that focuses on capturing the critical instructions in server workloads. We show that our proposed design can provide significant performance benefits for server workloads.

6.1 *Research Problem:* Server workloads overwhelm current μ-op caches

The µ-op cache has been primarily designed for power savings [76], by holding the µ-ops of frequently executed instructions. However, in modern x86 processors, its role goes beyond that. Indeed, since decoding multiple x86 instructions in parallel is a hard problem, decode width remains limited to 4-5 architectural instructions even in aggressive designs. However, the µ-op cache width can exceed this limit at minimal cost, by caching more µ-ops per entry. For instance, AMD Zen4 can provide up to 9 *macro ops*[1] per cycle from the µ-op cache, while it is limited to decoding 4 architectural instructions per cycle, which generally yield

[1]Amd translates x86 instructions to one or more *macro ops*. *Macro ops* are therefore decoded instructions.

fewer than 9 *macro ops* [21]. Therefore, from a performance standpoint, the larger width combined with the shortened frontend length stemming from bypassing decoders makes the µ-op cache an efficient pipeline (re)fill accelerator, as long as the requested µ-ops are found in the µ-op cache. We emphasize that caching µ-ops is not limited to microarchitectures implementing complex instruction sets. For instance, the ARM Neoverse V2 microarchitecture features a 1.5K-entry decoded cache [36].



(a) Speedup (b) µ-op cache hit rate

Figure 6.1: Analysis increasing the µ-op cache size. The blue line represents an ideal µ-op cache.

The average (amean) µ-op cache hit rate for a 4Kops reported in Fig. 6.1b is 71.6%. Overall, we found that about half of the applications considered in this thesis exhibit a hit rate of 70% or less, suggesting that the code footprint of datacenter workloads overwhelms the µ-op cache. We analyze whether larger µ-op caches would sufficiently increase the hit rate and therefore performance. Figure 6.1a and 6.1b reports IPC and µ-op cache hit rate when increasing the µ-op cache size from 4Kops to 64Kops. Doubling the size from 4Kops to 8Kops increases the hit rate from 71.6% to 78.2% and yields an IPC improvement of only 0.18%, with a maximum improvement of 1.3% and a maximum slowdown of -3.6%. Even a 16x larger µ-op cache provides IPC improvements of only 1.2% with a hit rate of 91.2%. This is still far from the average performance gain of an ideal µ-op cache, which stands at 10.8% (blue line).

6.2 *Insight:* Focusing on few but critical instructions

One would assume that instruction prefetching through decoupling branch prediction and fetch (FDP) would be sufficient to hide instruction misses, should branch prediction be able to run ahead far enough. However, FDP can only prefetch *predicted*-path instructions: On a branch misprediction, long latency instruction fetches can harm performance since the correct path was not prefetched. We found that overall processor performance is sensitive to the refill time after a conditional branch miss. Figure 6.2 shows the IPC improvement when all instructions after a conditional branch misprediction are marked as µ-op cache hits, until 8 conditional branches have been fetched. *IdealBRCond-16* is similar to *IdealBRCond-8* but marks all instructions as µ-op cache hits until 16 conditional branches have been fetched. *IdealBRCond-8* provides an improvement of 2.3%. When up to 16 branches are considered, the IPC increase is 2.9%. This improvement comes from expediting instruction dispatch after a branch misprediction, that is, on a pipeline refill. This is akin to perfectly prefetching µ-ops after branch mispredictions. This improvement comes from expediting instruction dispatch after a branch misprediction, that is, on a pipeline refill.

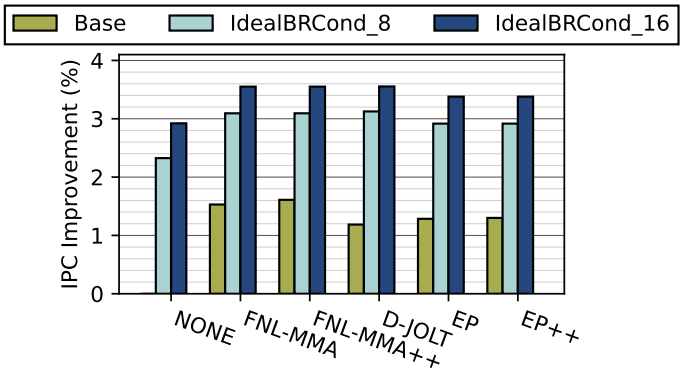


Figure 6.2: Speedup

6.3 *Proposal:* Alternate Path μ-op Cache Prefetching

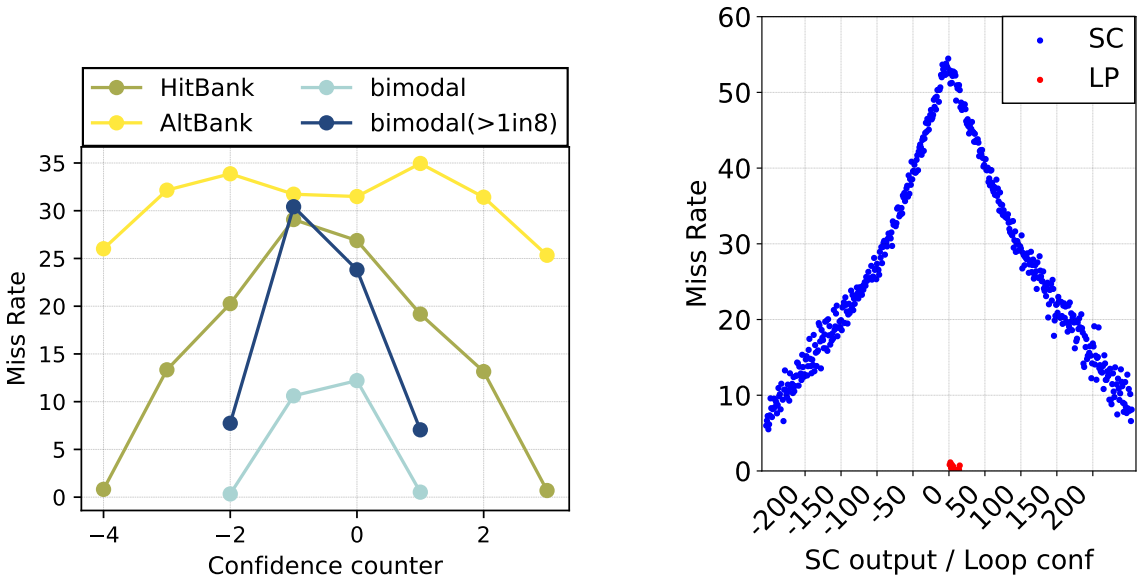
We propose to trigger alternate path µ-op cache prefetching (UCP) on low-confidence conditional branch predictions. By targeting conditional branches, there is a unique alternate path to follow, which simplifies our detection and

prefetching mechanisms. The first step to enable UCP consists in detecting low-confidence conditional branch predictions, which we treat as hard-to-predict (H2P) branches. Second, we start generating the alternate path addresses, prefetch their corresponding instructions, decode, and store them in the µ-op cache, without hindering the progress of the predicted path. Finally, the last step consists in determining when the alternate path is unlikely to be useful, in order to stop prefetching and prevent µ-op cache pollution.

Branch Prediction Confidence: Our predictor is based on the confidence estimation heuristic that can be built within the TAGE branch predictor [71], which determines the confidence of a direction prediction (low, medium, and high) based on the table that provided the prediction and the value of the saturating counter. TAGE employs 37 [72] tagged tables and a Bimodal base predictor. A prediction is provided either by Bimodal, or by one of the tagged tables, but TAGE distinguishes between the HitBank and the AltBank. Both are tagged tables that match the context, but HitBank is the one using the longest global branch history, while AltBank is the one using the second longer global branch history. In the initial heuristic, high confidence predictions are the ones that find the counter saturated regardless of which table provides it, unless the prediction comes from the bimodal table and there was at least one misprediction in the last eight predictions provided by the bimodal table.

Fig. 6.3 displays the average miss rate of a state-of-the-art 64KB TAGE-SC-L [72] predictor, depending on the component used for the prediction and the counter values. Fig. 6.3a shows that indeed, when the prediction uses the saturated counters of HitBank or of the bimodal predictor, the probability of a misprediction is close to zero. However, when there was a miss in the last eight predictions provided by the bimodal predictor (bimodal $>$1in8), the misprediction rate is higher than 6% on average, although the counters are actually saturated (-2 and 1). Fig. 6.3b shows a similar trend for the Statistical Corrector (SC), that is, the higher the absolute value of the output is, the higher the prediction confidence. Yet, the miss rate remains quite high (around 10%) even if the output value is saturated. Fig. 6.3 is completed by Fig. 6.4, which illustrates the misprediction contribution of different components within TAGE-SC-L. One can observe that, on average for the traces used in this thesis, 66.7% of the mispredictions are provided by the HitBank. The AltBank account for 8.1% of the total mispredictions. The bimodal component incurs 6.2% when no misses are found in the last 8 predictions and 7.5% otherwise. The Loop Pedictor (LP) negligibly contributes to mispredictions (0.1%). SC accounts for 11.1% of the mispredictions.

Driven by the miss rates shown in Fig. 6.3, we improve the TAGE confidence estimation heuristic [71] in several ways. First, we underline that the original



(a) Bimodal and tagged tables

(b) SC and LP

Figure 6.3: Average misprediction rate for different components in a 64KB TAGE-SC-L, per output value

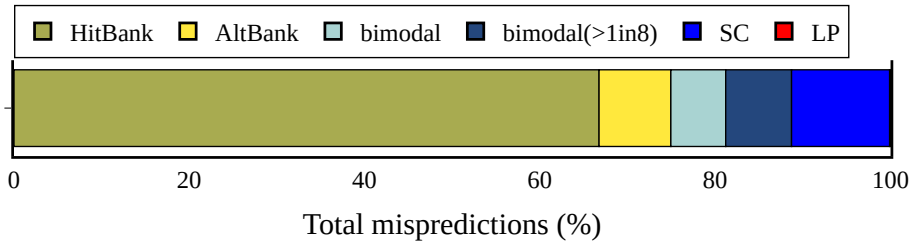


Figure 6.4: Contribution of 64KB TAGE-SC-L components to mispredictions

heuristic does not differentiate between predictions stemming from the HitBank or from the AltBank. In contrast, we analyze the confidence of the predictions per bank and show that predictions stemming from the AltBank *always* exhibit a very high miss rate, regardless of the value of their counter, as shown in Fig. 6.3a. Hence, in this thesis, we consider that any prediction provided by AltBank has low confidence, which is noticeable, given its 8.1% fraction of the total mispredictions.

Second, since the original TAGE confidence estimation was developed for a simpler TAGE predictor, we extend in this thesis the confidence estimation to LP and SC. Fig. 6.3b shows a particularly low miss rate in predictions originating from LP in TAGE-SC-L (<3%, independently from the confidence value) and

therefore consider LP predictions as high-confidence. On the other hand, the confidence of SC predictions in TAGE-SC-L vary depending on the absolute SC output value (Fig. 6.3b) from 10% to 50%, so they cannot be considered as high confidence. SC represents 11.1% of the total mispredictions. These extensions improve both the accuracy and coverage of the original TAGE confidence estimator and add support for LP and SC, without any extra storage.

Initiating the Alternate Path: Using our described confidence estimator built on top of TAGE-SC-L, we classify a given branch instance as H2P if its prediction is from (1) bimodal if there was a misprediction in the past 8 branches predicted by bimodal, (2) bimodal or HitBank for which the prediction counter is not saturated, (3) AltBank, and (4) SC. Generally, this corresponds to predictor entries for which the misprediction rate is above 5%, according to Fig. 6.3. At branch prediction time, if a conditional branch is identified as H2P, alternate path generation is initiated.

Generating the Alternate Path: To generate alternate path addresses past a single basic block, an entire BPU is required, including a BTB, an indirect target predictor, a RAS, and a branch predictor. Replicating those structures to predict the alternate path would add considerable area overhead, since they are the largest frontend structures, e.g. 560KB for the BTBs and branch predictor [33].

Hence, we opt for doubling the number of banks (from 16 to 32) of our baseline banked BTB design [57], which are shared between the predicted path and the alternate path. This lets us retrieve branch targets on both the predicted and alternate paths without implementing a separate BTB, at the cost of bank conflicts. Practically, at the beginning of the BTB access cycle, we determine which banks need to be accessed by the predicted and alternate path. On a conflict, rather than selecting a winner that "takes all", accesses are resolved in the following way: UCP keeps a 3-bit saturated counter to track the number of cycles that the current alternate path PC has been delayed due to a conflict. When the counter saturates, the alternate path is allowed to win the conflicted banks, causing the demand path to retry in the next cycle. The counter resets when the current PC of the alternate path changes. As banking incurs area and latency costs, other BTB organizations such as the region BTB (an entry covers n taken-at-least-once branches of an aligned code region) or block-based BTB (an entry covers a dynamic block of i instructions with at most n taken-at-least-once branches) could be considered [58]. With those, both paths would access a single entry, such that concurrent predictions could be achieved with only a handful of banks. However, since UCP is conceptually agnostic of the BTB organization, we only considered the instruction BTB.

For conditional branches, we use a small TAGE-SC-L branch predictor [73] (*Alt-BP*). The reason for building a dedicated conditional predictor on the alternate path is that naively banking the tagged tables by restricting each PC to a single bank within a tagged table significantly harms performance, and efficiently banking TAGE to enable multiple predictions per cycle has not been covered in the literature and is beyond the scope of this thesis. Alt-BP is updated along with the main branch predictor, meaning that its GHR will diverge from the predicted path only when alternate path is initiated. In practice, Alt-BP implements two GHRs. When alternate path starts, the predicted path GHR pre-H2P branch is copied into the alternate path GHR, and the two are speculatively updated with the predicted direction and its opposite, respectively. From that point on, the predicted path GHR of Alt-BP is speculatively updated with predictions from the main predictor, while the alternate path GHR is speculatively updated using predictions from Alt-BP, which are made using the alternate path GHR. When the alternate path exits, no specific care has to be taken, as the alternate path GHR will be resynchronized once a new alternate path starts again. Operating Alt-BP in this fashion implies that its prediction tables are not updated if the alternate path is incorrect. Indeed, during alternate path operation, predictions are generated for the alternate path only. Therefore, if the predicted path is correct, there is no corresponding state captured in the FIFO structure used to update Alt-BP (i.e. entry number, counter value). However, updates on the alternate path are performed if the alternate path becomes correct, as a pipeline flush will take place and Alt-BP will eventually be updated with the corrected path information.

Our UCP proposal leverages a small ITTAGE [70] (*Alt-Ind*) indirect predictor to prevent early exiting the alternate path because an indirect branch target is unknown. We use a dedicated predictor for the same reason as the branch predictor: banking efficiency. It operates similarly to Alt-BP (GHR, updates). However, and as we will show in results section, the gains brought by a dedicated indirect target predictor are generally limited on average and UCP could be implemented without a dedicated indirect target predictor to limit overhead. Finally, to handle returns on the alternate path, we use a dedicated RAS (*Alt-RAS*). The main RAS is copied into the Alt-RAS when alternate path UCP starts, and it is updated speculatively when walking the alternate path. Both main path and alternate path address generation are performed in parallel. The generated addresses from both paths are added to their respective FTQs (named *Alt-FTQ* for the alternate path).

Prefetching the Generated Alternate Path: Addresses at the head of the Alt-FTQ are first used to perform a µ-op tag check before initiating a prefetch

request, to prevent prefetching instructions already present in the µ-op cache. This tag check is conducted simultaneously with other ongoing tag checks on the predicted path, and is facilitated by set interleaving the µ-op cache into two 2-ported banks. In the event of a conflict during the tag check process, priority is given to the address on the predicted path, while the alternate path address attempts again in the next cycle, similar to the BTB accesses. Once the µ-op cache tag check completes, the address is removed from Alt-FTQ. Upon a µ-op cache miss, a prefetch request for the cache line corresponding to the missing instruction is recorded in the µ-op cache Miss Status Holding Register (MSHR) and inserted in the L1I prefetch queue (PQ). From the PQ, it proceeds as a standard L1I prefetch: if the entry is not already present in L1I, the cache line will be fetched from L2, LLC, or memory. The system is able to process only one prefetch request per cycle, but since the L1I is set-interleaved, both demand and prefetch requests can proceed in the same cycle if they map to different banks. When a cache line returns whose retrieval was initiated by alternate path UCP, the requested instructions are directed to a dedicated decode queue where they are decoded and inserted in the µ-op cache.

Stopping the Alternate Path: The alternate path address generation stops automatically in the following two cases: (1) a new H2P branch is detected, and therefore a new alternate path is initiated; (2) the path being explored is considered too unlikely to become the correct path. The heuristic to stop the alternate path builds on the heuristic for estimating confidence of branches and additionally considers target predictions. The stopping heuristic relies on a 7-bit saturated counter that is initialized to zero when alternate path prefetching is triggered. The counter is incremented with a different weight every time a branch is encountered on the alternate path. The weights are adjusted based on the average hit rate of each branch prediction category (Fig. 6.3) (approximately 1 unit per extra 5% miss rate – see Table 6.1 for details). The higher the value of the counter, the more unlikely for the next basic block in the alternate to become the correct path.

The alternate path stops either when the counter reaches an established threshold (e.g., 500) or when a clear low confidence event (e.g. a BTB miss) occurs. We also stop on the detection of an indirect branch if we do not employ an Alt-Ind predictor. Finally, to avoid indefinitely generating addresses for loops never predicted to end, and to restrict it to the critical instructions, the threshold is incremented by 1 for high confidence branches. As the threshold is updated only when a predicted branch is encountered, UCP can continue generating prefetching addresses if no branches are found. To avoid this, UCP keeps a 6-bit counter that resets on each predicted branch and is incremented by 1 for each

Table 6.1: Weights added to the saturation counter on specific events on the alternate path.

	Prediction Source	Predictor Output	Weight
Condition	BiModal	-2 & 1	1
		-1 & 0	2
	BiModal (>1in8)	-2 & 1	2
		-1 & 0	6
	HitBank	-4 & 3	1
		-3 & 2	3
		-2 & 1	4
		-1 & 0	6
	AltBank	-4 & 3	5
		-3, -2, -1, 0, 1, 2	7
	Loop Predictor	Any	1
	SC	128 to 255	3
		64 to 127	6
		32 to 63	8
		0 to 31	10
Target	BTB Miss	–	∞
	Indirect branch	–	1 (or ∞)
	Return branch	–	1

instruction on the alternate path. The alternate path will cease once the counter has reached its maximum value.

Overview and Hardware Overhead: Figure 6.5 depicts the modifications required for UCP. Gray boxes indicate the added components and dotted lines represent newly introduced data paths. Our design incorporates an 8KB TAGE-SC-L branch predictor (Alt-BP), a 4KB ITTAGE indirect target predictor [70] (Alt-Ind), and a 16-entry Alt-RAS (0.06KB), that are combined to the BTB for generating alternate path addresses. These addresses populate an alternate 24-entry FTQ that holds µ-op cache entry addresses (0.14KB) ❶. A 32-entry µ-op cache MSHR table (0.19KB) is also employed to monitor ongoing prefetch requests ❷. We double the tag check bandwidth to the µ-op cache by banking the µ-op cache and managing conflicts ❸, as in the BTB. Prefetches that miss the µ-op cache are inserted in the L1I Prefetch Queue (PQ, 0.25KB) ❹. After prefetch completion ❺, instructions enter a 32-entry alternate decode queue (0.12KB) and are subsequently decoded using 6 dedicated *Alt-Decoders* ❻ before being added to the µ-op cache ❼. The overall memory overhead required by UCP is 12.95KB

(8.95KB when not leveraging an Alt-Ind predictor).

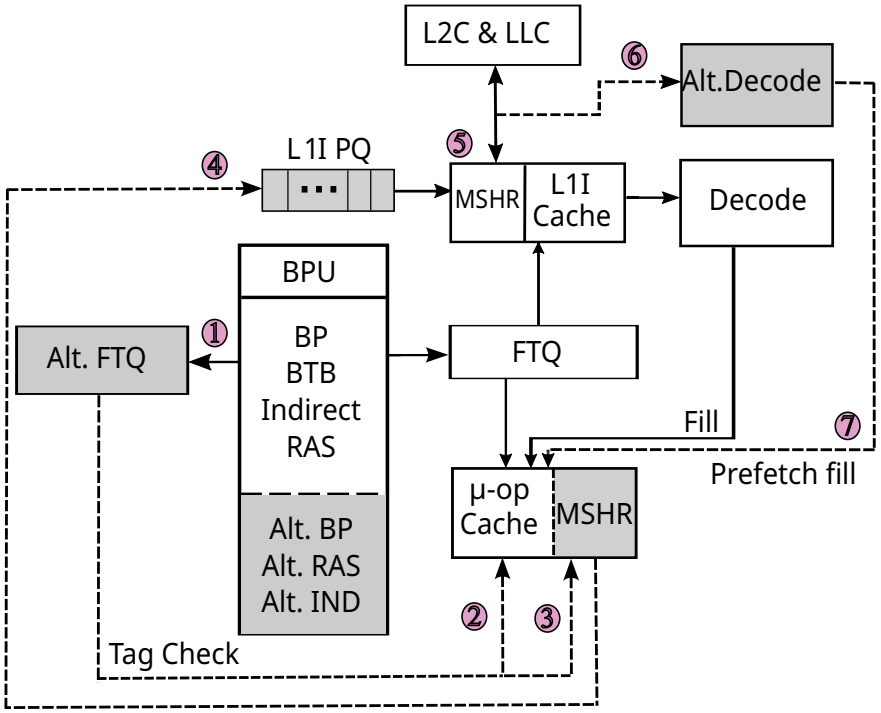


Figure 6.5: New structures and data-paths required by UCP

Results: UCP shows that by focusing on few but critical instructions, significant performance benefits can be obtained. Specifically, we prefetch in the µ-op cache only a few cache lines worth of the alternate path of hard-to-predict conditional branches, achieving an average of 2% and up to 12% speedup (resp. 1.9% and up to 10.6%) with a moderate storage overhead of 12.95KB (resp. 8.95KB), which includes alternate predictors and queues to track and follow the alternate path. Figure 6.6 shows the performance improvement and storage requirement of UCP compared to other techniques. UCP has few variations which include 1. *UCP-NoIndirect:* UCP without a dedicated indirect predictor for alternate path 2. *UCP-L1I:* Prefetching only till L1I, 3. *UCP-SharedDecoders:* UCP with shared decoders with the demand path, 4. *UCP-AltDecoders:* UCP with dedicated decoder for prefetched instructions, 5. *UCP-NoBTBConflict:* UCP with no BTB conflict resolution. UCP outperforms larger µ-op caches or prefetching in the µ-op cache using a standalone L1I prefetcher.

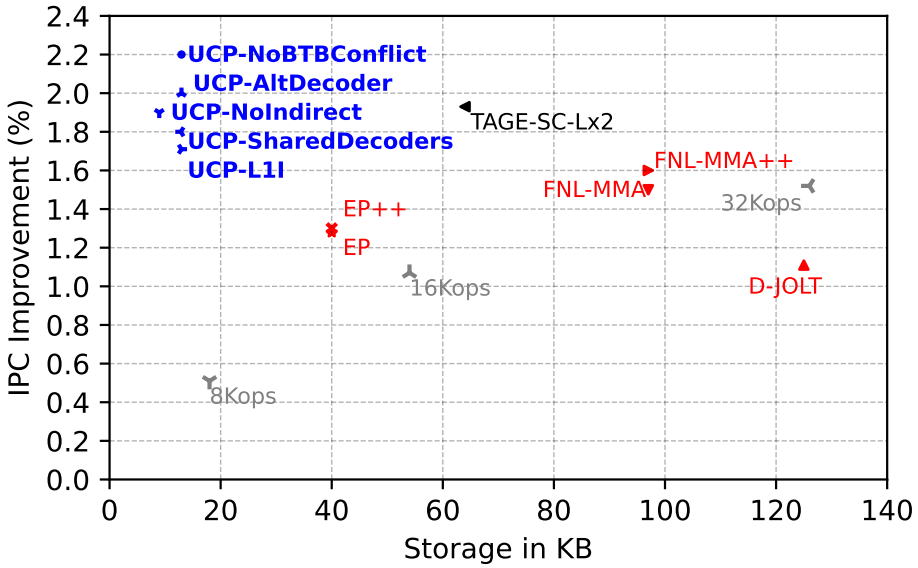


Figure 6.6: Performance improvement and storage requirements of UCP are shown in blue, L1I prefetchers (EP, EP++, DJOLT, FNL-MMA, and FNL-MMA++) in red, and µ-op caches (8Kops, 16Kops, 32Kops) in gray. TAGE-SC-L with double size is shown in black

.

Chapter 7

Conclusion and Future Works

This thesis tackles the challenge of enhancing modern processor performance by investigating new methods and refining existing ones. For the processor front-end, we introduced a technique that focuses on a few critical instructions. Our findings demonstrate that instruction criticality-driven solutions can effectively manage demanding server workloads. To address stalls caused by saturated structures like the ROB, LQ, and SQ, we proposed a prediction-based fusion mechanism. This mechanism advances current fusion techniques by enabling the fusion of non-consecutive instructions. Additionally, we explored a software-hardware co-design approach to prevent stalls in the SQ by selectively executing certain write operations out-of-order (OoO). This approach also allows for skipping costly LQ searches, thereby reducing the processor's overall power consumption.

Key Contributions:

Store Buffer (SB) Optimization: Regional Out-of-Order Writes (ROOW) One of the major challenges in out-of-order processors is maintaining memory consistency, particularly in the Total Store Order (TSO) model, which ensures that store operations are performed in program order. The SB becomes a bottleneck when the head entry waits for memory access (such as during cache misses), stalling subsequent store operations. To mitigate this, we introduced ROOW (Regional Out-of-Order Writes in Total Store Order). This mechanism enables safe out-of-order writes within designated Data-Race-Free (DRF) regions of code. By using compiler-delineated DRF regions where memory accesses from different threads or cores do not overlap, ROOW allows stores to bypass the head entry in the SB, avoiding unnecessary stalls while preserving TSO semantics.

ROOW demonstrated a significant 7.11% reduction in processor stalls and an 8.13% improvement in execution time across a variety of benchmarks. This optimization not only enhances the efficiency of the SB but also has broader implications for processors dealing with high-latency memory operations, making it a crucial advancement for both general-purpose and specialized high-performance processors.

Load Queue (LQ) Optimization: The LQ is a critical structure in OoO processors, ensuring memory consistency and tracking all in-flight loads. Frequent searches in the LQ for potential memory violations especially in Simultaneous Multithreading (SMT) processors where multiple threads share the same core lead to performance degradation and energy inefficiency. To address this, the thesis proposed CELLO (Compiler-Assisted Efficient Load-Load Ordering). CELLO leverages compiler-generated DRF information to selectively filter unnecessary LQ searches, particularly in regions where memory order violations are unlikely.

By reducing the number of LQ searches, CELLO improves both energy efficiency and performance. The implementation led to a 47% reduction in LQ searches, 33% energy savings, and a 2.8% overall performance improvement. CELLO's ability to intelligently manage LQ accesses while ensuring correctness makes it a powerful technique, particularly in SMT processors with high memory contention.

Instruction Fusion: Modern processors break down complex instructions into simpler micro-operations (μ-ops), which are easier for hardware to handle. However, the ability to fuse these μ-ops back together known as instruction fusion can greatly enhance processor throughput by reducing the number of pipeline entries consumed. Helios extended the traditional concept of instruction fusion by fusing non-consecutive μ-ops and instructions with different base registers.

Helios used a predictive model to identify fusion opportunities beyond consecutive instructions, achieving a 5.5% increase in dynamic μ-op fusion compared to baseline implementations. This resulted in a 8.2% performance improvement by reducing contention in the Reorder Buffer (ROB), Load Queue (LQ), and Store Queue (SQ). The innovation provided a more efficient use of pipeline resources, proving especially beneficial for high-throughput applications where resource congestion is a critical issue.

μ-op Cache Prefetching: Large server workloads and data center applications present unique challenges due to their massive instruction footprints, which often exceed the capacity of the L1 Instruction Cache (L1I) and the μ-op cache. Frequent cache misses lead to frontend stalls, severely impacting performance. To address this, we introduced μ-op Cache Prefetching (UCP), which targets the alternate path for branch mispredictions, prefetching μ-ops into the μ-op cache.

By focusing on hard-to-predict (H2P) branches, UCP reduces the penalty of branch mispredictions by ensuring that μ-ops from the alternate path are already cached and ready for execution. This technique achieved a 2% performance improvement with minimal hardware overhead. UCP's ability to reduce frontend stalls in data-heavy server workloads showcases its effectiveness in environments where instruction cache misses are frequent.

Broader Implications:

The work presented in this thesis addresses some of the most critical bottlenecks in modern out-of-order processors, especially in the context of high-performance and server-grade workloads. By enhancing structures such as the SB, LQ, μ-op cache, and pipeline resources, these innovations pave the way for more efficient and scalable processor designs. ROOW shows that even within strict memory models like TSO, there are safe opportunities for out-of-order execution that can yield substantial performance gains. This technique is particularly relevant for multi-threaded workloads common in data centers and high-performance computing environments. CELLO highlights the power of software-hardware co-design, demonstrating how compiler-generated information can be leveraged to reduce hardware complexity and energy consumption without sacrificing performance. Helios expands the scope of instruction fusion, proving that relaxing traditional fusion constraints can unlock significant gains in pipeline efficiency. Lastly, UCP addresses the persistent issue of frontend stalls in workloads with large instruction footprints, offering a path forward for processors that need to handle massive datasets in real-time.

Future Works:

By reordering memory operations to lay them out consecutively in the code, designers can eliminate the prediction mechanism needed for non-consecutive fusion, as most non-consecutive pairs will be converted into consecutive pairs. Another potential improvement is to use the compiler to predict which memory operations can be fused and pass these hints to the processor. The processor can then try to fuse the corresponding memory operations at decode. Furthermore, the Helios design shows that fusing only memory pairs, rather than all instructions, can be more beneficial. This insight opens up new possibilities for exploring criticality-driven fusion by focusing on pairs that are critical to processor performance.

UCP proposed in this thesis leverages hard-to-predict (H2P) branch information. However, the overall coverage and accuracy of current hard-to-predict branch predictors remain limited. Future work could explore more sophisticated predictors and techniques such as machine learning to improve the prediction accuracy. Profile-guided optimizations can also be used to identify critical branches

and provide hints to the processor.

Bibliography

[1] Touppercase78/intel-processors: Datasets for all processors maufactured by intel. https://github.com/toUpperCase78/intel-processors. [Online; accessed 3-Sep-2024].

[2] The 2nd data prefetching championship (dpc-2), June 2015.

[3] *ARM Architecture Reference Manual ARMv8-A*, 2015.

[4] Amd's zen 4 part 1: Frontend and execution engine. https://chipsandcheese.com/2022/11/05/amds-zen-4-part-1-frontend-and-execution-engine/, November 2022.

[5] ChampSim simulator, develop branch. https://github.com/ChampSim/ChampSim/tree/develop, November 2022.

[6] Intel says Moore's Law is still alive and well. Nvidia says it's ended. — cnbc.com. https://www.cnbc.com/2022/09/27/intel-says-moores-law-is-still-alive-nvidia-says-its-ended.html, September 2024.

[7] The Death of Moore's Law: What it means and what might fill the gap going forward — cap.csail.mit.edu. https://cap.csail.mit.edu/death-moores-law-what-it-means-and-what-might-fill-gap-going-forward, September 2024.

[8] Advanced Micro Devices. Software Optimization Guide for AMD EPYC™ 7003 Processors, Pub 56665, Rev 3. [Online; accessed Aug.-2023].

[9] Advanced Micro Devices. Software Optimization Guide for AMD EPYC™ 7003 Processors, Pub 56665, Rev 3. [Online; accessed Apr.-2022].

[10] Advanced RISC Machines. Arm® Cortex™-A77 Core Software Optimization Guide, Issue 3. [Online; accessed Apr.-2022].

[11] Advanced RISC Machines. Arm® Neoverse™-N2 Core Software Optimization Guide, issue 3. [Online; accessed Apr.-2022].

[12] Sarita V. Adve and Mark D. Hill. Weak ordering – a new definition. In *17th Int'l Symp. on Computer Architecture (ISCA)*, pages 2–14, June 1990.

[13] Niket Agarwal, Tushar Krishna, Li-Shiuan Peh, and Niraj K. Jha. GARNET: A detailed on-chip network model inside a full-system simulator. In *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, pages 33–42, April 2009.

[14] Truls Asheim, Boris Grot, and Rakesh Kumar. A storage-effective btb organization for servers. In *29th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 1153–1167. IEEE, 2023.

[15] Grant Ayers, Nayana Prasad Nagendra, David I. August, Hyoun Kyu Cho, Svilen Kanev, Christos Kozyrakis, Trivikram Krishnamurthy, Heiner Litz, Tipp J Moseley, and Parthasarathy Ranganathan. Asmdb: Understanding and mitigating front-end stalls in warehouse-scale computers. In *46th Int'l Symp. on Computer Architecture (ISCA)*, pages 462–473, June 2019.

[16] Mohammad Bakhshalipour, Mehran Shakerinava, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. Bingo spatial data prefetcher. In *25th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 399–411, February 2019.

[17] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *17th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 72–81, October 2008.

[18] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations. In *Conf. on Supercomputing (SC)*, pages 52:1–52:12, November 2011.

[19] Christopher Celio, Palmer Dabbelt, David A Patterson, and Krste Asanović. The renewed case for the reduced instruction set computer: Avoiding isa bloat with macro-op fusion for risc-v. *arXiv preprint arXiv:1607.02318*, 2016.

[20] George Z. Chrysos and Joel S. Emer. Memory dependence prediction using store sets. In *25th Int'l Symp. on Computer Architecture (ISCA)*, pages 142–153, June 1998.

[21] Advanced Micro Devices. Software Optimization Guide for AMD Zen4 Microarchitecture, Pub 57647, Rev 1, section 2.9.1, January 2023. [Online; accessed Nov.-2023].

[22] Martin Dixon, Per Hammarlund, Stephan Jourdan, and Ronak Singhal. The next-generation Intel core microarchitecture. *Intel Technology Journal*, 14(3):8–28, March 2010.

[23] Jack Doweck, Wen-Fu Kao, Allen Kuan-yu Lu, Julius Mandelblat, Anirudha Rahatekar, Lihu Rappoport, Efraim Rotem, Ahmad Yasin, and Adi Yoaz. Inside 6th-generation intel core: New microarchitecture code-named skylake. *IEEE Micro*, 37(2):52–62, 2017.

[24] Yuelu Duan, David Koufaty, and Josep Torrellas. Scsafe: Logging sequential consistency violations continuously and precisely. In *22nd Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 249–260, March 2016.

[25] Josué Feliu, Arthur Perais, Daniel A. Jiménez, and Alberto Ros. Rebasing microarchitectural research with industry traces. In *2023 IEEE International Symposium on Workload Characterization (IISWC)*, pages 100–114, 2023.

[26] Josué Feliu, Alberto Ros, Manuel E. Acacio, and Stefanos Kaxiras. ITSLF: Inter-Thread Store-to-Load Forwarding in Simultaneous Multithreading. In *54th Int'l Symp. on Microarchitecture (MICRO)*, pages 1296–1308, October 2021.

[27] Michael Ferdman, Almutaz Adileh, Yusuf Onur Koc̨berber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *17th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, pages 37–48, March 2012.

[28] Cormac Flanagan and Stephen N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *2009 Conf. on Programming Language Design and Implementation (PLDI)*, pages 121–133, June 2009.

[29] Alok Garg, Castro Fernando, Huang Michael, Daniel Chaver, Luis Pinuel, and Manuel Prieto. Substituting associative load queue with simple hash tables in out-of-order microprocessors. In *Proceedings of the 2006 international symposium on Low power electronics and design*, pages 268–273, October 2006.

[30] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Two techniques to enhance the performance of memory consistency models. In *20th Int'l Conf. on Parallel Processing (ICPP)*, pages 355–364, August 1991.

[31] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. Persistency for synchronization-free regions. In *39th Conf. on Programming Language Design and Implementation (PLDI)*, pages 46–61, June 2018.

[32] Antonio Gonzalez, Fernando Latorre, and Grigorios Magklis. *Processor microarchitecture: An implementation perspective*, volume 12. Morgan & Claypool Publishers, 2010.

[33] Brian Grayson, Jeff Rupley, Gerald D. Zuraski, Eric Quinnell, Daniel A. Jiménez, Tarun Nakra, Paul Kitchin, Ryan Hensley, Edward Brekelbaum, Vikas Sinha, and Ankit Ghiya. Evolution of the samsung exynos cpu microarchitecture. In *47th Int'l Symp. on Computer Architecture (ISCA)*, pages 40–51, June 2020.

[34] Vishal Gupta and Biswabandan Panda. Micro btb: A high performance and storage efficient last-level branch target buffer for servers. In *Proceedings of the 19th ACM International Conference on Computing Frontiers*, pages 12–20, 2022.

[35] Nikos Hardavellas, Ippokratis Pandis, Ryan Johnson, Naju Mancheril, Anastassia Ailamaki, and Babak Falsafi. Database servers on chip multiprocessors: Limitations and opportunities. In *Proceedings of the Biennial Conference on Innovative Data Systems Research*, 2007.

[36] ARM Inc. Arm® Neoverse™ V2 Core Technical Reference Manual, revision r0p2, Section 3.1, December 2022. [Online; accessed Nov.-2023].

[37] Intel. Intel® 64 and IA-32 Architectures Optimization Reference Manual, Pub 248966-045. [Online; accessed Apr.-2022].

[38] Intel. Intel® 64 and ia-32 architectures optimization reference manual. `www.intel.com`, June 2016.

[39] Intel Corporation, White paper. *First the Tick, Now the Tock: Next Generation Intel Microarchitecture (Nehalem)*, April 2009.

[40] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a

warehouse-scale computer. In *42nd Int'l Symp. on Computer Architecture (ISCA)*, pages 158–169, June 2015.

[41] Richard E Kessler, Edward J McLellan, and David A Webb. The alpha 21264 microprocessor architecture. In *Proceedings International Conference on Computer Design. VLSI in Computers and Processors (Cat. No. 98CB36273)*, pages 90–95, October 1998.

[42] Tanvir Ahmed Khan, Nathan Brown, Akshitha Sriraman, Niranjan K Soundararajan, Rakesh Kumar, Joseph Devietti, Sreenivas Subramoney, Gilles A Pokam, Heiner Litz, and Baris Kasikci. Twig: Profile-guided BTB Prefetching for Data Center Applications. In *54th Int'l Symp. on Microarchitecture (MICRO)*, pages 816–829, 2021.

[43] Joonsung Kim, Hamin Jang, Hunjun Lee, Seungho Lee, and Jangwoo Kim. Uc-check: Characterizing micro-operation caches in x86 processors and implications in security and performance. In *54th Int'l Symp. on Microarchitecture (MICRO)*, pages 550–564, 2021.

[44] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. Language-level persistency. In *44th Int'l Symp. on Computer Architecture (ISCA)*, pages 481–493, June 2017.

[45] Jagadish B Kotra and John Kalamatianos. Improving the utilization of micro-operation caches in x86 processors. In *53rd Int'l Symp. on Microarchitecture (MICRO)*, pages 160–172. IEEE, 2020.

[46] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers (TC)*, 28(9):690–691, September 1979.

[47] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd Int'l Symp. on Code Generation and mmization (CGO)*, pages 75–88, March 2004.

[48] Sheng Li, Ke Chen, Jung Ho Ahn, Jay B. Brockman, and Norman P. Jouppi. Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques. In *2011 Int'l Conf. on Computer-Aided Design (ICCAD)*, pages 694–701, November 2011.

[49] Milo M.K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and

David A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *ACM SIGARCH Computer Architecture News*, 33(4):92–99, September 2005.

[50] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0*, June 2021.

[51] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114—117, April 1965.

[52] Andreas Moshovos and Gurindar S. Sohi. Streamlining inter-operation memory communication via data dependence prediction. In *30th Int'l Symp. on Microarchitecture (MICRO)*, pages 235–245, December 1997.

[53] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. Cuda, release: 10.2.89, 2020.

[54] OpenMP tutorial. `http://computing.llnl.gov/tutorials/openMP`, April 2015.

[55] Irma E. Papazian. New 3rd gen Intel® Xeon® Scalable processor (Codename: Ice Lake-SP). In *32nd HotChips Symp.*, pages 1–22, August 2020.

[56] Irma Esmer Papazian. New 3rd gen Intel Xeon Scalable processor (codename: Ice Lake-SP). In *32nd HotChips Symp.*, pages 1–22, August 2020.

[57] Dharmesh Parikh, Kevin Skadron, Yan Zhang, and Mircea Stan. Power-aware branch prediction: Characterization and design. *IEEE Transactions on Computers*, 53(2):168–186, 2004.

[58] Arthur Perais and Rami Sheikh. Branch target buffer organizations. In *56thInt'l Symp. on Microarchitecture (MICRO)*, New York, NY, USA, 2023. Association for Computing Machinery.

[59] Glenn Reinman, Todd M. Austin, and Brad Calder. A scalable front-end architecture for fast instruction delivery. In *26th Int'l Symp. on Computer Architecture (ISCA)*, pages 234–245, May 1999.

[60] Glenn Reinman, Brad Calder, and Todd Austin. Fetch directed instruction prefetching. In *32nd Int'l Symp. on Microarchitecture (MICRO)*, December 1999.

[61] Xida Ren, Logan Moody, Mohammadkazem Taram, Matthew Jordan, Dean M Tullsen, and Ashish Venkat. I see dead μops: Leaking secrets via intel/amd micro-op caches. In *48th Int'l Symp. on Computer Architecture (ISCA)*, pages 361–374. IEEE, 2021.

[62] RISC-V Software. Spike RISC-V ISA Simulator.

[63] Alberto Ros, Trevor E. Carlson, Mehdi Alipour, and Stefanos Kaxiras. Non-speculative load-load reordering in tso. In *44th Int'l Symp. on Computer Architecture (ISCA)*, pages 187–200, June 2017.

[64] Efraim Rotem, Adi Yoaz, Lihu Rappoport, Stephen J. Robinson, Julius Yuli Mandelblat, Arik Gihon, Eliezer Weissmann, Rajshree Chabukswar, Vadim Basin, Russell Fenger, Monica Gupta, and Ahmad Yasin. Intel Alder Lake CPU architectures. *IEEE Micro*, 42(3):13–19, March 2022.

[65] Efraim Rotem, Adi Yoaz, Lihu Rappoport, Stephen J. Robinson, Julius Yuli Mandelblat, Arik Gihon, Eliezer Weissmann, Rajshree Chabukswar, Vadim Basin, Russell Fenger, Monica Gupta, and Ahmad Yasin. Intel Alder Lake CPU architectures. *IEEE Micro*, 42(3):13–19, March 2022.

[66] Christos Sakalis, Carl Leonardsson, Stefanos Kaxiras, and Alberto Ros. Splash-3: A properly synchronized benchmark suite for contemporary research. In *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, pages 101–111, April 2016.

[67] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, July 2010.

[68] André Seznec. A 256 kbits L-TAGE branch predictor. *Journal of Instruction-Level Parallelism (JILP) Special Issue: The Second Championship Branch Prediction Competition (CBP-2)*, pages 1–6, December 2007.

[69] André Seznec. The L-TAGE branch predictor. *The Journal of Instruction-Level Parallelism*, 9:1–13, May 2007.

[70] André Seznec. A 64-Kbytes ITTAGE indirect branch predictor. In *2nd JILP Workshop on Computer Architecture Competitions (JWAC-2): Championship Branch Prediction*, June 2011.

[71] André Seznec. Storage free confidence estimation for the tage branch predictor. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 443–454. IEEE, 2011.

[72] André Seznec. TAGE-SC-L branch predictors again. In *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, June 2016.

[73] André Seznec. Tage-sc-l branch predictors again. In *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, 2016.

[74] André Seznec, Stephen Felix, Venkata Krishnan, and Yiannakis Sazeides. Design tradeoffs for the alpha ev8 conditional branch predictor. In *29th Int'l Symp. on Computer Architecture (ISCA)*, pages 295–306, May 2002.

[75] André Seznec and Pierre Michaud. A case for (partially) tagged geometric history length branch prediction. *Journal of Instruction-Level Parallelism (JILP)*, page 23, February 2006.

[76] Baruch Solomon, Avi Mendelson, Doron Orenstein, Yoav Almog, and Ronny Ronen. Micro-operation cache: a power aware frontend for the variable instruction length isa. In *Proceedings of the 2001 international symposium on Low power electronics and design*, pages 4–9, 2001.

[77] Yulei Sui, Peng Di, and Jingling Xue. Sparse flow-sensitive pointer analysis for multithreaded programs. In *14th Int'l Symp. on Code Generation and mmization (CGO)*, pages 160–170, March 2016.

[78] Amanda Tomlinson and George Porter. Something old, something new: Extending the life of cpus in datacenters. *ACM SIGENERGY Energy Informatics Review*, 3(3):59–63, 2023.

[79] Andrew Waterman and Krste Asanović. *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA*. RISC-V Foundation, December 2019.

[80] Wikipedia. Advanced Vector Extensions — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Advanced%20Vector%20Extensions&oldid=1223962098`, 2024. [Online; accessed 17-May-2024].

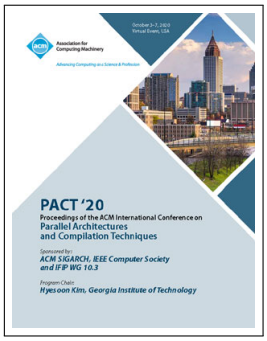
[81] Wikipedia. Intel 4004 — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Intel%204004&oldid=1222173429`, 2024. [Online; accessed 17-May-2024].

[82] Wikipedia. RISC-V — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=RISC-V&oldid=1223296280`, 2024. [Online; accessed 21-May-2024].

[83] Tse-Yu Yeh and Yale N Patt. Two-level adaptive training branch prediction. In *24th Int'l Symp. on Microarchitecture (MICRO)*, pages 51–61, November 1991.

Publications Composing the Thesis

1 Regional Out-of-Order Writes in Total Store Order



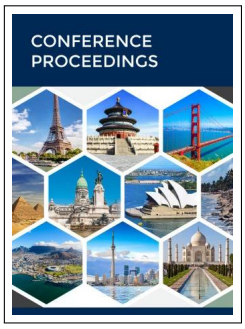
Name:	Regional Out-of-Order Writes in Total Store Order
Authors:	**Singh, Sawan** and Jimborean, Alexandra and Ros, Alberto
Conference:	Parallel Architectures and Compilation Techniques (PACT)
Place:	Virtual
Raking:	A-
Publisher:	Association for Computing Machinery (ACM)
Year:	2020
Month:	October
DOI:	https://doi.org/10.1145/3410463.3414645
Status:	Published

Abstract

The store buffer, an essential component in today's processors, is designed to hide memory latency by moving stores off the processor's critical path. Furthermore, under the Total Store Order (TSO) memory model, the store buffer ensures the in-order retirement of stores. Problems arise when the store buffer is full or, under TSO, when the leading store encounters a cache miss, which blocks all subsequent stores and incurs severe performance bottlenecks.This work presents a software-hardware co-designed approach to cope with this bottleneck for processors with strong consistency guarantees. Our proposal is driven by the insight that store operations can be reordered if their reordering does not change the observable program behavior. The compiler delineates safe regions within

which stores can be shuffled while still delivering the same observable behavior as if they performed in program order and unsafe regions within which stores must be kept in program order. This is leveraged by a novel dual-mode store buffer that switches between the out-of-order and in-order execution of stores within the safe and respectively unsafe regions. Correctness is preserved through well-placed fences inserted by the compiler, which impede the execution of stores from the following regions until all stores of the current region complete. Our dual-mode store buffer only requires one extra bit per entry, significantly decreases processor stall cycles, and brings 8.13% performance improvements compared to a mainstream store buffer.

2 CELLO: Compiler-Assisted Efficient Load-Load Ordering in Data-Race-Free Regions



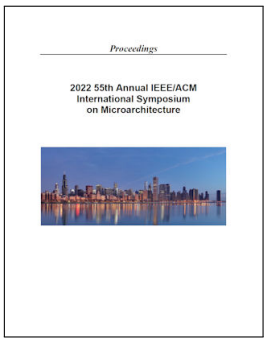
Name:	CELLO: Compiler-Assisted Efficient Load-Load Ordering in Data-Race-Free Regions
Authors:	**Singh, Sawan** and Feliu, Josue and Acacio, Manuel E. and Jimborean, Alexandra and Ros, Alberto
Conference:	Parallel Architectures and Compilation Techniques (PACT)
Place:	Vienna, Austra
Ranking:	A-
Publisher:	Institute of Electrical and Electronics Engineers (IEEE)
Year:	2023
Month:	October
DOI:	https://doi.org/10.1109/PACT58117.2023.00009
Status:	Published

Abstract

Efficient Total Store Order (TSO) implementations allow loads to execute speculatively out-of-order. To detect order violations, the load queue (LQ) holds all the in-flight loads and is searched on every invalidation and cache eviction. Moreover, in a simultaneous multithreading processor (SMT), stores also search the LQ when writing to cache. LQ searches entail considerable energy consumption. Furthermore, the processor stalls upon encountering the LQ full or when its ports are busy. Hence, the LQ is a critical structure in terms of both energy and performance. In this work, we observe that the use of the LQ could be dramatically optimized under the guarantees of the datarace-free (DRF) property

imposed by modern programming languages. To leverage this observation, we propose CELLO, a software-hardware co-design in which the compiler detects memory operations in DRF regions and the hardware optimizes their execution by safely skipping LQ searches without violating the TSO consistency model. Furthermore, CELLO allows removing DRF loads from the LQ earlier, as they do not need to be searched to detect consistency violations. With minimal hardware overhead, we show that an 8-core 2-way SMT processor with CELLO avoids almost all conservative searches to the LQ and significantly reduces its occupancy. CELLO allows i) to reduce the LQ energy expenditure by 33% on average (up to 53%) while performing 2.8% better on average (up to 18.6%) than the baseline system, and ii) to shrink the LQ size from 192 to only 80 entries, reducing the LQ energy expenditure as much as 69% while performing on par with a mainstream LQ implementation.

3 Exploring Instruction Fusion Opportunities in General Purpose Processors



Name:	Exploring Instruction Fusion Opportunities in General Purpose Processors
Authors:	**Singh, Sawan** and Perais, Arthur and Jimborean, Alexandra and Ros, Alberto
Conference:	IEEE/ACM International Symposium on Microarchitecture (MICRO)
Place:	Chicago, USA
Ranking:	A
Publisher:	Institute of Electrical and Electronics Engineers (IEEE)
Year:	2022
Month:	October
DOI:	https://doi.org/10.1109/MICRO56248.2022.00026
Status:	Published

Abstract

The Complex Instruction Set Computer (CISC) paradigm has led to the introduction of instruction cracking in which an architectural instruction is divided into multiple microarchitectural instructions (μ-ops). However, the dual concept, instruction fusion is also prevalent in modern microarchitectures to maximize resource utilization. In essence, some architectural instructions are too complex to be executed as a unit, so they should be cracked, while others are too simple to waste resources on executing them as a unit, so they should be fused with others. In this paper, we focus on instruction fusion and explore opportunities for fusing additional instructions in a high-performance general purpose pipeline. We show that enabling fusion for common RISC-V idioms improves performance

by 7%. Then, we determine experimentally that enabling fusion only for memory instructions achieves 86% of the potential of fusion in this particular case. Finally, we propose the Helios microarchitecture, able to fuse non-consecutive and noncontiguous memory instructions, and discuss microarchitectural changes required to do so efficiently while preserving correctness. Helios allows to fuse an additional 5.5% of dynamic instructions, yielding a 14.2% performance uplift over no fusion (8.2% over baseline fusion).

4 Alternate Path μ-op Cache Prefetching

Name:	Alternate Path μ-op Cache Prefetching
Authors:	**Singh, Sawan** and Perais, Arthur and Jimborean, Alexandra and Ros, Alberto
Conference:	International Symposium on Computer Architecture (MICRO)
Place:	Buenos Aires, Argentina, USA
Ranking:	A++
Publisher:	Institute of Electrical and Electronics Engineers (IEEE)
Year:	2024
Month:	July
DOI:	https://doi.org/10.1109/MICRO56248.2022.00026
Status:	Published

Abstract

Datacenter applications are well-known for their large code footprints. This has caused frontend design to evolve by implementing decoupled fetching and large prediction structures – branch predictors, Branch Target Buffers (BTBs) – to mitigate the stagnating size of the instruction cache by prefetching instructions well in advance. In addition, many designs feature a micro operation (µ-op) cache, which primarily provides power savings by bypassing the instruction cache and decoders once warmed up. However, this µ-op cache often has lower reach than the instruction cache, and it is not filled up speculatively using the decoupled fetcher. As a result, the µ-op cache is often over-subscribed by datacenter applications, up to the point of becoming a burden.

This paper first shows that because of this pressure, blindly prefetching into the µ-op cache using state-of-the-art standalone prefetchers would not provide

significant gains. As a consequence, this paper proposes to prefetch only critical µ-ops into the µ-op cache, by focusing on execution points where the µ-op cache provides the most gains: Pipeline refills. Concretely, we use hard-to-predict conditional branches as indicators that a pipeline refill is likely to happen in the near future, and prefetch into the µ-op cache the µ-ops that belong to the path opposed to the predicted path, which we call *alternate* path. Identifying hard-to-predict branches requires no additional state if the branch predictor confidence is used to classify branches. Including extra *alternate* branch predictors with limited budget (8.95KB to 12.95KB), our proposal provides average speedups of 1.9% to 2% and as high as 12% on a subset of CVP-1 traces.