Precise Characterization of Coherence Activity in Multicores using gem5

Joaquín Ferrer, Juan M. Cebrian, Ricardo Fernández-Pascual and Manuel E. Acacio

Computer Architecture and Parallel Systems Group, University of Murcia, Spain.

*Corresponding author(s). E-mail(s): joaquin.ferrerc@um.es; jcebrian@um.es; rfernandez@um.es; meacacio@um.es;

Abstract

Simulation enables cost-effective and rapid prototyping in computer architecture research. It helps assess the impact of architectural changes on performance, area, and energy consumption, playing a crucial role in early-stage development.

Gem5 has become a widely used simulation tool in academia and industry for researching multicore architectures. However, its accuracy depends on proper configuration. Key parameters, such as core microarchitecture, memory hierarchy, and interconnection network, must be carefully calibrated to ensure realistic results.

This work highlights the importance of a well-adjusted simulation environment for modeling modern multicore setups, with a focus on coherence directory. We refine core, memory, and interconnection parameters, identifying and addressing deficiencies in the simulation infrastructure. We introduce new functionalities and statistics to enhance system characterization. We implement Intel's Top-Down methodology in gem5, extending it with two new levels to analyze coherence activity's impact on performance. Lastly, we enable gem5 to support various sparse directory architectures.

Keywords: Multicores, gem5, cache coherence, Top-Down

1 Introduction

In the field of computer architecture research, simulation tools serve as indispensable instruments for evaluating new designs, predicting system performance, and exploring architectural trade-offs. As real-world experimentation with cutting-edge hardware is often impractical due to cost, complexity, or limited availability, the use of simulators provides researchers with a flexible and powerful alternative. Widely adopted by academia and industry alike (over 6000 citations), gem5 [1] offers the ability to model various hardware components, including CPUs, memory hierarchies, interconnects, and system configurations, across a range of architecture types (e.g. Arm or x86). Its modularity and configurable capabilities make it an ideal choice for simulating both simple and complex systems, and it is currently the *de facto* standard for conducting research in multicore architectures, which is the focus of this work.

However, the effectiveness of any simulation tool is contingent upon the accuracy of its configuration and tuning. A poorly calibrated simulation environment can generate skewed results that misrepresent the performance characteristics of the system being modeled. This can lead to incorrect conclusions, misguided optimizations, and even failed hardware designs, especially in research settings where performance predictions drive key decision-making processes. In particular, in a multicore setup, the microarchitectural parameters, the memory hierarchy, and the interconnection network must be carefully adjusted to reflect realistic conditions and target architectures.

Moreover, the complexity of modern systems has grown dramatically, combining diverse processing units, accelerators, and memory subsystems. This growing complexity further amplifies the need for precision in simulation. Small misconfigurations in a tool like gem5 can result in substantial deviations from real-world performance, potentially misleading researchers about the true efficiency or bottlenecks in their designs. Hence, the need for careful tuning of simulators becomes critical, not only to avoid erroneous conclusions, but also to ensure that the results are valid, reproducible, and meaningful in the context of actual hardware implementations.

This work emphasizes the importance of meticulous simulator tuning, with a specific focus on gem5, and examines common challenges that researchers face in configuring simulations for modeling a multicore setup. By analyzing real-world case studies and exploring best practices, the goal is to illustrate how properly calibrated simulation environments contribute to more accurate research outcomes, while also offering guidelines for improving simulation reliability. Ultimately, a well-tuned simulator is not just a research tool, but a safeguard against imprecise results that could have lasting repercussions in both academic and industrial contexts.

In the case of gem5, we will study and tune the configuration applied to several simulated components such as cores, branch predictors, Translation Lookaside Buffers (TLBs), caches, and interconnection network. In the process of tuning these structures, we identify and correct some bugs found in the simulator and also add new functionalities implemented in modern processors and statistics to better characterize the performance of the modeled system. To have an additional perspective when analyzing system performance, we will also implement Intel's Top-Down methodology [2] and extend it with two new levels of analysis to better understand the impact that coherence activity has on performance. Finally, we also extend gem5 with a variety

of sparse directory architectures and prove how important tuning the simulator is to obtain realistic and valid conclusions about the modeled systems. We plan to release[1] the patches needed to fix the deficiencies found in gem5 during our analysis, and also, the additional functionality developed as part of this work. As the result of this work, we provide a modified version of gem5 where we have used an extended version of the Top-Down method in order to detect and improve any misconfiguration in the system to be able to model a contemporary architecture. This simulator is ready to be used to conduct evaluations over different coherence directory designs.
The principal contributions of this work are as follows:

- We perform a deep fine-tuning of gem5, applying several bug fixes and adding new features to model contemporary multicore architectures.
- We implement Intel's Top-Down methodology in gem5 and extend it to characterize coherence activity in a multicore setup using parallel applications.
- We add support for configurable sparse directory organizations and sharer codifications.

2 Background

2.1 Simulation of multicore architectures

Based on the level of detail, simulation infrastructures can be classified as: cycle-level, timing/functional, and analytical modeling, from more to less accurate. Additionally, based on the workload used to simulate, we can distinguish between execution-driven, trace-based, or full-system simulators.

Cycle-level simulators are complex pieces of software that model every part of the system with a high level of detail. However, this implies long simulation times. On the other hand, timing simulators and functional modeling are not as detailed. Still, they offer a good trade-off between accuracy and simulation speed, since most of the application instructions are processed in a simplified way. They serve to explore the design space and to enable software development on unavailable hardware. In addition, hybrid methodologies, such as interval simulation, provide a balance between cycle-accurate and functional simulation. Finally, analytical modeling uses mathematical formulas to model the performance and/or energy consumption of the architecture. We focus our attention on cycle-level simulation of multicore architectures, with particular emphasis on modeling coherence activity.

Among the initial efforts to provide detailed execution-driven cycle-level simulation of parallel infrastructures, we highlight RSIM [3]. In particular, this simulation platform was aimed at modeling shared memory multiprocessors made up of aggressive superscalar, out-of-order processors. It allowed detailed simulation of the processor microarchitecture, the cache hierarchy, the memory system (including a directory-based cache coherence protocol), and the interconnection network. Although this simulator was popular in the early 2000s for research on cache coherence (e.g. [4, 5]), it

[1] A version of the enhanced gem5 simulator can be found at https://github.com/joaquin-ferrerc-um-es/gem5

was focused on multiprocessor setups with single-core processors and was discontinued long ago.

Instead of an execution-driven approach, other cycle-level simulators adopt trace-driven simulation, which uses traces obtained in a real system as inputs. An example of this kind is ZSim [6], an x86-64 simulator aimed at making thousand-core simulation practical by speeding up and parallelizing the simulation process. This tool has been used in several research works targeting cache coherence, such as [7]. Nevertheless, ZSim does not model contention on the interconnection network, which can be a critical aspect when evaluating new organizations of the coherence directory information.

Full-system simulation also considers the activity of the operating system. GEMS [8], which was the cycle-level simulator precursor to gem5, combined the SIMICS functional simulator [9] with detailed models for the processor, memory hierarchy (including cache coherence), and interconnection network. Since its release, it has been extensively used by the research community to study cache coherence in multicores (e.g. [10–13]). Subsequently, gem5 replaced the proprietary SIMICS functional simulator with the open-source m5 simulator[14] and extended the capabilities of GEMS by enabling several architectures (e.g. x86 and Arm) and network models, among other things. This has enlarged the community behind its development and use, becoming the *de facto* standard in academic research (and part of industry research) on computer architecture in general, and multicore architectures in particular (over 6000 citations and counting).

In gem5, we use the *Ruby* memory system simulator. With *Ruby* we can model the cache hierarchy, the cache replacement policies, the coherence protocols, the interconnection network, and the memory controllers. All this modeling is completely modular and configurable, so we can set the memory subsystem configuration to suit our needs. Within *Ruby* we use the *SLICC* (*Specification Language for Implementing Cache Coherence*) language, which is a domain-specific language designed to specify the behaviour of the cache controllers that implement the coherence protocol. After all, the behaviour of a coherence protocol can be represented by a state machine per memory address, so with *SLICC* we specify the behaviour of these state machines inside every cache controller, indicating how to act on each possible event (a network message, a petition from the core), making the necessary transitions, and generating the necessary requests and responses to other agents.

In addition, *SLICC* ties together several components of the memory model, as the state machine takes its inputs from the input ports of the interconnection network, processes these events in the corresponding memory and cache controllers, and enqueues the generated requests and messages in the output ports of the interconnection network. In this way, *SLICC* and *Ruby* attempt to simulate the behaviour of the real hardware memory subsystem as realistically and closely as possible.

Finally, the Sniper multicore simulator [15] is an x86 simulation tool based on the interval core model and the Graphite simulation infrastructure [16], allowing fast and accurate simulation and trading off simulation speed for accuracy to allow a range of flexible simulation options when exploring different homogeneous and heterogeneous multicore architectures. Its lack of cycle-accuracy for the processor model

makes it unsuitable for studying the impact of modifications to the cache coherence infrastructure on performance.

In this work, we focus on gem5. Although its simulation time increases compared to other alternatives, it models most of the system components with a high level of detail, including the core architecture, cache and memory subsystems, interconnection network, etc. Therefore, one can achieve high accuracy in this simulator, if the different structures are set up properly.

As we will see in more detail later on, the configuration of these structures plays a crucial role when it comes to configuring the simulator to be representative of real contemporary multicore architectures. For example, a misconfiguration of the TLB size may prevent us for setting realistic cache access latencies. Cache latencies also need to be corrected, as a bug in the simulator was found that added an extra cycle of latency to each cache access. In addition, the coherence directory structure should be modified to match how it is implemented in most real systems. Finally, we will also point out the importance of a correct modelling of the interconnection network that does not overestimate its performance. If the network is able to absorb any increase in traffic without due consideration of the resulting contention, incorrect results and erroneous conclusions about a given implementation may be obtained.

We first review the current state of gem5 with bibliographical references that discuss features and fixes to improve accuracy, including Gutierrez *et al.* [17], Butko *et al.* [18], Akram *et al.* [19], Walker *et al.* [20] and Cebrian *et al.* [21], especially those related to the memory subsystem.

2.2 Characterization

A key aspect to evaluate the modeled system are the statistics provided by the simulation tool. Gem5 provides several statistics to evaluate execution times, the use of different structures such as caches, a breakdown of the interconnection network traffic, and many other aspects of the system. However, to have a first approach to evaluate the performance of the system and identify the roots of potential inefficiencies, we implemented and extended the Top-Down model in gem5.

This methodology, proposed by Yasim *et al.* [2], is described by the authors as "a practical method to quickly identify true bottlenecks in out-of-order processors". The key aspect of this method is its hierarchical division of the performance metrics, which helps to accurately identify, step by step, which area of the system is responsible for performance loss. For these performance metrics, they use the Performance Monitoring Unit (PMU) commonly included in most modern CPUs.

Among all the levels of detail available in the Top-Down methodology, we will focus on three of them:

- **Top Level**. This is the first level and divides the status of the issued μops into four categories: retiring category represents μops finishing normally and leaving the pipeline; bad-speculation represents μops squashed due to a mispredicted branch; the frontend-bound category represents the ratio of μops stalled in fetch/decode stages, while the backend-bound covers ready-to-issue μops that cannot continue along the pipeline due to a lack of resources.

- **Backend Level**. This level divides the backend-bound into two categories: CPU and memory-bound, depending on the resources causing these stalls.
- **Memory Level**. This level breaks down the memory bound from the Backend Level into L1-bound, L2-bound, L3-bound, External Memory bound, and Store bound, again depending on the memory structure causing the stall.

Figure 1 shows the hierarchical organization of these three levels, with their categories. The original Top-Down methodology was focused on a single-core setup. As explained later on, one of the contributions of this work is the extension of this methodology to analyze coherence activity in a multicore architecture.

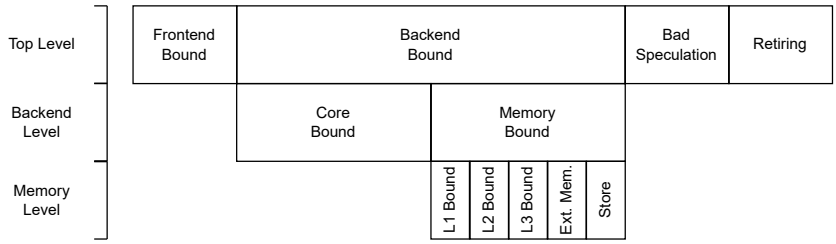


Fig. 1 Hierarchical division of the Top-Down levels and categories.

2.3 Coherence directory designs

When maintaining cache coherence in large systems, directory-based protocols [22] scale significantly better than snooping-based protocols [23], as they use less traffic over the interconnection network and, more importantly, they do not require a fully ordered network. These protocols use a directory structure, with one entry per data block. These entries store the state of the block in the directory (which depends on the states of that block in each of the private caches), and the set of nodes that own a copy of the block. This list of sharers of the block is updated as coherence events unfold between nodes, adding and removing nodes from the list as they request and invalidate the block in their caches. This directory information is used to only forward coherence messages to the nodes that actually have a copy of the block, reducing traffic in the network. It also serves as a serialization point.

Different approaches have been considered for storing directory information. Traditionally, directory information was stored in memory, extending it to store the directory state of all the blocks. This, however, is not practical in modern systems. We consider two main designs:

- Directory information embedded in an inclusive Last Level Cache (LLC): this is the simplest design. By simply adding extra bits to each LLC block, we can store the directory information we need, embedding it in the LLC. However, these extra bits on every LLC entry (which will be mostly unused) and LLC inclusion may make this design not the best choice.

- Standalone inclusive directory cache (also known as sparse directory [24]): this design does not rely on LLC inclusion. Now, the directory cache is a separate structure, as a normal cache. In this case, the LLC does not need to be inclusive, but the directory cache has to. It must contain the directory information for all the blocks being cached in the private upper-level caches to keep coherence. This requires duplicating all the tags of the blocks stored in the private caches. A cache block tag is a unique identifier of that block. It consists of the most significant bits of the memory address. From that address, we take the least significant bits to get the block address, and the set of the cache in which the block is stored, the remaining bits constitute the block tag. This design is more flexible since we don't need the LLC to be inclusive, but we have the extra storage for the duplicated tags.

With a standalone inclusive directory cache, we have a new concept called coverage. The coverage of a directory cache is the percentage of private cache entries that can fit their directory information in the directory cache. When a cache entry cannot store its directory information due to insufficient space in the directory cache, this causes a directory cache replacement, and as a consequence of a directory entry replaced, the copies of the corresponding block need to be invalidated on every private cache (although there might be enough space on the LLC to store it), increasing the miss rate of the caches. Ideally, we would like to have a coverage of 100% (even a little more to avoid any conflict), so every private cache entry could have their directory information, but we can have a lower coverage of 80%, or 50%, saving space in the directory cache as we will need fewer entries, and still get good results, as not all private cache entries will need their directory information at the same time.

In many instances, most data blocks have (in practice) very few sharers, most often a single sharer. This is why one approach to reduce the size of the sharing information is to have only a limited number of pointers (instead of using full bit vectors). In this way, we can have P pointers, each with a length of $1 + \log_2 N$ bits, where N is the number of nodes in the system and each pointer has a valid bit. Thus, the complete sharing information occupies $P \times (1 + \log_2 N)$ bits for each cache block. In Figure 2 we can see a simplified example of what a directory entry with limited pointers would look like. In this approach, we must consider how to act when for a block we have P sharers and we want to add a new sharer so that they would become $P + 1$ sharers. We will consider using imprecise information resorting to a broadcast representation.

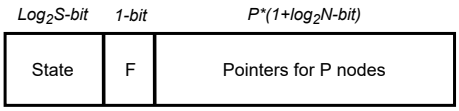


Fig. 2 Basic structure of a directory entry based on limited pointers in a system with S possible states of a block and N nodes.

The directory-based coherence protocols implemented in gem5 assume that the directory information is embedded in the LLC, in a bit-vector format. This results in over-provisioning of directory information (which is largely unused) and the need for the LLC to be inclusive. This also may differ from the way coherence directories are

implemented on a processor with a large number of cores, where a sparse directory would normally be preferred and also some imprecise representation for the sharers could also be used.

Another contribution of this work is the implementation of a *sparse directory organization and different imprecise sharing information representations and the evaluation of their impact on system performance.* At that point, we will *demonstrate the importance of a proper system configuration to obtain valid and realistic results when evaluating these implementations.* Without our particular configuration of the simulator, the results obtained could have led us to incorrect conclusions about the modeled designs.

3 Gem5 fine tuning and extensions

The main aim of our research was to design, implement, and study new directory organizations. While implementing and evaluating various state-of-the-art solutions in gem5, we found unexpected results with aggressive imprecise representations barely hurting system performance. From that point on, our main objective was to obtain a proper base system as a starting point for our research on directory organizations.

Before starting to configure any structure, we had to fix some bugs previously identified in the simulator. For example, Cebrian *et al.* [21] found that on every message to the memory subsystem, the gem5 O3CPU message queuing system added one extra cycle to the returning messages. Whereas this is correct with the simple memory model, in the case of Ruby, it already sends the response message with the correct timing, and no additional cycles are required. With this patch, we ensure that, if we are using Ruby for simulating the memory subsystem, we do not incur that extra cycle.

The first step we took was to configure gem5 using real hardware specifications as a reference, to help us understand the sources of inefficiencies when running parallel workloads on a multicore architecture. To develop this improved configuration, we considered the principal contributors to the final performance in a multicore setup: microarchitecture of the cores, cache configuration regarding latencies and sizes, cache coherence paying attention to the organization of the directory, and the interconnection network. In Fig. 3 we can see an example of the structure of a multicore processor with 8 cores, and its memory hierarchy: private L1 (divided into instruction and data) and L2 cache levels for each core, and shared LLC. These cores communicate with each other, and with the LLC, via the network-on-chip (NoC).

Even though we are considering complex and aggressive out-of-order cores that can easily hide specific component latencies, the particular configuration of the core model ultimately plays an important role in determining coherence activity, for example when it comes to increasing the data cache access latency without harming too much the performance. To this end, some new configurations and mechanisms may need to be implemented in the simulator. This is why we set up to configure the x86 core model included in gem5 according to the specifications of contemporary products. Similar problems happened with the network-on-chip. An idealized network may hide bottlenecks caused by the coherence protocol if it is capable of processing a excessive number of messages in the same cycle, especially when aggressive imprecise

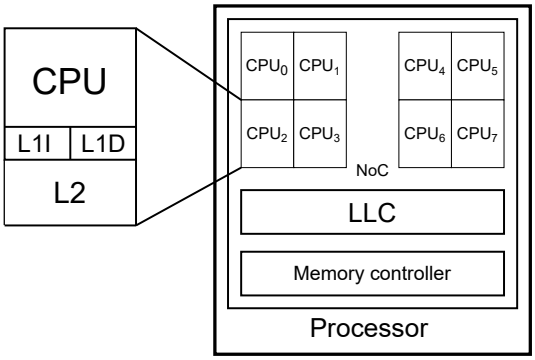


Fig. 3 Basic structure of a multicore processor with 8 cores.

sharer representations are assumed. We must solve these problems first to ensure a representative simulation setup.

We used the Top-Down model to summarize the micro-architecture performance counters components and to detect wrongly configured structures in the processor. We also run some microbenchmarks to stress both the cache and network to detect possible inappropriate configurations in these key components. Finally, to model a contemporary processor in gem5, we configured the core microarchitecture parameters taking into account the Golden Cove cores available in Intel Alderlake architecture.

3.1 Extended Top-Down

Top-Down metrics implemented so far focus on the backend and memory performance areas. However, we wanted to expand the analysis in two main ways. First of all, providing a broader view of the system and to be able to measure the percentage of the execution time in which the CPU is running or idle. The Top Level (and inner levels of the method) makes a division and classification of the executing cycles and the CPU running time and indicates which area causes more detentions, focusing on a single thread. However, in a multicore architecture, it is interesting to know how much time each core is idle waiting for other cores because of synchronization or maybe because it finished its work earlier.

For this first level, we propose to create *Level 0* or *Pre-Top Level* and take the overall process execution time and the number of cycles each core has executed. With these two statistics, we can obtain, for each core, how much of the overall execution time it has been idle and how much it has been executing and running code.

The second addition we made to the original Top-Down methodology is to generate a more detailed division of the L3 Bound on the Memory Level. In this case, we want to know if these L2 load misses are due to cache coherence or other causes. If the cause is cache coherence, when the L2 miss reaches the shared LLC, it will be required to forward the request to the exclusive node that holds this data. This way, we can know if the request is affected by coherence or not.

We also want to record the time needed to invalidate all possible copies of a data element when some node wants to write it (GetX or Upgrade request). If the sharing information is represented with an imprecise codification, we will be considering more

Table 1 Summary of the Top-Down metrics implemented and their correspondence with gem5 statistics.

Level 0 - Pre-Top Level	
Idle	((simTicks / 500) - cpu.numCycles) / (simTicks / 500)
Executing	cpu.numCycles / (simTicks / 500)
Top Level	
Slots	commitWidth * numCycles
Retiring	commitedOps / Slots
Bad Speculation	((instsIssued - commitedOps) + commitWidth * squashCycles) / Slots
Frontend Bound	(uopsNotDeliveredBlock + uopsNotDeliveredRun + uopsNotDelivered-Squash) / Slots
Backend Bound	1 - (Retiring + Bad Speculation + Frontend Bound)
Backend Level	
Bound At Exe	(iewExecuteStallCycles + (iewExecuteGE1 - iewExecuteGE2)) / num-Cycles
Core Bound	Bound At Exe - Memory Bound
Memory Bound	(iewExecuteStallAnyPendingCycles + renameBoundOnStores) / numCy-cles
Memory Level	
L1 Bound	(iewExecuteStallAnyPendingCycles - iewExecuteStallL1PendingCycles) / numCycles
L2 Bound	(iewExecuteStallL1PendingCycles - iewExecuteStallL2PendingCycles) / numCycles
L3 Bound	(iewExecuteStallL2PendingCycles - iewExecuteStallL3PendingCycles) / numCycles
Ext. Memory Bound	iewExecuteStallL3PendingCycles / numCycles
Stores Bound	renameBoundOnStores / numCycles
L3 Level	
FwdGetS Bound	iewExecuteStallL2FwdGetSPendingCycles / numCycles
FwdGetX Bound	iewExecuteStallL2FwdGetXPendingCycles / numCycles
FwdInv Bound	iewExecuteStallL2FwdInvPendingCycles / numCycles
Others Bound	iewExecuteStallL2OtherPendingCycles / numCycles

sharers than the actual ones, so invalidations will take more time to be sent and processed. For this new level of analysis, which we call the *L3 Level*, new stats are implemented to track the corresponding events. When a new GetX or Upgrade request is received, a FwdInv flag is set until all invalidations have been sent and replied to the requester.

To tag an L2 miss as a cache coherence-related miss it has to be solved by another L2. That is, L2 *a* asks L3 for a data block. L2 *b* holds this data with exclusive access, so L3 forwards this petition to L2 *b*. This miss is classified as a coherence miss because another L2 has exclusive access to the data block, and to maintain coherence, L2 *b* is asked for these data in case it has modified it. When the L3 forwards the request to the corresponding L2, it tags it as FwdGetS or FwdGetX, depending on the original request. If no forwarding or in-flight invalidations occur, the miss is classified as "Other". As with the base Top-Down metrics, the counters needed for these new two levels of analysis could be implemented adding the corresponding counters to the CPU PMU to account for these new events. These PMU counters can be implemented in different CPUs and architectures, so the Top-Down method and its metrics are

Table 2 Statistics of gem5 used in the Top-Down model implementation.

simTicks	Number of simulator ticks the entire simulation took
cpu.numCycles	Number of cycles this cpu has executed
commitWidht	Number of instructions that are commited per cycle
commitedOps	Number of Ops simulated
instsIssued	Number of instructions issued
squashCycles	Number of cycles Issue/Execute/Writeback (IEW) is squashing
uopsNotDeliveredBlock	Instructions not delivered to rename from decode when status is blocked
uopsNotDeliveredRun	Instructions not delivered to rename from decode when status is running or unblocking
uopsNotDeliveredSquash	Instructions not delivered to rename from decode when status is squashing
iewExecuteStallCycles	Number of cycles execute is stalled
iewExecuteGE1	Number of cycles execute executed at least 1 uops
iewExecuteGE2	Number of cycles execute executed at least 2 uops
iewExecuteStallAnyPendingCycles	Number of cycles execute is stalled and there is at least one Cache miss pending
renameBoundOnStores	Number of cycles where the Store Buffer was full and no outstanding load
iewExecuteStallL1PendingCycles	Number of cycles execute is stalled and there is at least one L1 miss pending
iewExecuteStallL2PendingCycles	Number of cycles execute is stalled and there is at least one L2 miss pending
iewExecuteStallL3PendingCycles	Number of cycles execute is stalled and there is at least one L3 miss pending
iewExecuteStallL2FwdGetSPendingCycles	Number of cycles execute is stalled and there is at least one L2 forwarded GetS miss pending
iewExecuteStallL2FwdGetXPendingCycles	Number of cycles execute is stalled and there is at least one L2 forwarded GetX miss pending
iewExecuteStallL2FwdInvPendingCycles	Number of cycles execute is stalled, there is at least one L2 miss pending and it is collecting the forwarded invalidations
iewExecuteStallL2FwdOtherPendingCycles	Number of cycles execute is stalled and there is at least one L2 miss pending different from a forwarded GetS or forwarded GetX, and it is not collecting forwarded invalidations

hardware and architecture independent and could be implemented in different CPU models. The Top-Down method just depends on the PMU implemented in the CPU, that must be capable of counting the right events.

In addition to these two new levels of analysis, we also adapted Top-Down base performance counters to properly study the L3 Bound. The Top-Down model follows a core perspective. Each core has its Core and Memory Bounds, L1, L2 Bounds, etc. On the Memory Level, it is easy to track the L1 and L2 Bounds since each core/node has its own private L1 and L2 caches. However, when it comes to the L3 Bound, it is more complex since each core needs to track its requests for the whole L3 cache, not only the L3 tile attached to its node. In our case, all nodes share the L3, so each core

Table 3 Cache configurations applied in the simulator.

L1D	48 KiB, private, 12-ways, 5 cycles
L1I	48 KiB, private, 12-ways, 2 cycles
L2	1 MiB, private, 16-ways, 20 cycles
L3	4 MiB, shared, 16-ways, 30-75 cycles

needs to track how many misses caused by itself are pending on any L3 tile. Finally, Table 1 shows all the stats added to gem5 for our Extended Top-Down Model, while Table 2 explains each of the gem5 statistics used for its implementation.

3.2 Memory hierarchy configuration

After configuring the microarchitectural parameters of the core model, we continued to configure the cache hierarchy. The size, associativity, and access latency configuration (round trip) for the different cache levels are shown in Table 3. As stated in the table, the L3 (LLC) is physically distributed with 4 MiB for each core but is logically shared, so each core sees a global, shared L3 of N $\times$ 4 MiB, where N is the number of cores of the system.

In the baseline configuration of gem5, cache latencies are pretty optimistic compared to a real system. The L1D cache latency was 2 cycles, while the L2 cache latency was 4 cycles. We wanted to apply the latencies shown in Table 3. These are not the exact latencies we see in the Golden Cove, but they are very similar, and, more importantly, they are more representative latencies than the ones provided in the base configuration of the simulator.

However, simply increasing latencies has a very detrimental effect on performance for some applications (up to 2$\times$), which we would not see in a real system. We will now explain why this issue occurs and propose several fixes to configure cache latencies without incurring unrealistic performance losses.

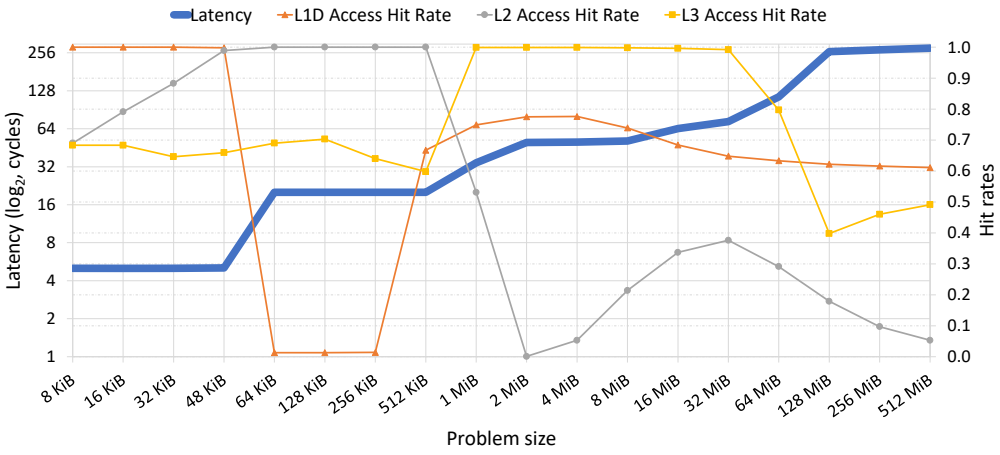
3.2.1 Instruction cache latency

To increase cache latencies without harming too much performance, we will serve different latencies depending on the type of access we are solving. We focus first on the instruction cache. Since gem5 does not model μop cache, we need a fast instruction cache. Instruction fetches will have 1 cycle enqueue latency + 1 cycle response latency. This is an oversimplification of current real-hardware front-ends. Significant efforts from the gem5 community are being made to provide a decoupled front-end for gem5 [2].

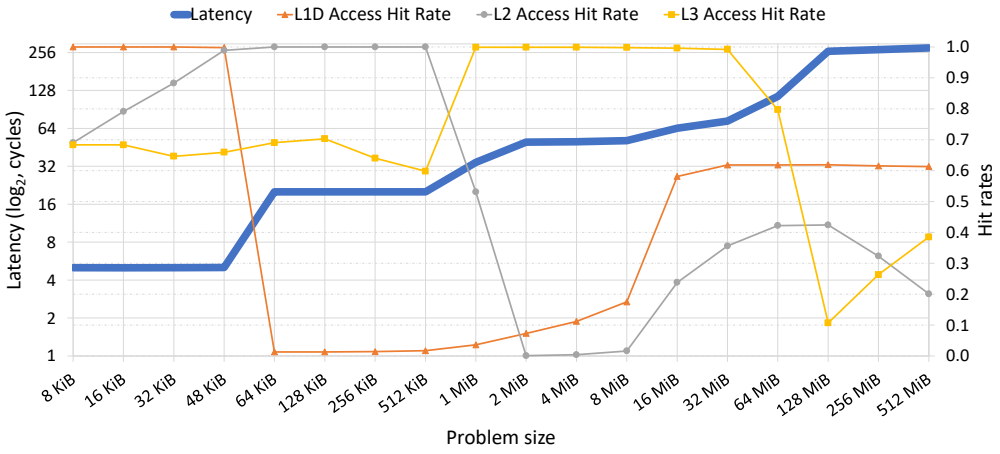
3.2.2 Data cache latency

x86 processors implement the Total Store Order (TSO) consistency model. In TSO, stores must be made visible to other cores in order, limiting the level of memory-level paralelism for stores. When a store commits, it enters a Store Buffer (SB), where it remains while waiting to be stored in the L1D cache in program order. If a store misses

[2] e.g., https://github.com/dhschall/gem5-fdp



(a)



(b)

Fig. 4 MemoryLatency benchmark results for increasing problem sizes, (a) 64 entries TLB and (b) 2048 entries TLB.

in the L1D cache, the SB can commonly hide this miss latency. However, if the SB fills, the pipeline stalls. To reduce SB-induced stalls, we need to reduce the waiting time of the store at the head of the SB. To this end, we propose two patches that extend the core model with contemporary mechanisms implemented in modern processors.

The first one is to pipeline the stores. By pipelining the stores, if a store hits in the L1D, the next store will overlap its execution and show essentially one cycle latency. If the store misses, the next one will expose its full latency (4 + 1 cycles). Load accesses will still have 4 + 1 cycle access latency, however note that they do not need to be pipelined, since TSO does not prevent multiple loads to be performed in parallel. This patch reduces store latency in case of a previous hit, emptying the SB faster, and thus reducing the possibility of it filling up.

The second one is Prefetch on Commit. While the order of the stores cannot be altered in TSO to exploit memory-level parallelism, write permissions for their respective cache lines (blocks) can be requested in parallel. Consequently, we extend the core model in gem5 by implementing this feature. To do so, we modify the Load Store Queue (LSQ), and every time a store commits, we generate a prefetch request (for write permissions) on the data L1 cache. We check if the store has a request and if it has, we launch the prefetch of that request. Thanks to this feature, we can save time in store latency by prefetching the needed data when another store commits, trying to avoid store misses in the L1D, and therefore having lower store latencies, as we pipelined the stores with the previous modifications, and reducing the SB overload.

With these two patches, we can increase cache latencies while still maintaining realistic performance for stores. However, the increased latency of $4 + 1$ cycles for the loads produced a $2\times$ slowdown on some applications, while others did not experience any slowdown. To check if these new latencies are working properly, we use a microbenchmark to measure access latencies to the different levels of the memory hierarchy. More specifically, we use the Chips and Cheese MemoryLatency microbenchmark [25].

This microbenchmark measures the latency of random accesses to arrays of increasing size, trying to avoid any spatial locality. Finally, it reports the mean latency of all the performed accesses. If the problem size fits on the L1, the measured value will be equal to the L1 latency. If it is bigger than L1, but fits L2, the measured value will be equal to the L2 latency, etc.

We execute the MemoryLatency benchmark to measure cache latencies, with increasing problem sizes from 8 KiB up to 512 MiB, with a 16-core system configuration. With this system configuration, we get the results shown in Fig. 4a. In these graphs, access latency is reported as the main result. We also plot hit rates for the different cache levels. For problem sizes ranging between 8 KiB to 48 KiB, we experience the L1D access latency. From 64 KiB to 1 MiB, L1 + L2 access latency (access to L1, misses, and then access to L2). From 2 MiB to 64 MiB, the L1 + L2 + L3 access latency. Above 128 MiB, only the memory access latency.

In this graph, we see that, when the problem size fits in each of the levels, its hit rate is close to 100%. This is the expected behavior since, as long as the problem fits in that cache, all the requests made will result in cache hits. When the problem size increases and no longer fits in that cache, its hit rate drops, since most of the requests (almost all of them) will miss that cache because of the random nature of the accesses made.

Although access latencies reported are as expected, we observe a sudden increase in the hit rate for the L1, and later on, for the L2 and L3 once we reach a problem size of 256 KiB. After exhaustive testing, we found out that this increase in the hit rates was caused by an inadequate TLB configuration. Nowadays, many architectures implement a 2-level TLB. However, gem5 assumes by default a small single-level TLB. In these tests, its size was 64 entries. With this limited, single-level TLB, when we reach this problem size, we start to have several TLB misses. To solve these TLB misses, the page-table walker starts to access the caches, searching for the translation that missed the TLB. That caused a lot of extra cache accesses, most of which were cache hits, which artificially increased the hit rates of all the cache levels. So, those

extra hits were not related to the MemoryLatency benchmark accesses but to the TLB misses.

This reduced TLB size also explains the slowdown in some applications. When the page-table walker accesses the caches in TLB misses, it pays the 5 cycles access latency (compared to the 2 cycles in the baseline), slowing down execution. If we had fewer TLB misses, we would reduce the number of accesses to the caches to search for the translations, which would reduce the unexpected high hit rates and the performance penalty observed in several applications.

To corroborate our hypothesis, we increased the size of the TLB to compensate for the lack of a second TLB level. After some testing with different TLB sizes, we found that a TLB size of 2048 entries provides similar results for this microbenchmark compared with a real system, and is similar to the size of a second TLB level of a current processor, which should be sufficient to achieve similar performance.

Now, in Fig. 4b we see the results with this increased TLB. We observe that the sudden hit rate increase is delayed to a problem size between 1 MiB and 16 MiB, especially when we go from 8 MiB to 16 MiB. These results prove that the TLB size was indeed the problem. Additionally, we tested this microbenchmark on real hardware and found similar results with this increase in L1 hit rate at high problem sizes. Thus, we can identify this behavior as "normal". From now on, we will use a TLB of 2048 entries, but ideally, we should implement a two-level TLB.

subsubsectionTop-Down analysis of the improved configuration

Now, with this improved configuration, we are going to compare the baseline CPU configuration against our enhanced configuration for the simulator using the Top-Down model. In Fig. 5 we observe the results for the five levels of the analysis, normalized to the baseline configuration. In the first graph, we observe, for each benchmark, how the execution time is distributed between useful execution time, when the core is actually executing instructions, and the time when the core is idle, waiting for some resource or dependency. We can see that, on average, we spend more time executing with the enhanced configuration of the simulator, with a considerable increase in raytrace. Other benchmarks like fmm, radix and water_nsquared experience a speedup with our configuration and, therefore, have lower Executing categories. This is expected, since all these changes and configurations aim to model a more realistic system, but this does not imply that its performance must be better than the currently modeled system (our configuration has realistic timings for caches, memory and network).

On the next graph, the Top Level focuses on the Executing Bound from Level 0, and classifies each executed slot between Retiring, Bad Speculation, Frontend Bound and Backend Bound. Here we observe that the main increase in execution time is due to the Backend Bound, so we expand this category on the Backend Level in the third graph. In this graph, the executed cycles of this category are distributed between Core and Memory Bound. We can observe that the higher Backend Bound on the enhanced configuration is, on average, mostly due to the Memory Bound. Some benchmarks like barnes and radiosity do not increase the Memory Bound, but the Core Bound. In the forth graph, we put the focus on this Memory Bound at the Memory Level. On average, we observe an increase in the L3 Bound. With raytrace this is also the case, meaning that this increase in the execution time is due to the higher L3 access latency

applied in our configuration. Others like cholesky and fft also experience an increase in the External Memory Bound.

Finally, on our newly added L3 Level in the last graph, we classify the L3 misses cycles depending on whether they are coherence or non-coherence related. Since for now we are not modifying the way coherence is represented in the system, in this level most of the extra cycles are attributed to the Other Bound category due to the higher access latencies. Later we will appreciate the potential of this new level when we study how our modifications to the directory structure can have an impact on performance.

3.3 New directory designs

Although gem5 SLICC base coherence protocols use a directory to maintain coherence, it is implemented as an embedded directory by extending the LLC entries to hold the directory information of each cache line. Fig. 6 shows an example of the structure of an embedded directory. One of the main drawbacks of this design is that this directory information is unused on every cache entry with 0 sharers (cache blocks present in the shared LLC but not present in any private cache) but still consumes resources. Moreover, with this design, the LLC has to be inclusive with the upper-level caches because every cache block that is in a private cache from the upper level needs to have an entry on the directory and, therefore, it also needs an entry on the LLC since the directory is embedded on the LLC entries. Thus, we cannot modify the inclusiveness of the LLC and are forced to use an inclusive LLC.

3.3.1 Sparse directory support

To solve these problems, we implement a sparse directory cache to detach the directory from the LLC. With a separate cache for the directory, we can store the directory entries for those cache blocks stored in, at least, one private cache and eliminate the directory information for those cache blocks that do not need it. This new design requires the directory cache to be inclusive with the upper private caches, but now the LLC can be exclusive or non-inclusive. Every cache block present in a private cache needs to have its corresponding entry in the directory cache. This inclusiveness between the directory cache and the private caches allows us to soften the inclusiveness of the LLC. Now, when a cache block is present in a private cache, it does not necessarily have to be present on the LLC, but only in the directory cache. Sometimes, there will be a copy of such block in the LLC, but it will not be strictly necessary.

For our new system design and the cache coherence protocol, we removed the directory information from the LLC entries and stored it in a separate structure, creating a standalone directory cache. This directory cache will be attached to the same controller as the LLC. In the directory cache entries, we will store the directory information previously stored in the cache entries. In Fig. 7 we can see the structure of this proposed implementation for the sparse directory.

The main limitation of this standalone directory cache is the duplication of tags for every directory cache entry. Usually, the additional memory required for duplicated tags is higher than the memory required to store the actual directory information. However, we found that, for our configuration (which is based on coverages of 50%,

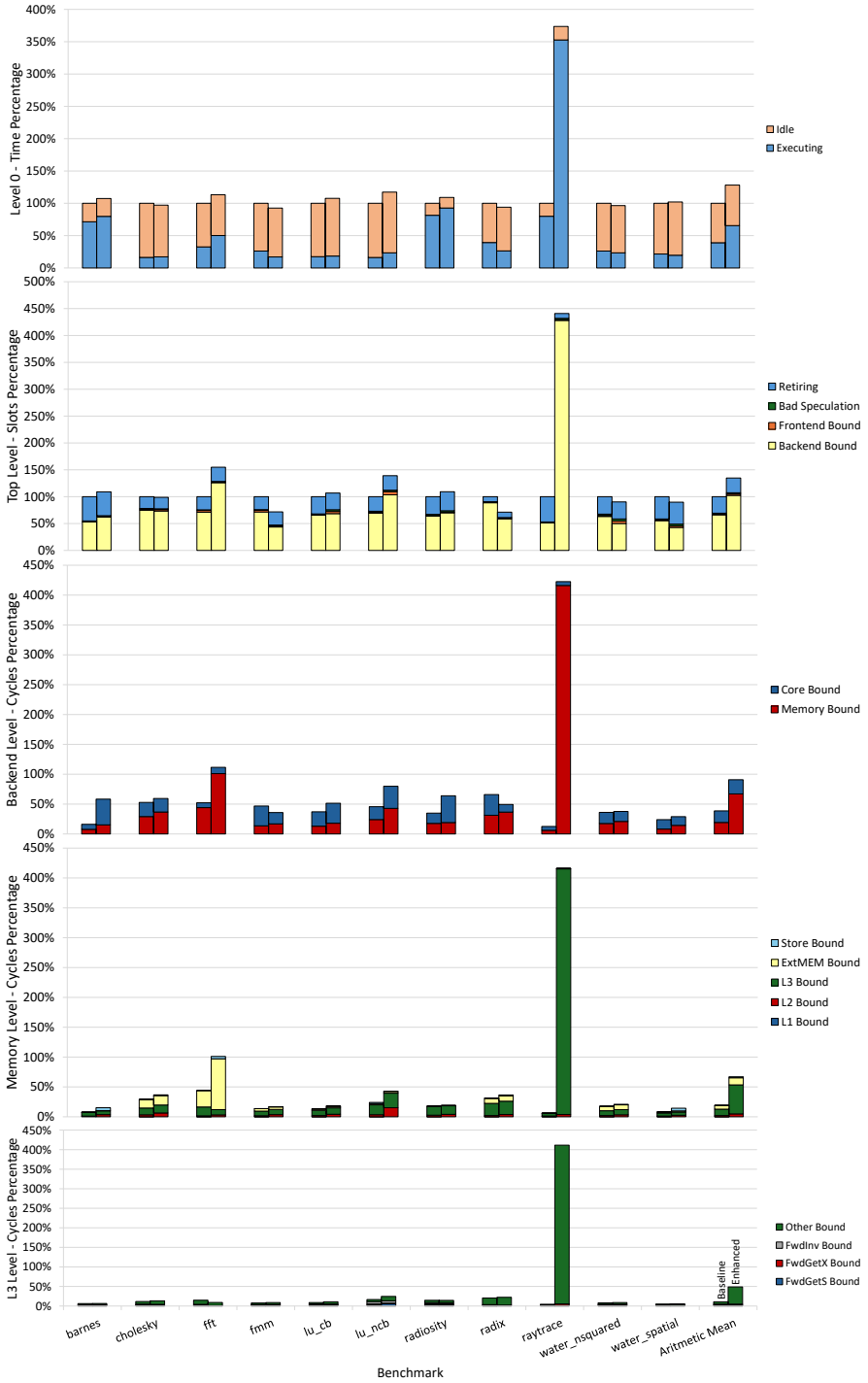


Fig. 5 Top-Down results for the baseline and the enhanced simulator configuration.

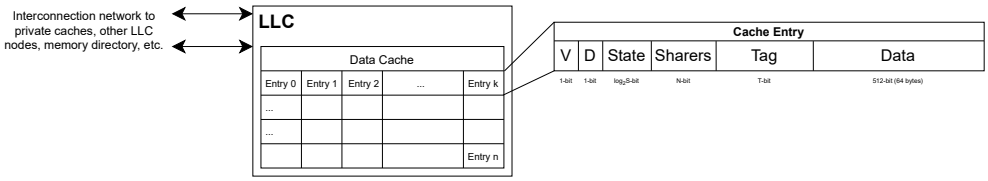


Fig. 6 Design and distribution of each LLC node in the original system, with N cores and S possible states for a cache block in the LLC. The data cache is k-way associative and we have T bits for the entry tags.

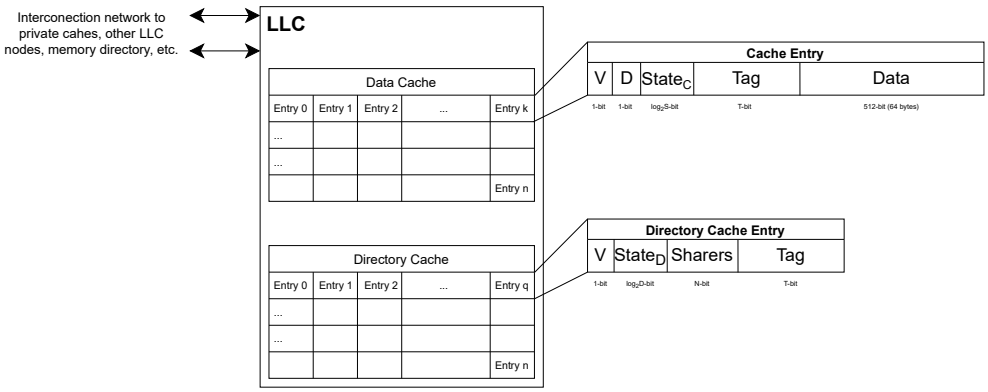


Fig. 7 Design and distribution of each LLC node in the proposed system with a standalone directory cache, N cores, S possible states for a cache block in the LLC cache, and D possible states for an entry in the directory cache. The data cache is k-way associative, the directory cache is q-way associative, and we need T bits for the tag of every entry.

100%, and 200%), the size of the directory cache will be 39.84%, 79.68%, and 159.37% of the size of the original embedded directory design, respectively, including the duplicated tags. We also have to modify the coherence protocol to adapt its behavior to this new sparse directory structure and modify the inclusiveness of the LLC.

3.3.2 Limited pointers codification

One way to save space on every directory entry is by encoding sharer information using limited pointers. With this codification, we can reduce the number of bits needed for this sharing information in exchange for not being able to encode this information in a complete and exact manner when the number of sharers exceeds the number of pointers. If the number of pointers is lower than the number of nodes in the system, there will be some cases in which the number of sharers will be higher than the number of pointers used. In that case, we will consider broadcasting coherence messages as the overflow policy. When there are more sharers than pointers, an overflow bit is activated and all nodes in the system are considered as sharers. This broadcast representation

is maintained until the directory can ensure that there are no sharers for that cache block (invalidated or replaced) or just a single sharer (upgrade or write request).

We made the appropriate modifications to the coherence protocol to work with this new sharing representation. An interesting case was the PutX requests made by an L1 when considering that the directory entries have no pointer (the so-called Dir_0B representation [26]). Some PutX requests need to be discarded because the requester has been invalidated before its PutX request is solved, maybe because of a write request from another node, so modified data have been forwarded and, therefore, there is no need to write it on the LLC. In this case, the LLC controller was originally able to detect this situation by searching the current sharers and verifying that the requester is no longer a sharer of the block.

However, with 0 pointers, this sharing information is inexistent, and the LLC controller cannot detect this situation. To solve this, we modify the way PutX requests are handled. Instead of sending the PutX request to the LLC, the L1 sends a PutX_Probe to ask for permission to make a PutX request. LLC answers this PutX_Probe when it is ready to solve it, and at this point, it is the requester L1 who decides if the PutX request is still valid or not. If it was invalidated while waiting for the LLC response to that PutX_Probe, the PutX request is no longer valid and the LLC will be sent a Nack message.

3.4 Interconnection network model

One of the key areas in our simulated system is the interconnection network. In gem5, there are two network models, Simple and Garnet, which trade off detailed modeling versus simulation speed, respectively. The Simple network models hop-by-hop network traversal but abstracts out detailed modeling within the switches regarding contention, something that reveals critical as increases the traffic on interconnection network (something that occurs when aggressive sharer representations, such as Dir_0B, are used). On the other hand, gem5 also offers HeteroGarnet [27], which builds upon the original Garnet [28] model and improves it by enabling accurate simulation of emerging interconnect systems.

Garnet provides a cycle-accurate micro-architectural implementation of an on-chip network router. It models all network elements with a higher level of detail, which allows for more accurate results and better modeling of the contention between all of the elements that make up the interconnection network. This is significantly relevant to research studies on cache coherence, as we will see.

With our imprecise representation designs for directory-sharing information, we are introducing a significant overhead in the interconnection network, as we need considerably more messages to maintain coherence. These extra messages can affect the overall system performance, saturating the interconnection network and slowing execution. To observe the effect of the network on performance, we need to model contention properly. For this reason, we decided to use the Garnet network model in our studies. If we had used the Simple network, since it does not model contention correctly, we could have obtained erroneous results and reached incorrect conclusions.

The Simple network model, however, adds statistics to classify bytes sent through the interconnection network among different types of messages. This includes:

- Request_Control: GetS, GetX, Upgrade, Forwards and Invalidation messages.
- Wirteback_Control: PutX messages without data.
- Response_Data: Data and DataExclusive messages.
- Response_No_Data: Ack and Unblock messages.
- Writeback_Data: PutX messages with data.

The Garnet network model lacks this information, which we also had to implement. With the Garnet network model, we ensure that realistic interconnection network, that takes into account the contention present when the number of messages travelling across the network is high, is modelled.

4 Methodology

To evaluate all the modifications proposed in this work, Table 4 summarizes the most relevant parameters used to configure gem5. We use gem5 version 21.1. We model a tiled manycore architecture with Golden Cove-like CPU cores. Structure sizes, pipeline widths, and ALU latencies are based on several sources, including Chips And Chips Golden Cove article [29], and Agner Fog's work [30]. For the cache hierarchy, we model three levels: L1 and L2 are private to each core, and L3 is shared and distributed among the cores (one L3 tile per core). It uses a MESI cache coherence protocol, adapted in each case to the different directory designs considered in this work. The latency of L2 is for round-trip.

4.1 Evaluated directory configurations

We evaluate three different directory designs. The first design is the base directory implementation, with an embedded directory in L3 and a bit-vector to encode sharing information. The second design is the limited pointer implementation, using an embedded directory in L3 and 0 or 1 pointers to encode sharing information. The third design is the sparse directory implementation, which models a standalone sparse directory cache distributed in one bank per node and a bit-vector to encode sharing information. Depending on the coverage we want to obtain, the size of each tile of this directory cache will vary. For coverages of 50%, 100%, and 200%, the size of each tile of the directory cache is 8192 entries, 16-ways, 16384 entries, 16-ways, and 32768 entries, 16-ways, respectively.

4.2 Applications

The benchmark suite used in this work is *Splash-3* [31]. The Splash-3 benchmark suite is an improved version of the Splash-2 [32] suite, which consists of a mixture of full applications and computational kernels representing a variety of computations in scientific, engineering, and graphical computing. To keep simulation times tractable, we use the *simsmall* problem size for all benchmarks except for FFT and Ocean-CP, where we had to increase the problem size to *simmedium* to improve their scalability. Table 5 shows a summary of the input parameters that each benchmark receives.

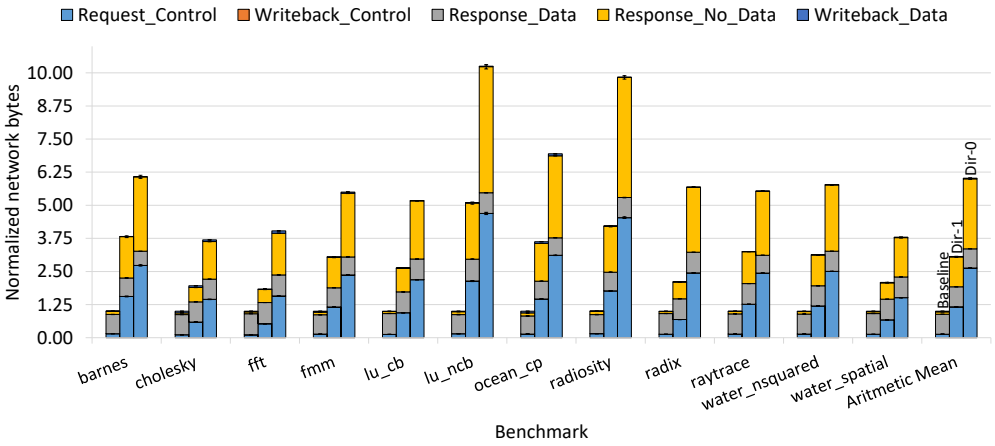
Table 4 System configuration parameters used in gem5.

Core Settings	
Architecture	x86_64
Processor frequency	2 GHz
Number of cores	64
TLB size	2048 entries
Branch predictor	64 KiB L-TAGE + ITTAGE
LQEntries	192
SQEntries	114
LFST/SSITSize	1024
numPhysInst/Float Regs	332
numIQEntries	208
numROBEntries	512
Fetch Width	8
Decode / Rename Width	6
Dispatch / Issue Width	12
Commit Width	8
Memory Settings	
L1 data cache	Private, 48 KiB, 12-ways, 5 cycles latency
L1 instructions cache	Private, 48 KiB, 12-ways, 2 cycles latency
L2 cache	Private, 1 MiB, 16 ways, 20 cycles latency
L3 cache	Shared, 4 MiB, 16-ways per tile. Total size of 256 MiB, 30-75 cycles latency
Memory	3 GiB, SimpleMemory, 160 cycles latency
Network Settings	
Network model	Garnet
Network topology	Mesh_XY
Mesh dimensions	8×8

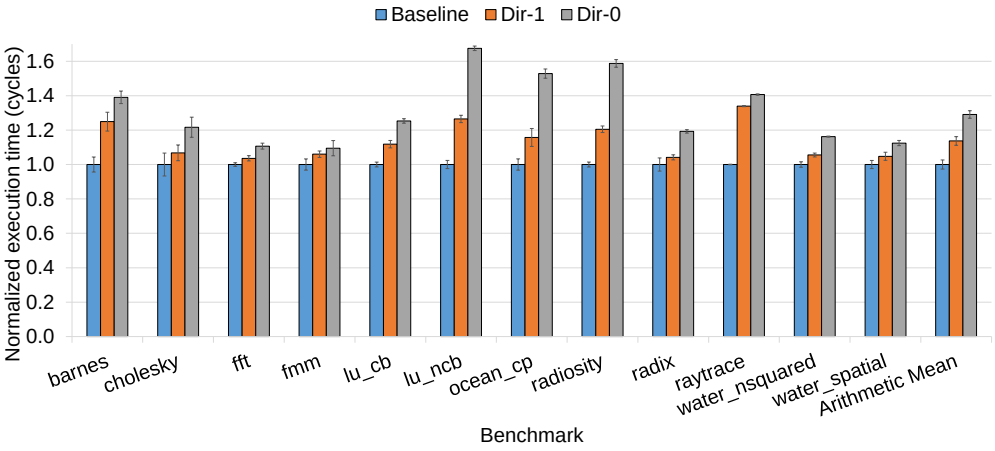
Table 5 Input parameters of benchmarks used.

Barnes	16K particles
Cholesky	Size of matrix: 13992×13992, Non-zero value elements: 316740
FFT	2^{20} complex points in total
FMM	16K particles
LU-CB	Size of matrix: 512×512, block size: 16
LU-NCB	Size of matrix: 512×512, block size: 16
Ocean-CP	Grid size: 514×514
Radiosity	Refinement BF=$1.5e^{-1}$
Radix	1M keys, Radix=1K
Raytrace	Car scene
Water-Nsquared	512 molecules
Water-Spatial	512 molecules

Since these benchmarks are parallel applications, we bind each application thread to a different core to improve workload distribution and results.



(a)



(b)

Fig. 8 Results for the imprecise sharer representations (Dir-1 and Dir-0).

5 Results

This section evaluates the impact of the proposed directory organizations on performance. First, we characterize the impact that imprecise sharing codifications have on network traffic and analyze how the Simple network model overlooks the significant impact this has on performance. Then, we evaluate the sparse directory model with different coverage levels. For the evaluations, the baseline system includes all the improvements and modifications proposed in this work and an embedded directory with a bit-vector sharing codification.

5.1 Imprecise sharing codification

For this set of experiments, we replace the bit-vector sharing code used in the baseline system with two implementations of the limited pointers codification presented

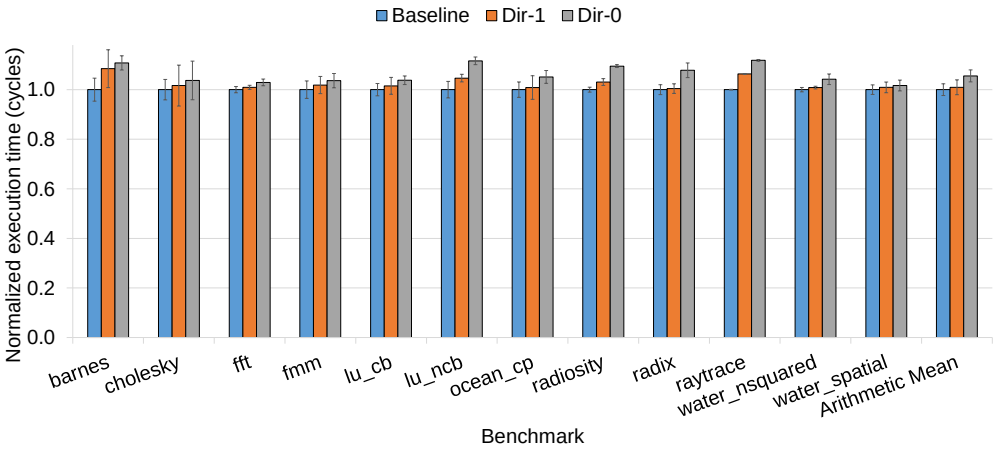


Fig. 9 Execution time results for Dir-1 and Dir-0 when the Simple network model is employed.

in Section 3.3 with 0 (Dir-0) and 1 (Dir-1) pointers, respectively. Fig. 8a shows the total number of bytes sent through the interconnection network, classified according to the different message types. All results have been normalized to the baseline. We observe that, as expected, these imprecise sharer codifications have a noticeable impact on network traffic. In particular, the amount of traffic due to Request_Control and Response_No_Data messages drastically increases as a consequence of having to broadcast coherence messages (e.g. invalidation messages) and receive their corresponding acknowledgments when the overflow bit is set.

We observe an average increase in network traffic of 3× and 6× for Dir-1 and Dir-0 respectively, with *lu_ncb* reaching growth of 5.1× and 10.2×. Fig. 8b shows the normalized execution time for these configurations. We can see how the increase in network traffic finally translates into significant degradations of the execution times. Higher levels of network traffic lead to higher contention in the interconnection network and, consequently, longer cache miss latencies, ultimately resulting in lower performance (average performance degradations of 14% and 29% for Dir-1 and Dir-0, respectively).

Fig. 9 shows execution time results with an identical configuration except for the interconnection network model. In this case, we use the Simple network model. Here, we demonstrate how important a correct system configuration can be. With the Simple network, we observe an almost negligible performance degradation (less than 1% for Dir-1 and 5% for Dir-0) despite the huge increase in traffic levels, confirming that the Simple network does not model contention as accurately as the Garnet model, and therefore, does not result appropriate for estimating the effect on performance of optimizations that can result into increased traffic levels.

Going back to the Garnet model, we can also study our newly added L3 Level of the extended Top-Down methodology presented in Section 3.1. Fig. 10 shows the results of the Top-Down method with these imprecise codifications, from Level 0 to L3 Level. In the first four graphs we observe that, with Dir-1 and specially with Dir-0, i) we spend more time executing (shown in the Level 0 graph); ii) this comes from

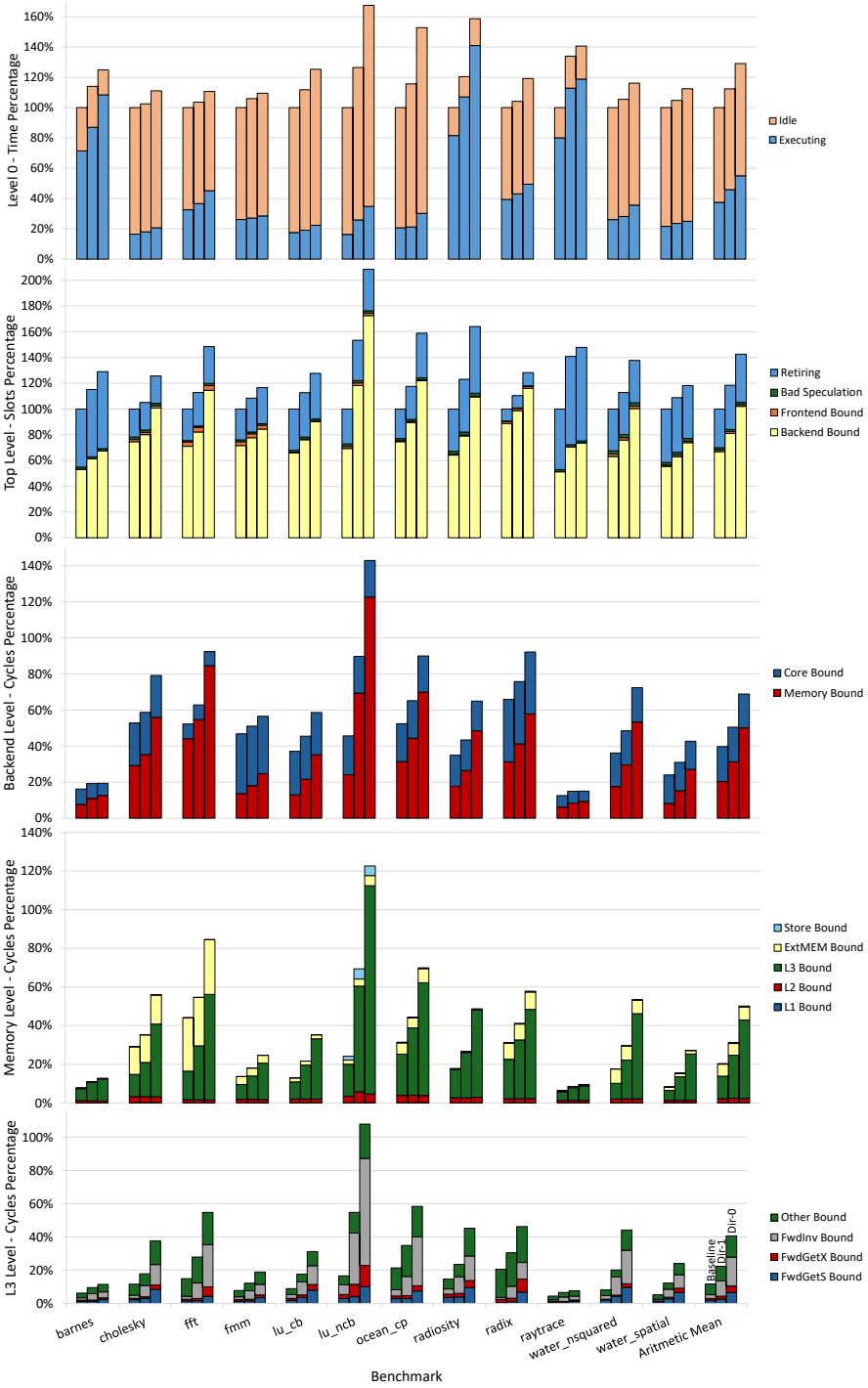


Fig. 10 Top-Down results for the imprecise sharing codifications. It shows how the L3 Level of the extended Top-Down methodology proposed in this work captures the effects of Dir-0 and Dir-1.

a significant increase in the Backend Bound (shown in the Top Level graph); iii) in turn, this is a consequence of increasing the Memory Bound (shown in the Backend Level graph), and iv) the sources of these increments come from the L3 Bound (shown in Memory Level graph). At this precise point, our new L3 Level comes into play to help us understand what is exactly happening at the L3 level (shown in the L3 Level graph).

Here, we classify the percentage of cycles when there is a pending L2 miss, depending on the miss type: a forwarded GetS petition, a forwarded GetX petition, forwarded invalidations due to a write petition, or other misses. We observe how the percentage of cycles spent on each category increases as we lose precision in the representation of the sharers. Among all the categories, the FwdInv Bound category is the most affected. As explained, Dir-1, and especially, Dir-0 typically consider many more nodes as sharers, so they require more invalidation messages on write requests when the block is shared and there is more than 1 sharer (even a single sharer with the Dir-0 codification). This causes more stalls waiting for these extra invalidation messages to be processed, which increases cache miss latency and ultimately hurts performance.

5.2 Sparse directory model

We now compare the baseline system with an embedded directory design against a sparse directory design with coverages of 50%, 100%, and 200%. First, Fig. 11a shows the effect on network traffic. Here, we can see how, as we decrease the directory cache size (and coverage), the amount of traffic on the interconnection network gets increased. As the size of the directory cache decreases, we have more replacements, and consequently, more invalidation messages, which makes the Request_Control category grow. These invalidation messages, in turn, cause more data writebacks of invalidated and modified data, so the Writeback_Data category also grows. Furthermore, the Response_Data category also increases since we have modified the inclusiveness between L2 and L3, making more data move between these two levels when necessary.

Despite these reduced directory sizes, only a few benchmarks experience a noticeable increment in the amount of network traffic. On average, increases for coverages of 100% and 200% are negligible, while 50% coverage only increases network traffic by 1.12×. Some benchmarks, like fft or Ocean-CP, have more noticeable increases for 50% coverage, with Ocean-CP reaching 1.97×. Looking at execution times in Fig. 11b, we observe how these extra bytes also translate into higher execution times. However, they are not as considerable as with the imprecise representation. On average, a sparse directory with 50% coverage only increases execution times by 2% while saving more than 60% of the space used for sharing information with an embedded directory.

Again, in Fig. 12 we observe the results for the sparse directory when we use the Simple network model. As we have seen previously in Fig. 9, using the Simple network model can lead to misleading conclusions, since it does not reflect the performance degradations we observe in Fig. 11b caused by the increments in the network traffic shown in Fig. 11a. In these results, we observe negligible impact on performance (in some benchmarks like fft the performance of the sparse directory even improves the baseline embedded directory), and they do not reflect the increased network traffic

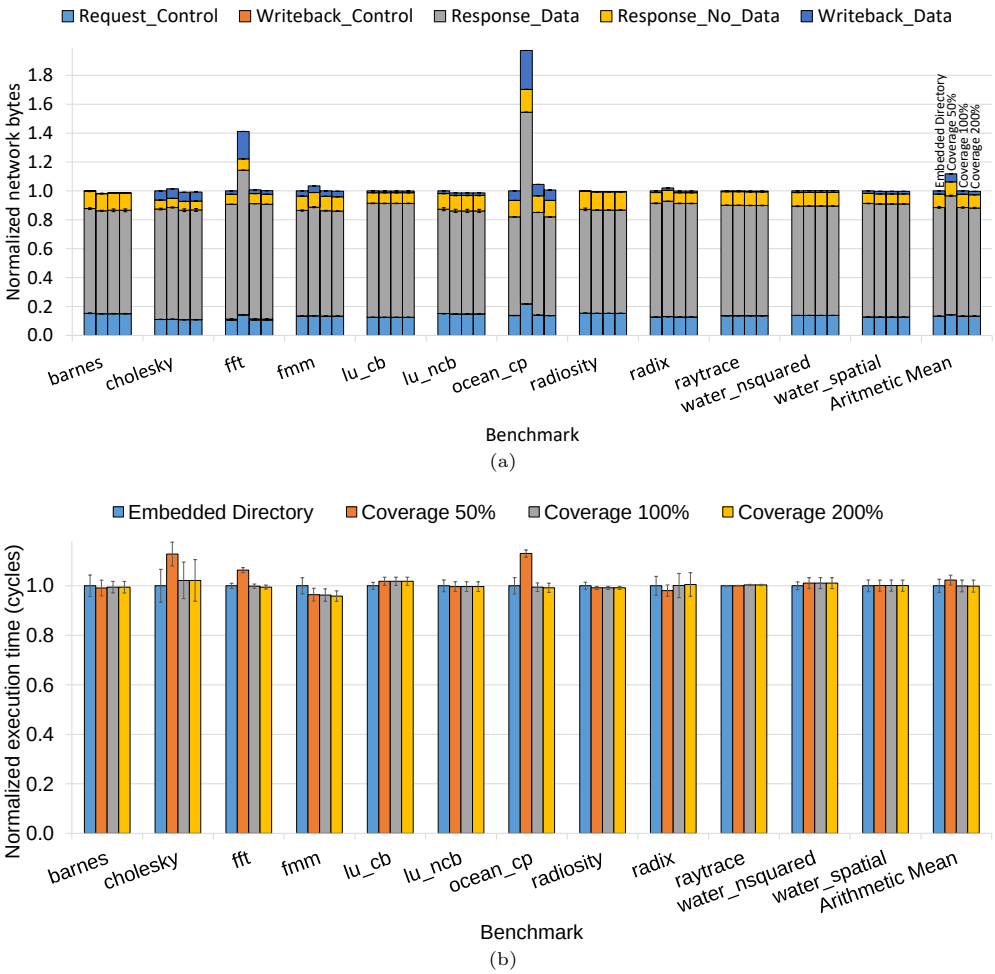


Fig. 11 Results for sparse directory.

in fft and specially Ocean-CP, confirming that this network model does not model contention accurately enough.

6 Conclusion

Gem5 and similar simulators are complex, extensive tools that cannot be used out of the box. Even small misconfigurations in key aspects of a multicore setup—core microarchitecture, memory hierarchy, or the interconnection network—can produce unrealistic results and flawed conclusions. In this work, we focus on simulating coherence activity in multicores.

In the case of gem5, we perform a deep fine-tuning of the simulator, applying several bug fixes and adding new features to model contemporary multicore architectures. We also implement and extend Intel's Top-Down methodology to characterize coherence

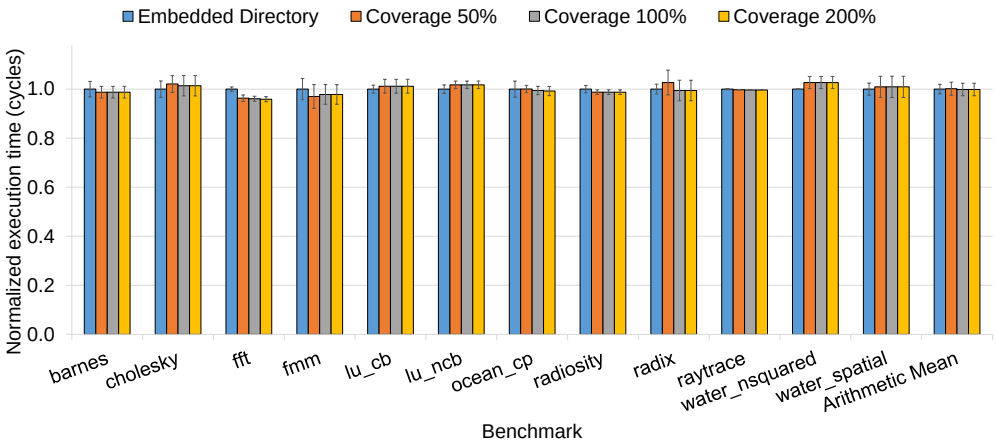


Fig. 12 Execution time results for sparse directory when the Simple network model is employed.

activity in a multicore setup. Finally, we also provide gem5 with the possibility of configuring a variety of sparse directory architectures and sharer codifications. The result of this work is a version of gem5 ready for conducting research in the field of cache-coherent multicores.

Acknowledgments

Work carried out in the context of the project PID2022-136315OB-I00 funded by MCIN/AEI/10.13039/501100011033/ and "ERDF A way of making Europe", EU. Also, in the context of the project TED2021-130233B- C33, funded by MCIN/AEI/10.13039/501100011033 and by the "European Union NextGenerationEU/PRTR". Joaquín Ferrer has been funded by grant 22723/FPI/24 from Fundación Séneca, Agencia de Ciencia y Tecnología de la Región de Murcia.

References

[1] Lowe-Power, J., Ahmad, A.M., Akram, A., Alian, M., Amslinger, R., Andreozzi, M., Armejach, A., Asmussen, N., Beckmann, B., Bharadwaj, S., Black, G., Bloom, G., Bruce, B.R., Carvalho, D.R., Castrillon, J., Chen, L., Derumigny, N., Diestelhorst, S., Elsasser, W., Escuin, C., Fariborz, M., Farmahini-Farahani, A., Fotouhi, P., Gambord, R., Gandhi, J., Gope, D., Grass, T., Gutierrez, A., Hanindhito, B., Hansson, A., Haria, S., Harris, A., Hayes, T., Herrera, A., Horsnell, M., Jafri, S.A.R., Jagtap, R., Jang, H., Jeyapaul, R., Jones, T.M., Jung, M., Kannoth, S., Khaleghzadeh, H., Kodama, Y., Krishna, T., Marinelli, T., Menard, C., Mondelli, A., Moreto, M., Mück, T., Naji, O., Nathella, K., Nguyen, H., Nikoleris, N., Olson, L.E., Orr, M., Pham, B., Prieto, P., Reddy, T., Roelke, A., Samani, M., Sandberg, A., Setoain, J., Shingarov, B., Sinclair, M.D., Ta, T., Thakur, R., Travaglini, G., Upton, M., Vaish, N., Vougioukas, I., Wang, W., Wang, Z., Wehn,

N., Weis, C., Wood, D.A., Yoon, H., F. Zulian: The gem5 Simulator: Version 20.0+ (2020). https://arxiv.org/abs/2007.03152

[2] Yasin, A.: A top-down method for performance analysis and counters architecture. In: 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 35–44 (2014). https://doi.org/10.1109/ISPASS.2014.6844459

[3] Hughes, C.J., Pai, V.S., Ranganathan, P., Adve, S.V.: Rsim: simulating shared-memory multiprocessors with ilp processors. Computer **35**(2), 40–49 (2002) https://doi.org/10.1109/2.982915

[4] Kaxiras, S., Young, C.: Coherence communication prediction in shared-memory multiprocessors. In: Proceedings Sixth International Symposium on High-Performance Computer Architecture. HPCA-6 (Cat. No.PR00550), pp. 156–167 (2000). https://doi.org/10.1109/HPCA.2000.824347

[5] Acacio, M.E., González, J., García, J.M., Duato, J.: A new scalable directory architecture for large-scale multiprocessors. In: 7th Int'l Symp. on High-Performance Computer Architecture (HPCA), pp. 97–106 (2001). https://doi.org/10.1109/HPCA.2001.903255

[6] Sanchez, D., Kozyrakis, C.: Zsim: fast and accurate microarchitectural simulation of thousand-core systems. SIGARCH Comput. Archit. News **41**(3), 475–486 (2013) https://doi.org/10.1145/2508148.2485963

[7] Sanchez, D., Kozyrakis, C.: SCD: A scalable coherence directory with flexible sharer set encoding. In: 18th Int'l Symp. on High-Performance Computer Architecture (HPCA), pp. 129–140 (2012). https://doi.org/10.1109/HPCA.2012.6168950

[8] Martin, M.M.K., Sorin, D.J., Beckmann, B.M., Marty, M.R., Xu, M., Alameldeen, A.R., Moore, K.E., Hill, M.D., Wood, D.A.: Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. ACM SIGARCH Computer Architecture News **33**(4), 92–99 (2005) https://doi.org/10.1145/1105734.1105747

[9] Magnusson, P.S., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Hogberg, J., Larsson, F., Moestedt, A., Werner, B.: Simics: A full system simulation platform. IEEE Computer **35**(2), 50–58 (2002) https://doi.org/10.1109/2.982916

[10] Martin, M.M.K., Hill, M.D., Wood, D.A.: Token coherence: Decoupling performance and correctness. In: 30th Int'l Symp. on Computer Architecture (ISCA), pp. 182–193 (2003). https://doi.org/10.1145/871656.859640

[11] Marty, M.R., Bingham, J.D., Hill, M.D., Hu, A.J., Martin, M.M.K., Wood, D.A.: Improving multiple-CMP systems using token coherence. In: 11th Int'l Symp. on High-Performance Computer Architecture (HPCA), pp. 328–339 (2005). https:

//doi.org/10.1109/HPCA.2005.17

[12] Fernández-Pascual, R., García, J.M., Acacio, M.E., Duato, J.: A low overhead fault tolerant coherence protocol for CMP architectures. In: 13th Int'l Symp. on High-Performance Computer Architecture (HPCA), pp. 157–168 (2007). https://doi.org/10.1109/HPCA.2007.346194

[13] Cuesta, B., Ros, A., Gómez, M.E., Robles, A., Duato, J.: Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks. In: 38th Int'l Symp. on Computer Architecture (ISCA), pp. 93–103 (2011)

[14] Binkert, N.L., Dreslinski, R.G., Hsu, L.R., Lim, K.T., Saidi, A.G., Reinhardt, S.K.: The m5 simulator: Modeling networked systems. IEEE Micro **26**(4), 52–60 (2006) https://doi.org/10.1109/MM.2006.82

[15] Carlson, T.E., Heirman, W., Eeckhout, L.: Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations. In: Conf. on Supercomputing (SC), pp. 52–15212 (2011). https://doi.org/10.1145/2063384.2063454

[16] Miller, J.E., Kasture, H., Kurian, G., Gruenwald, C., Beckmann, N., Celio, C., Eastep, J., Agarwal, A.: Graphite: A distributed parallel simulator for multicores. In: HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture, pp. 1–12 (2010). https://doi.org/10.1109/HPCA.2010.5416635

[17] Gutierrez, A., Pusdesris, J., Dreslinski, R.G., Mudge, T., Sudanthi, C., Emmons, C.D., Hayenga, M., Paver, N.: Sources of error in full-system simulation. ISPASS 2014 - IEEE International Symposium on Performance Analysis of Systems and Software, 13–22 (2014) https://doi.org/10.1109/ISPASS.2014.6844457

[18] Butko, A., Garibotti, R., Ost, L., Sassatelli, G.: Accuracy evaluation of GEM5 simulator system. ReCoSoC 2012 - 7th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip, Proceedings (2012) https://doi.org/10.1109/ReCoSoC.2012.6322869

[19] Akram, A., Sawalha, L.: A comparison of x86 computer architecture simulators. (2016). https://api.semanticscholar.org/CorpusID:12104824

[20] Walker, M., Bischoff, S., Diestelhorst, S., Merrett, G., Al-Hashimi, B.: Hardware-validated cpu performance and energy modelling. In: 2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 44–53 (2018). https://doi.org/10.1109/ISPASS.2018.00013

[21] Cebrian, J.M., Barredo, A., Caminal, H., Moretó, M., Casas, M., Valero, M.: Semi-automatic validation of cycle-accurate simulation infrastructures: The case for gem5-x86. Future Generation Computer Systems **112**, 832–847 (2020) https:

//doi.org/10.1016/j.future.2020.06.035

[22] Censier, L.M., Feautrier, P.: A new solution to coherence problems in multicache systems. IEEE Transactions on Computers (TC) **27**(12), 1112–1118 (1978) https://doi.org/10.1109/TC.1978.1675013

[23] Goodman, J.R.: Using cache memory to reduce processor-memory traffic. In: 10th Int'l Symp. on Computer Architecture (ISCA), pp. 124–131 (1983). https://doi.org/10.1145/800046.801647

[24] Gupta, A., Weber, W.-D., Mowry, T.C.: Reducing memory traffic requirements for scalable directory-based cache coherence schemes. In: 19th Int'l Conf. on Parallel Processing (ICPP), pp. 312–321 (1990). https://doi.org/10.1007/978-1-4615-3604-8_9

[25] Chips, Cheese: Microbenchmars GitHub repository. Available at https://github.com/ChipsandCheese/Microbenchmarks

[26] Agarwal, A., Simoni, R., Hennessy, J.L., Horowitz, M.A.: An evaluation of directory schemes for cache coherence. In: 15th Int'l Symp. on Computer Architecture (ISCA), pp. 280–289 (1988). https://doi.org/10.1145/633625.52432

[27] Bharadwaj, S., Yin, J., Beckmann, B., Krishna, T.: Kite: A family of heterogeneous interposer topologies enabled via accurate interconnect modeling. In: 2020 57th ACM/IEEE Design Automation Conference (DAC), pp. 1–6 (2020). https://doi.org/10.1109/DAC18072.2020.9218539

[28] Agarwal, N., Krishna, T., Peh, L.-S., Jha, N.K.: Garnet: A detailed on-chip network model inside a full-system simulator. In: 2009 IEEE International Symposium on Performance Analysis of Systems and Software, pp. 33–42 (2009). https://doi.org/10.1109/ISPASS.2009.4919636

[29] Chips, Cheese: Popping the Hood on Golden Cove. Available at https://chipsandcheese.com/2021/12/02/popping-the-hood-on-golden-cove/

[30] Fog, A.: Instruction Tables. Instruction latencies, throughputs and micro-operation breakdowns. Available at http://www.agner.org/optimize/instruction_tables.pdf (2018)

[31] Sakalis, C., Leonardsson, C., Kaxiras, S., Ros, A.: Splash-3: A properly synchronized benchmark suite for contemporary research. In: 2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 101–111 (2016). https://doi.org/10.1109/ISPASS.2016.7482078

[32] Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The SPLASH-2 programs: Characterization and methodological considerations. In: 22nd Int'l Symp. on Computer Architecture (ISCA), pp. 24–36 (1995). https://doi.org/10.1109/

ISCA.1995.524546