Main Memory Latency Simulation: The Missing Link

Rommel Sánchez Verdejo
rommel.sanchez@bsc.es
Barcelona Supercomputing Center (BSC)
Universitat Politècnica de Catalunya (UPC)
Barcelona, Spain

Kazi Asifuzzaman
kazi.asifuzzaman@bsc.es
Barcelona Supercomputing Center (BSC)
Universitat Politècnica de Catalunya (UPC)
Barcelona, Spain

Milan Radulovic
milan.radulovic@bsc.es
Barcelona Supercomputing Center (BSC)
Universitat Politècnica de Catalunya (UPC)
Barcelona, Spain

Petar Radojković
petar.radojkovic@bsc.es
Barcelona Supercomputing Center (BSC)
Barcelona, Spain

Eduard Ayguadé
eduard.ayguade@bsc.es
Barcelona Supercomputing Center (BSC)
Universitat Politècnica de Catalunya (UPC)
Barcelona, Spain

Bruce Jacob
blj@umd.edu
University of Maryland
College Park, Maryland, USA

ABSTRACT

The community accepted the need for a detailed simulation of main memory. Currently, the CPU simulators are usually coupled with the cycle-accurate main memory simulators. However, coupling CPU and memory simulators is not a straight-forward task because some pieces of the circuitry between the last level cache and the memory DIMMs could be easily overlooked and therefore not accounted for.

In this paper, we take an approach to quantify the missing cycles in the main memory simulation. To that end, we execute a memory intensive microbenchmark to validate a simulation infrastructure based on ZSim and DRAMsim2 modeling an Intel Sandy Bridge E5-2670 system. We execute the same microbenchmark on a real Sandy Bridge E5-2670 machine identifying a missing 20 ns in the simulator measurements. This is a huge difference that, in the system under study, corresponds to one-third of the overall main memory latency. We propose multiple schemes to add an extra delay in the simulation model to account for the missing cycles. Furthermore, we validate the proposals using the SPEC CPU2006 benchmarks. Finally, we repeat the main memory latency measurements on seven mainstream and emerging computing platforms. Our results show that latency between the Last Level Cache (LLC) and the main memory ranges between tens and hundreds of nanoseconds, so we emphasize on properly adjust and validate these parameters in system simulators before any measurements are performed. Overall, we believe this study would improve main memory simulation leading to the better overall system analysis and explorations performed in the computer architecture community.

CCS CONCEPTS

• **Computer systems organization** → *Processors and memory architectures*; • **Computing methodologies** → Massively parallel and high-performance simulations;

KEYWORDS

Main memory, DRAM, Simulation, High Performance Computing.

ACM Reference Format:
Rommel Sánchez Verdejo, Kazi Asifuzzaman, Milan Radulovic, Petar Radojković, Eduard Ayguadé, and Bruce Jacob. 2018. Main Memory Latency Simulation: The Missing Link. In *The International Symposium on Memory Systems (MEMSYS), October 1–4, 2018, Old Town Alexandria, VA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3240302.3240317

1 INTRODUCTION

CPU and memory simulators are inseparable parts of today's computer architecture research; they are extensively used to prototype hardware systems and to estimate performance and energy of a particular design. System simulators were initially focused solely on the CPUs whereas the memory systems were emulated with very simple models, *e.g.,* the fixed latency of the main memory access. However, work of Jacob *et al.* [7, 11] showed that simplistic DRAM modeling can lead to significant simulation errors. The authors advocated for the detailed timing simulation of the main memory and released DRAMsim [27] the first cycle-accurate DRAM simulator, followed by DRAMsim2 [21]. The community accepted the need for a detailed simulation of main memory, and currently, the CPU simulators are usually coupled with the cycle-accurate main memory simulators such as DRAMsim2 [21], Ramulator [13] or NVMain [20].

The CPUs and main memory are clearly separate devices, with different functionalities and typically manufactured by different

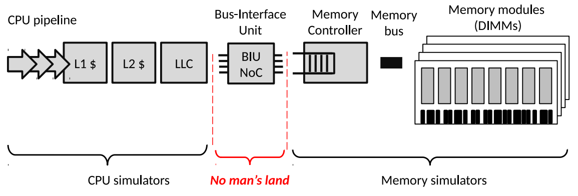

Figure 1: When CPU and memory simulators are coupled, the timings of the memory request between the LLC and the memory controller could be easily overlooked.

companies. However, decoupling their functionalities in total system simulation is not straight-forward. In particular, the DRAM DIMMs are passive devices managed by the memory controller which is a part of the CPU. This functionality includes scheduling, reordering and queuing of the memory requests as well as the detailed DRAM commands, such as `PRECHARGE`, `ACTIVATE`, `READ`, `WRITE`, etc. Since the memory simulators mimic the functionality of the main memory, they also simulate the functionality and the timings of the memory controller (see Figure 1). CPU simulators, on the other hand, typically perform *functional* and *timing* simulation of the memory request only until the LLC.

This means that by coupling CPU and a main memory simulators, all the delays of the memory request of the Bus-Interface Unit (BIU) between the LLC and the memory controller could be easily overlooked, and therefore not accounted for, as we illustrate in Figure 1. The BIU overhead is that of: (1) the request traversing the NoC to the memory controller (longer for writes, due to the data); (2) the request arbitrating for the right to be enqueued in the memory controller's request queue; (3) the request potentially stalling if that queue is full; (4) at the end of a read request, the multi-cycle cost of transferring the data over the NoC between the core and the memory controller.

In this paper, we take an approach to quantify the missing cycles in the main memory simulation. To that end, we execute a memory intensive microbenchmark to validate a simulation infrastructure based on ZSim [23] and DRAMsim2 [27] modeling an Intel Sandy Bridge E5-2670 system. We execute the same microbenchmark on a real Sandy Bridge E5-2670 machine identifying 20 ns missed in the simulator measurements. This is a huge difference that, in the system under study, corresponds to one-third of the overall main memory latency. We also propose multiple schemes to add an extra delay in the simulation model to account for the missing cycles. Then, we execute applications from the SPEC CPU2006 benchmark suite to validate the approach of adding an extra latency to achieve better simulation accuracy. Finally, we quantify the LLC to memory latency for various high-end and emerging platforms and we show its significant range, between 30 ns (POWER8) and 277 ns (Knights Landing); therefore, it is really important to properly adjust and validate this parameter in system simulators before any measurements are performed. Overall, we believe that the issues addressed in this paper would help researchers of the computer architecture community to improve main memory system simulation.

The rest of the paper is organized as follows. Section 2 explains simulation environment and evaluates main memory latency with a microbenchmark for real and simulated systems. This section also proposes approaches to fix the deviation identified between real and simulated main memory latency measurements. Section 3 details the validation of the proposed approaches with SPEC CPU2006 benchmarks, while Section 4 discusses LLC to main memory latency of various high-end and emerging High Performance Computing (HPC) platforms. Section 5 analyzes the validation procedure of state-of-the-art system and memory simulators. Finally, Section 6 presents the conclusions of the study.

Table 1: Cache parameters of the Sandy Bridge EP class processor used in the study.

	L1-D	L2	L3
Size	32 KiB	256 KiB	20 MiB
Latency (in CPU cycles)	4	8	28
Cache line size	64 B	64 B	64 B
Set associativity	8-way	8-way	20-way

2 MAIN MEMORY LATENCY EVALUATION AND SIMULATION ENHANCEMENTS

In this section, we detail the methodology used to model a targeted system into a simulation infrastructure and we describe the microbenchmarks used to discover the main memory access latency. The targeted system we aimed to model is an Intel Xeon E5-2670 Sandy Bridge-EP processor [9] operating at 3.0 GHz. Sandy Bridge is a micro-architecture that is still in use, specially in smaller Tier-1 systems [24] as in the Barcelona Supercomputing Center (BSC) where we conduct our experiments. The main memory comprises four 4 GiB DIMMs devices [22] connected to the processor using four `DDR3-1600` channels. Each processor comprises eight cores where the hyper-threading feature has been disabled like in most HPC systems [5].

2.1 Simulation environment

The simulator infrastructure we chose to use is an integration of two simulators: ZSim [23] as CPU simulator and DRAMsim2 [21] as main memory simulator.

ZSim is a user-level, execution-driven CPU simulator widely used in the computer architecture research community. Developed by researchers from MIT and Stanford University, ZSim is designed for simulation of large-scale systems. However, ZSim was originally developed to simulate Intel Westmere architecture which is no longer being used in HPC domain. One of the tasks that we had to perform was to upgrade and validate ZSim for Intel Sandy Bridge processor. The work to upgrade ZSim consisted of the following steps: First, we adjusted the simulator by updating the instruction latencies obtained through the execution of `CPU` microbenchmarks [25] in the real hardware; Second, we improved the micro-operation fusion and we increased the number of entries in the Reorder Buffer (ROB) from 128 (Westmere) to 168 (Sandy Bridge); Third, we configured the cache hierarchy according to the Intel documentation [9] for a Sandy Bridge-EP Class, summarized in Table 1. Finally, we updated the `L3` caching mechanism implementing the hashing function described in work by Maurice *et al.* [15].

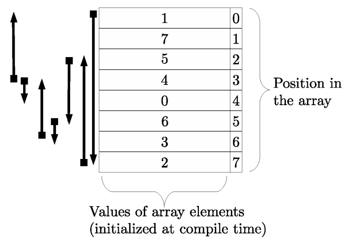

Figure 2: Illustration of pointer a chasing memory access pattern used in the microbenchmark.

Table 2: Pseudo-code: structure of the memory latency microbenchmark.

Line	Source code	Explanation
0001	`register struct cache_line *first asm("rbx");`	`struct cache_line` abstracts the pointer chase data structure
0002	`register int loop_counter asm("ecx");`	`ecx` is the loop counter
0003	`loop_counter = 1000000;`	initialization of the loop counter
0004	`first = getPointerChasePattern();`	`first` holds a pointer to front address of the pointer chase accesses
0005	`first = ptr->next;`	first indirect memory access
0006	`core_loop:`	microbenchmark kernel
0007	`mov (%rbx), %rbx`	indirect load instruction
0008	`mov (%rbx), %rbx`	indirect load instruction
...	. . .	...
10009	`mov (%rbx), %rbx`	indirect load instruction
10010	`dec %ecx`	decrement loop counter
100011	`jnz core_loop`	jump to core_loop

ZSim is easily integrated with a main memory simulator such as DRAMsim2 . DRAMsim2 is a cycle-accurate simulator validated against `Verilog` models for memory devices. We configured DRAMsim2 following manufacturer's documentation with specific timings for the memory device part [22].

2.2 Memory latency microbenchmark

State-of-the-art memory benchmarks such as LMbench [17], `stream` [16] and Intel's Memory Latency Checker (`imlc`) [26] can be used for main memory latency measurements. However, they are not a good fit for our study because it is very difficult to use them in ZSim simulation. LMbench and `stream` rely on compiler's optimization and, `imlc` is a binary-only distributed program so no tailored analysis nor modification to the code could be made. Therefore, as none of the open source existing benchmarks was appropriate for our analysis, we had to design a specific microbenchmark to use for our experiments.

Our microbenchmark is designed to stress the caches and main memory implementing the concept of pointer chasing. Because the microbenchmarks are designed to run on top of an Operating System (OS), a `C` program wraps all functionality outside the microbenchmark objective as: memory initialization, metrics collection, and program cleanup. By doing so, the microarchitectural implications of running on top of an OS are diminished.

In the microbenchmarks prologue, we allocate a contiguous section of memory that stores an array of pointers. The elements on the array are initialized as a circular linked list that follows a pointer chase pattern. Figure 2 portrays an example of such ordering. Our design goals for the microbenchmarks are summarized as: (1) iteratively traverse the whole array; (2) for every memory request, different cache lines must be reached; (3) the memory accesses should have a random pattern, preventing the operation of hardware prefetchers .

The microbenchmarks kernel is written directly in `x86` assembly to meticulously craft the `x86` instruction sequence and avoid compiler optimizations. Table 2 lists the pseudo code for the main memory latency microbenchmarks. Its behavior is explained as follows. (1) The microbenchmark core loop is wrapped-up by the `C` program which is shown from line 1 to line 5. (2) In line 3, the register used as a loop iteration counter (`ecx`) is initialized. (3) In line 4, we summarized the operation to read the contents of a previously generated file containing information about the array size and the random access pattern trough the function call `getPointerChasePattern()`. (4) In line 5, the initial address of the array is assigned to a variable which content is passed as input parameter to the assembly code. (5) From line 5 onwards the assembly is listed: the main part of the benchmark is a sequence of indirect load instructions (`mov(%rbx), %rbx`) that traverse the memory access pattern. (6) The sequence of target instructions is finalized with the decrement of the loop counter register and an exit condition or jump to the beginning of the iteration.

By setting the array size, we target a given level of the memory hierarchy: `L1`, `L2`, `L3` or the main memory. Since there is a dependency between every two consecutive instructions (pointer chasing), the instructions are executed in-order. Therefore, we can compute the latency of each instruction as

$$Memory\ instruction\ latency = \frac{Microbenchmark\ execution\ time}{Number\ of\ instructions}$$

There are two modes for the `C` program to wrap the microbenchmark core, one for the execution on top of the simulator, and other for the execution in the real machine. The difference between the two modes relies only on the way to collect the number of instructions and cycles. On the real system, the measurements are collected via system calls to the Linux `perf` subsystem. The calls are made just before entering the main loop and intermediately after the loop is finished. On the simulator, we use a ZSim feature that allows to enable and disable fast-forwarding up to a specific point in the program execution; the points in the program simulation are set, as in real machine, just before entering the microbenchmark core and as soon as the microbenchmark core is done.

2.3 Methodology

To measure the latency of different levels of memory hierarchy, we vary the array size from 4 KiB up to 3.52 GiB, as shown in Table 3. At each level of memory hierarchy, `L1`, `L2`, `L3` cache and main memory, we select the array sizes to have equidistant measurements.

ZSim does not model the effects of the address translation nor the impact of Translation Look-aside Buffer (TLB) misses. To mitigate the address translation overheads in the real system, we used the 4×1 GiB Huge Pages available in the Sandy Bridge architecture [10] and allocate a contiguous memory space up to 3.52 GiB so few memory pages fit into the TLB [1].

[1]We also quantified the address translation overheads when standard memory pages (4 KiB) are used, but this analysis exceeds the scope of this paper.

Table 3: By setting the microbenchmark array size can we measure the latency of different level in the Intel Xeon E5-2670 Sandy Bridge-EP memory hierarchy. We traverse each memory level with various measurement for array sizes.

Size of each memory level	Microbenchmark array size & stride size	Number of measurements
L1 cache: 32 KiB	4 KiB to 32 KiB,3.5 KiB	8
L2 cache: 256 KiB	60 KiB to 256 KiB,24.5 KiB	8
L3 cache: 20 MiB	2.71 MiB to 20 MiB,2.46 MiB	8
Main memory, 16 GiB	532 MiB to 3.52 GiB,512 MiB	7

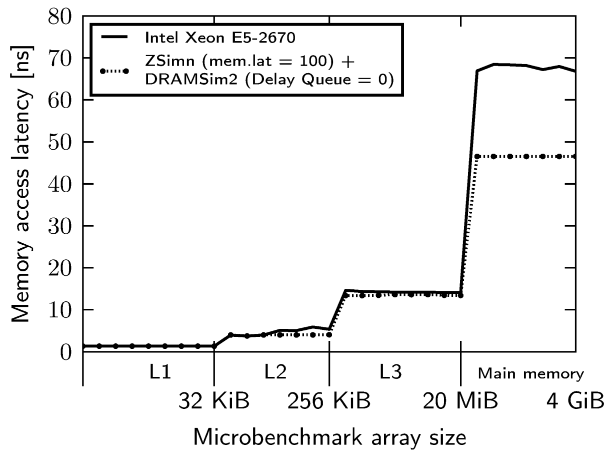

Figure 3: When using default configuration parameters, there is a missing memory latency of 20 ns **between the real system and the integration of ZSim + DRAMsim2.**

2.4 Evaluation

In this section, we investigate main memory access latency deviation from the real system versus the simulation infrastructure using microbenchmarks. We also propose multiple schemes to mitigate this discrepancy.

In Figure 3, the comparison between the real system and the ZSim + DRAMsim2 simulators for caches and main memory latencies is depicted. The X-axis of the figure represents the size of the traversed array, while the Y-axis shows the memory latency in ns. In Figure 3, the four *steps* of the memory hierarchy are distinguishable, each one corresponding to three levels of processor caches and main memory. For the three cache levels (L1, L2, L3), there is a scarcely difference between the two systems, meaning that ZSim's cache contention model accurately represents the real system. Nevertheless, for the memory latency, the difference is noticeable. The main memory access latency of the real system is approximately 66 ns while the ZSim + DRAMsim2 simulate the latency of 46 ns. As we discussed in Section 1, it is not difficult to find an explanation for the missing 20 ns from the main memory access latency. While the ZSim provides timings up to LLC, DRAMsim2 models timings from the memory controller, meaning that the latency of BIU between the LLC and the memory controller is not accounted.

2.5 Potential enhancements

To account for the missing cycles in the main memory access latency as identified in Figure 3, we propose three approaches by adjusting two parameters from ZSim and DRAMsim2.

2.5.1 ZSim enhancement: the `mem.latency` *parameter.* To interpret the function of this parameter, it is essential to understand how ZSim is operated. ZSim is driven by a two-phase algorithm: the *Bound* and *Weave* phases. In the *Bound phase* every core is simulated as they were isolated and for every memory request, a fixed latency is assumed. This fixed latency is configured trough the *mem.latency* configuration parameter. Then, on the *Weave phase* latencies of memory requests from the *Bound phase* are updated with their corrected values; i.e., if DRAMsim2 is integrated, CPU cycles are added up from DRAM simulation for each memory transaction and thus, constitutes the total memory access latency. The `mem.latency` is a ZSim configuration parameter which value is set in the CPU cycle domain. In the current ZSim distribution, the default `mem.latency` value is set to 100 CPU cycles, that in our environment with a 3.0 GHz CPU clock corresponds to the 33 ns.

2.5.2 DRAMsim2 enhancement: `Delay Queue`*.* An extensive approach to add latency to the main memory requests would be to enhance the memory controller simulated in the DRAMsim2. In particular, we upgraded DRAMsim2 memory controller with a `Delay Queue` structure. The purpose of the `Delay Queue` is to insert delay cycles for all main memory transactions to adjust the latency deviation identified from the real system measurements. We implement the `Delay Queue` into DRAMsim2 memory controller using the following design:

(1) The queue has an *unlimited* size
(2) When a transaction arrives at the memory controller, it is immediately redirected to the queue.
(3) Each element (transaction) that joined the queue, is bound to a counter holding the configured delayed.
(4) On each update to the memory controller clock, all the elements on the queue are visited getting their corresponding counter decreased by one.
(5) When an element counter reaches 0, the memory controller fires the transaction to the main memory and deletes the element from the queue.

Since the `Delay Queue` is part of the DRAMsim2 simulator, its value corresponds to the added latency in *the DRAM clock cycles.*

2.5.3 Selected configurations. In this paper, we select and analyze three approaches for the main memory latency adjustments:

(1) Adjusting only `mem.latency` parameter.
(2) Adjusting only `Delay Queue` parameter.
(3) Adjusting both parameters.

Figure 4 portrays the results from the proposed approaches. The X-axis of the figure lists the array size, essentially characterizing different level of caches and main memory, while the Y-axis shows the corresponding memory access latency. We can conclude that all three approaches fix the main memory latency gap between the

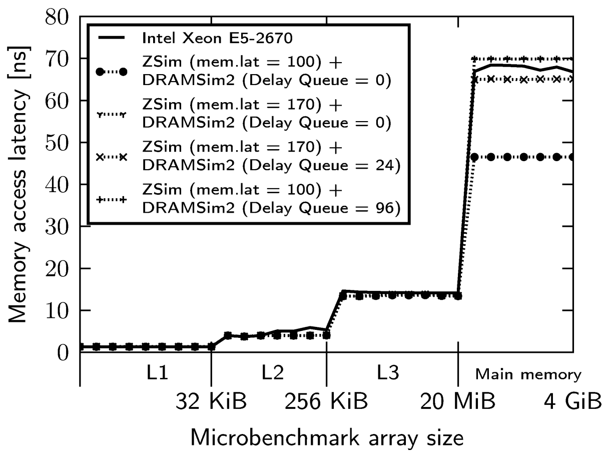

Figure 4: Memory latency for the real system, the default integration of ZSim + DRAMsim2 and three proposed configurations to match the simulator infrastructure with the targeted system.

simulators and the real systems with slight margin[2], while having no impact on the on-chip cache latencies.

Fixing the simulator's memory latency required the following parameter values:

(1) `mem.latency`=170, instead of default 100.
(2) `mem.latency`=100 and `Delay Queue latency`=96.
(3) `mem.latency`=170 and `Delay Queue latency`=24.

However, we should not forget that these measurements are taken for a simple *microbenchmark*. In the following sections, we investigate the impact of the simulator enhancements on more complex workloads as the SPEC CPU2006 benchmark suite [8].

3 EVALUATION: SPEC CPU2006 BENCHMARKS

In the previous section, we analyzed and validated various ways to fix the main memory latency gap between the simulators and the real system. In this section, we investigate the impact of these enhancements on the SPEC CPU2006 benchmark performance and behavior. We execute and evaluate the simulator enhancements on a set of eleven integer and fourteen floating point benchmarks from the SPEC CPU2006 suite [8]. Table 4 lists the benchmarks with their application areas used for the study.

3.1 System performance

For each SPEC CPU2006 benchmark, as it is validated in ZSim main paper [23], we configure the execution to last for the first 50 billion instructions. Then, we compare the three versions of the enhanced simulators versus the default configuration (baseline). The simulators are compared based on the performance difference, calculated as:

$$IPC\,relative\,difference = \frac{IPC_{Enhanced\ Sim.} - IPC_{Default\ Sim.}}{IPC_{Default\ Sim.}}$$

Results are presented in Figure 5. Negative values on the performance difference indicate that the default simulators configuration (baseline) estimates better performance, i.e., higher Instruction per Cycle (IPC) with respect to to the enhanced simulators. This is an expected outcome because all three simulator enhancements increase the main memory access time which leads to performance loss. The first and the third simulator enhancement approaches, `mem.latency = 170` and `mem.latency = 170 & Delay Queue latency = 24`, show similar performance, very close to the default simulator configuration. The significant differences are detected only for `milc` and `libquantum`, that reach up to 20 % of the performance difference. The second approach, adjusting the `Delay Queue` parameter to 96 DRAM cycles, leads to significant differences that, *e.g.,* exceed 50 % for the `milc` and `libquantum` benchmarks.

Overall, we can conclude that, although all three simulator enhancements lead to the same main memory latency of the memory stressing microbenchmarks, their performance impact on the SPEC CPU2006 workloads may differ significantly.

3.2 System behavior

To identify the differences in the benchmark behavior on different systems under test, we use the Top-Down method [29].

3.2.1 Top-Down method: Overview. The Top-Down is designed to understand the application behavior and identify bottlenecks in

Table 4: SPEC CPU2006 benchmarks used in the study

Benchmark	Application Area	Language
bzip2	Compression	C
gcc	C Language Optimizing Compiler	C
bwaves	Fluid Dynamics	Fortran
gamess	Quantum Chemistry	Fortran
mcf	Combinatorial optimization	C
milc	Quantum Chromodynamics	C
gromacs	Molecular Dynamics	C,Fortran
cactusADM	General Relativity	C,Fortran
leslie3d	Fluid Dynamics	Fortran
namd	Molecular Dynamics	C++
gobmk	Artificial Intelligence	C
dealII	Finite Element Analysis	C++
soplex	Simplex Linear Program Solver	C++
calculix	Structural Mechanics	C,Fortran
hmmer	Gene Sequence Analysis	C
sjeng	Artificial Intelligence	C
GemsFDTD	Computational Electromagnetics	Fortran
libquantum	Quantum Computing	C
h264ref	Video Compression	C
tonto	Quantum Chemistry	Fortran
lbm	Fluid Dynamics	C
omnetpp	Discrete Event Simulation	C++
astar	Path-finding Algorithm	C++
sphinx3	Speech Recognition	C,Fortran
xalancbmk	XML Processing	C++

[2]For the microbenchmarks, main memory access latency in real system was measured to be 210 CPU cycles, in the worst case. Among this 210 cycles, 40 cycles correspond to LLC latency. Therefore, *mem.latency* was updated with the remaining 170 cycles.

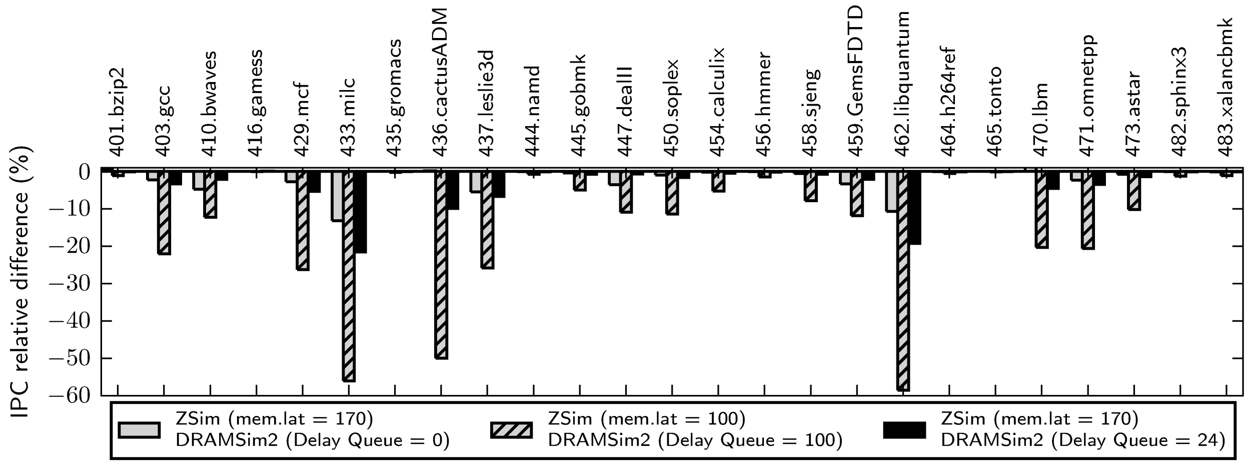

Figure 5: Although all three simulator enhancements lead to the same main memory latency of the memory stressing microbenchmarks, their performance impact on the SPEC CPU2006 workloads may differ significantly.

modern Out of Order (OoO) processors. The model conceptually breaks the CPU engine into two major portions: frontend and backend. The frontend is in charge of decoding instructions from memory and translating them into micro-operations, while the backend executes and retires the work generated as the outcome of scheduling the micro-operations. The place where the frontend feeds the backend with micro-operations is the *issue point*. In the Sandy Bridge micro-architecture, the issue point is 4-slot wide, meaning that it can deliver to the backend up to four micro-operation per cycle. The Top-Down method categorizes application performance in four main groups: *frontend bound*, *bad speculation*, *retiring*, and *backend bound*. The issued micro-operations that are retired at the end of the pipeline are the ones that correspond to the useful pipeline work; the Top-Down classify these issue slots as *retiring*. However, some issued micro-operations are not retired, e.g., because they are part of the mispredicted branch path. These slots are categorized as *bad speculation*. The issue slot can also be empty because the CPU frontend is unable to fill them; these slots are categorized as *frontend bound*. If the frontend is ready to deliver a new micro-operation in the issue point, but there are no available slots because the backend has not freed them from previous micro-operations, the slots are categorized as *backend bound*.

Top-Down go further building a hierarchical tree with other subcategories beneath this four main groups. For example, *backend* category spawns a tree over two other main categories: *Core* Bound and *Memory* Bound where the latter further breaks into *stores*, *L1*, *L2*, *L3* and *main memory*.

Top-Down was designed by Intel, implemented targeting the Ivy Bridge microarchitecture. The authors had given a conceptual approach for each category and the real system, also a list of hardware counters they used to determine each one of the mentioned categories. From the conceptual description and with the hardware counter definition, we have applied the method not only to obtain measurements from the Sandy Bridge micro-architecture but to implement an approximation of this features into the simulator to export the corresponding data. Nevertheless, we just did it for the main four categories. In the simulator used in the study, implementation of the hardware counters required for more detailed analysis, *e.g.,* core, *L1*, *L2*, *L3* and *main memory*, would require significant effort. Sandy Bridge processor used in the study has just a subset of the hardware counters available in the Ivy Bridge architecture used for the Top-Down development. Therefore, for the processor under study, it is unfeasible to perform Top-Down analysis that we provide more details from the one that we presented in this paper.

3.2.2 *Top-Down method: Analysis.*

Figure 6 shows the Top-Down analysis for each benchmark. We plot five bars that are shown in the following order, from left to right:

(1) `mem.latency` = 100 (Default configuration)
(2) `mem.latency` = 170
(3) `mem.latency` = 170 and `Delay Queue latency` = 24
(4) `mem.latency` = 100 and `Delay Queue latency` = 96
(5) Real system measurements.

Each bar shows a Top-Down issue slot breakdown between the four main categories: *retiring*, *bad speculation*, *frontend bound*, and *backend bound*.

For the real system measurements, the right-most bar for each benchmark in Figure 6, we use the standard Top-Down representation in which the sum of all issue slot is scaled to 1. The Top-Down comparison of different systems, however, is not trivial. Scaling Top-Down bars of all the systems to 1 is confusing because improvement of one component, *e.g.,* reduction of the empty issues slots due to the main memory access could be seen as deteriorating of another. Therefore, Top-Down comparison of different systems requires a reference point that will enable meaningful presentation and analysis of the results. Since in all the configuration, we execute the same code for the same number of instructions, our reference point is the number of *retiring* micro-operations. This means that, for a given benchmark, the height of the *retiring* bar is *the same* for the real system and all the simulator configurations. For each bar,

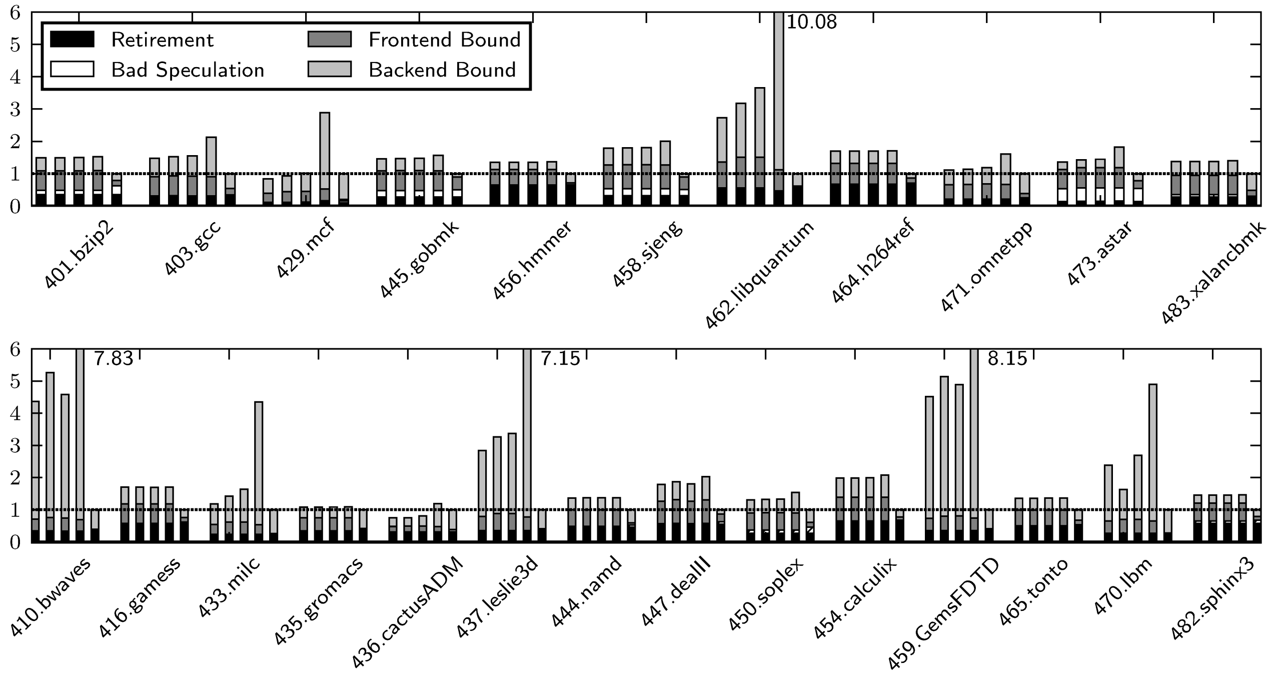

Figure 6: TopDown analysis. For memory intensive benchmarks, simulator enhancements based on the changes in the `mem.latency` ZSim parameter (2nd and 3rd bar for each benchmark) show moderate changes w.r.t. to the default ZSim+DRAMSim2 configuration (1st bar). The simulator enhancements based on the `Delay Queue` in the DRAMSim2 (4th bar), show much larger differences. We also detect a huge difference in the Backend bound issues stalls between the real platform (5th bar) and the all simulator configurations.

i.e., each benchmark and simulator configuration, other Top-Down components, are scaled relative to the *retiring* category. This enables direct visual comparison of different benchmark Top-Down categories in different configurations.

The results are summarized in Figure 6. First, we will focus on the comparison of the different simulator configurations, that is the first four bars for each benchmark. For most of the benchmarks, there is a low to moderate difference between the different simulator configurations. This comes mainly from the fact that these benchmarks have low memory usage [30], so any memory-related configuration has a low impact on the overall application behavior. For the benchmarks with high stress to the main memory, such as the `libquantum`, `bwaves`, `milc`, `leslie3d`, `soplex`, `GemsFDTD`, and `lbm` [30]; the behavior depends significantly on the approach used to correct the latency of the memory requests. The simulator enhancements based on the changes in the `mem.latency` ZSim's parameter, plotted as the second and the third bar for each benchmark, show moderate changes with respect to the default ZSim + DRAMsim2 configuration. The simulator enhancements based on the `Delay Queue` in the DRAMsim2, show much larger differences.

Overall, it is interesting to detect that different simulator enhancements led to practically the same main memory latency of the microbenchmarks (see Figure 4), can lead to significantly different performance and behavior of more complex benchmarks, as shown in Figures 5 and 6.

This opens a question on which out of proposed simulator enhancements should be used to adjust the main memory latency. To address this question, in Figure 6 we also plot the Top-Down breakdown of the Sandy Bridge platform used in the study. However, the most important finding of the presented results is a huge difference in the *backend bound* issues stalls between the real platform and the all simulator configurations for all memory intensive benchmarks.

We analyzed this difference by extensive benchmark profiling (hardware counters) and analysis of the main differences between the simulator and the simulated platform. Our conclusion is that the difference comes mainly from the fact that the data prefetching incorporated in the Sandy Bridge platform leads to significant performance improvements, while the ZSim simulator incorporates no data prefetcher. This makes a huge difference in performance and behavior of the memory intensive benchmarks. Until the data prefetching gap is removed or at least mitigated, we could conclude which out of the proposed simulator enhancements is the closest match to the real system behavior.

4 OTHER SYSTEMS

Up to now, our study focused on the analysis of the main memory latency and the missing cycles in the main memory simulation for the Intel Sandy Bridge E5-2670 processor. In this section, we analyze the LLC to main memory latency on other HPC platforms, including two mainstream HPC architectures which have been

Table 5: Details memory hierarchy for main memory latency measurements

	Mainstream architectures		Emerging architectures				
Platforms	**Nehalem X5560**	**Haswell E5-2698v3**	**Knights Landing**	**Power8**	**ThunderX**	**X-Gene 2**	**X-Gene 1**
Manuf.	Intel	Intel	Intel	IBM	Cavium	APM	APM
Arch.	Nehalem	Haswell	MIC	POWER8	ARMv8-A	ARMv8-A	ARMv8-A
Released	2009	2014	2016	2014	2014	2015	2013
Sockets	2	2	1	2	2	1	1
Cores per Socket	4	16	68	10	48	8	8
CPU freq. GHz	2.8	2.3	1.4	3.49	1.8	2.4	2.4
Out-of-order	Yes	Yes	Yes	Yes	No	Yes	Yes
DP Flops, per cycle, per core	4	16	32	8	2	2	2
L1i	32 KiB	32 KiB	32 KiB	32 KiB	48 KiB	32 KiB	32 KiB
L1d	32 KiB	32 KiB	32 KiB	64 KiB	32 KiB	32 KiB	32 KiB
L2	256 KiB	256 KiB	1 MiB	512 KiB	16 MiB	256 KiB	256 KiB
L3	8 MiB	40 MiB	/	80 MiB	/	8 MiB	8 MiB
Memory conf. per socket	3 ch. DDR3 1333	4 ch. DDR4 2133	8 ch. MCDRAM[a] + 6 ch. DDR4 2400	4 ch. DMI 28.8 GBps	4 ch. DDR3 1600	4 ch. DDR3 1600	4 ch. DDR3 1600
Memory capacity per node	24 GiB	128 GiB	16 GiB (MCDRAM) + 192 GiB (DDR4)	256 GiB	128 GiB	128 GiB	64 GiB

[a] KNL system has been set to *flat mode*, therefore both memories, MCDRAM and DDR4, are exposed as separate NUMA nodes, and the user can choose in which memory the workload executes.

predominantly used in HPC systems so far, Nehalem X5560 and Haswell E5-2698v3, as well as five emerging ones: Knights Landing, POWER8, ThunderX, X-Gene 1 and X-Gene 2. The most important features and the memory hierarchy of the architectures used in this part of the study are summarized in Table 5.

To quantify access latencies to different levels of the memory hierarchy, we used one of the benchmarks from the LMbench benchmark suite [17]. It is a suite of simple, portable benchmarks, which compare different performance characteristics of Unix-like systems. It comprises both bandwidth benchmarks (cached file read, memory read/write, pipe, etc.) and latency benchmarks (context switching, various networking latencies, memory read latency, etc.). We used the memory read latency benchmark, with random-access *read*s in order to mitigate the impact of the data prefetching. By varying input dataset size, we could measure access latency to all memory hierarchy levels. The measured latency comprises the latency of the hardware components (caches, memory controller, main memory), but also the latency of the system software, and virtual to physical memory translation.

Our experiments show significant range in the main memory access latency. In this paper, we focused on the latency between the LLC and the main memory, so in Figure 7 we plot these measurements for the platforms under study. The POWER8 platform has the lowest LLC to main memory latency of 30 ns, followed by the mainstream x86 platforms, Haswell (73 ns) and Nehalem (71 ns), and emerging Arm-based servers Thunder X (82 ns) and XGene 2 (81 ns). The XGene 1 latency is slightly higher, 124 ns, KNL latency reaches 245 ns and 277 ns for DDR4 and MCDRAM memory, respectively.

Overall, we see that the latency between the LLC and the main memory can vary significantly between different platforms. Since its value ranges between tens and hundreds of nanoseconds, it is really important to properly adjust and validate this latency in the system simulators before any measurements are performed.

5 STATE OF THE ART

5.1 Main memory simulators

DRAMsim [27] is the first cycle-accurate DRAM system simulator. Developed with the intention to explore different parameters to achieve optimal memory system performance, authors have abstracted several timing models for technologies like SDRAM, DDR, DDR2, DRDRAM and, FB-DIMMs. Limitations mentioned by Wang *et al.* in DRAMsim original work, were addressed in 2011 by Rosenfeld *et al.* in DRAMsim2 [21] proposal. DRAMsim2 provides a detailed timing simulation of main memory following DDR2 and DDR3 standards. DRAMsim and DRAMsim2 had been validated against Micron's DDR3 DRAM Verilog models [18] where no violation of timing constraints were detected.

Ramulator [13], released in 2016, is another publicly-available DRAM simulator. It provides faster main-memory simulation and enables simulation of the high-end DRAM standards and products such as GDDR5 or HBM. Ramulator DDR3 timings were validated against Micron's DDR3 DRAM Verilog models [18], and no timing constrains violations were detected. An important remark is that Ramulator's correctness was not validated for the memories other than DDR3. Lastly, Ramulator claims to be up to 3 times faster than other state-of-the-art main memory simulators. On ISCA 2012 edition, winners of the Memory Scheduling Championship presented USIMM [6]: a trace-based simulation infrastructure for DRAM devices focusing on memory controller scheduling algorithms. In the same year, Jeong *et al.* [12] have proposed DrSim: a stand-alone traced-based DRAM simulator with the possibility to run in execution mode with a specific version of gem5. To our best knowledge neither USIMM or DrSim had been validated against real models, and unfortunately, both simulators are no longer maintained.

5.2 CPU simulators

`gem5` [2] is the most widely used system simulator. It originated as a merge of the `M5` simulator [3] of the CPU pipeline (validated against an Alpha machine) and the memory hierarchy inherited from `GEMS` [14]. The gem5 can be easily configured to simulate various platforms, and since 2002, more than a hundred of publications are referred to be improving, extending or simply using this simulator. Validation of the simulator versus the actual hardware is delegated to the `gem5` users.

Sniper [4] is an enhancement of the Graphite parallel simulation infrastructure [19]. The simulator models the main memory accesses trough a fixed latency with the option to add a normal distribution on top of it. Sniper is validated against an Intel Xeon X5550 processor (Nehalem architecture) with a set of the SPLASH-2 benchmarks [28]. The validation results show that the Sniper IPC error with respect to the actual hardware is below 25 %.

ZSim [23] simulator is built upon the Dynamic Binary Translation Technique. Currently, it is the fastest simulator capable to perform the simulations up to 300 MIPS of over a thousand cores.

The reason for ZSim's speed is that it partitions the simulation in two phases, *Bound* and *Weave* phases, as described in Section 2.5.

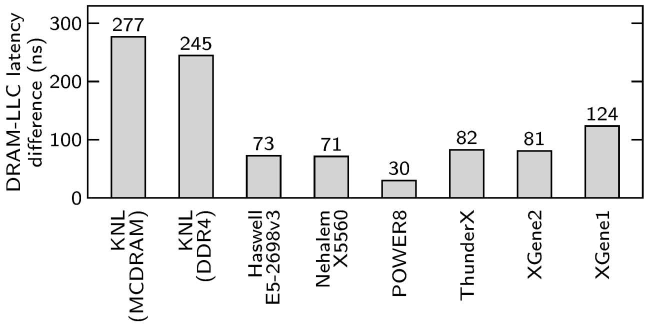

Figure 7: LLC to main memory latency can span over a wide range for various high-end and emerging HPC platforms

In the *Bound phase*, each core is simulated as they were isolated, with no interaction with other cores. This enables fast parallel simulation. Then, in the *Weave phase*, the simulation is corrected to account for a potential collision between concurrent events from different cores such as hitting the same address for main memory accesses. ZSim supports different alternatives for the main memory simulation two of which are: an internal memory model based on the `M/D/1` queue contention, and a software interface to use DRAMsim2. ZSim with the internal `M/D/1` memory model is validated with an actual Intel Xeon L5640 machine (Nehalem architecture) running SPEC CPU2006 and PARSEC [1] benchmarks. The authors report the IPC errors of below 10 %. Furthermore, ZSim authors clarified that using DRAMsim2 restricts the simulation to 3 MIPS, landing outside their design goals. Thus, validation with DRAMsim2 it is not performed in the original paper.

6 CONCLUSIONS

In this paper, we take an approach to quantify the missing cycles in the main memory simulation, and we show that significant latency can be overlooked when CPU and memory simulators are merged without considering delays of all the circuitries that resides between LLC and main memory. In particular, we validate a simulation infrastructure based on ZSim and DRAMsim2 modeling Intel Sandy Bridge E5-2670. Our experiments identify that, in comparison to the real machine measurements, approximately one-third of total main memory latency is not taken into account in the simulated system. Such deviation in main memory latency estimation could place the reliability of a simulation infrastructure in question. We propose multiple schemes to add an extra delay in the simulation model to account for these missing cycles, and validate the approaches using the SPEC CPU2006 benchmarks. Moreover, we measured main memory latency on seven mainstream and emerging compute platforms; the results show a huge range of values for main memory latency between the LLC and main memory devices. Therefore, it is really important to properly adjust and validate related parameters in system simulators before any measurements are performed. We strongly believe, this study identifies an important issue in main memory latency simulation and the approaches proposed in the work will certainly improve main memory simulation techniques.

ACKNOWLEDGMENTS

This work was supported by the collaboration agreement between Samsung Electronics Co., Ltd. and the Barcelona Supercomputing Center (BSC); Spanish Ministry of Science and Technology (project TIN2015-65316-P); Generalitat de Catalunya (contracts 2014-SGR-1051 and 2014-SGR-1272); the Severo Ochoa Programme (SEV-2015-0493) of the Spanish Government; the U.S. Department of Defense under Contract FA8075-14-D-0002-0007, TAT 15-1158; and the U.S. National Science Foundation under Award 1642424.

REFERENCES

[1] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*. ACM, New York, NY, USA, 72–81. https://doi.org/10.1145/1454115.1454128

[2] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7. https://doi.org/10.1145/2024716.2024718

[3] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. 2006. The M5 Simulator: Modeling Networked Systems. *IEEE Micro* 26, 4 (July 2006), 52–60. https://doi.org/10.1109/MM.2006.82

[4] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. 2011. Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulations. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 52:1–52:12.

[5] O. Celebioglu, A. Saify, T. Leng, J. Hsieh, V. Mashayekhi, and R. Rooholamini. 2004. The performance impact of computational efficiency on HPC clusters with hyper-threading technology. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings*. 250–. https://doi.org/10.1109/IPDPS.2004.1303311

[6] Niladrish Chatterjee, Rajeev Balasubramonian, Manjunath Shevgoor, Seth H. Pugsley, Aniruddha N. Udipi, Ali Shafiee, Kshitij Sudan, Manu Awasthi, and Zeshan Chishti. 2012. USIMM: the Utah SImulated Memory Module A Simulation Infrastructure for the JWAC Memory Scheduling Championship. (2012).

[7] Vinodh Cuppu, Bruce Jacob, Brian Davis, and Trevor Mudge. 1999. A Performance Comparison of Contemporary DRAM Architectures. In *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA '99)*. IEEE Computer Society, Washington, DC, USA, 222–233. https://doi.org/10.1145/300979.300998

[8] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News* 34, 4 (Sept. 2006), 1–17. https://doi.org/10.1145/1186736.1186737

[9] Intel Corporation 2016. Intel Product Specification site. (2016). Retrieved May, 2018 from https://ark.intel.com/

[10] Intel Corporation 2016. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Intel Corporation. Reference Number: 248966-033.

[11] Jacob, Bruce and Ng, Spencer and Wang, David. 2007. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[12] Min Kyu Jeong, Doe Hyun Yoon, and Mattan Erez. 2012. DrSim: A Platform for Flexible DRAM System Research. http://lph.ece.utexas.edu/public/DrSim. (2012).

[13] Y. Kim, W. Yang, and O. Mutlu. 2016. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE Computer Architecture Letters* 15, 1 (Jan 2016), 45–49. https://doi.org/10.1109/LCA.2015.2414456

[14] 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. 2005. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *SIGARCH Comput. Archit. News* 33, 4 (Nov. 2005), 92–99. https://doi.org/10.1145/1105734.1105747

[15] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. 2015. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters. In *Proc. of the 18th Int. Symp. on Res. in Attacks, Intrusions and Defenses (RAID'15)*.

[16] John D. McCalpin. 1995. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec. 1995), 19–25.

[17] Larry McVoy and Carl Staelin. 1996. Lmbench: Portable Tools for Performance Analysis. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference (ATEC '96)*. USENIX Association, Berkeley, CA, USA, 23–23. http://dl.acm.org/citation.cfm?id=1268299.1268322

[18] Micron Technology, Inc. 2018. Micron DDR3 SDRAM Verilog Model. (2018). Retrieved Apr 2018 from https://www.micron.com/resource-details/

3caf8e7e-ace6-4a7c-bf04-e374d9c08564

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

[20] M. Poremba and Y. Xie. 2012. NVMain: An Architectural-Level Main Memory Simulator for Emerging Non-volatile Memories. In *2012 IEEE Computer Society Annual Symposium on VLSI*.

[21] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. 2011. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE Computer Architecture Letters* 10, 1 (Jan 2011), 16–19. https://doi.org/10.1109/L-CA.2011.4

[22] Samsung Electronics Co., Ltd. 2011. *240pin Registered DIMM based on 2Gb D-die.* M393B5273DH0 Datasheet.

[23] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-core Systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 475–486. https://doi.org/10.1145/2485922.2485963

[24] Top500 Main site 2018. Top 500 Supercomputer sites. (2018). https://www.top500.org/

[25] R. S. Verdejo and P. Radojković. 2017. Microbenchmarks for Detailed Validation and Tuning of Hardware Simulators. In *2017 International Conference on High Performance Computing Simulation (HPCS)*. 881–883. https://doi.org/10.1109/HPCS.2017.135

[26] Viswanathan, Vish and Kumar, Karthik and Willhalm, Thomas and Lu, Patrick and Filipiak, Blazej and Sakthivelu, Sri. 2018. Intel MLC. (2018). Retrieved Apr 2018 from https://software.intel.com/en-us/articles/intelr-memory-latency-checker

[27] David Wang, Brinda Ganesh, Nuengwong Tuaycharoen, Kathleen Baynes, Aamer Jaleel, and Bruce Jacob. 2005. DRAMsim: A Memory System Simulator. *SIGARCH Comput. Archit. News* 33, 4 (Nov. 2005), 100–107. https://doi.org/10.1145/1105734.1105748

[28] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22Nd Annual International Symposium on Computer Architecture (ISCA '95)*. ACM, New York, NY, USA, 24–36. https://doi.org/10.1145/223982.223990

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

[30] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. 2013. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 55–64. https://doi.org/10.1109/RTAS.2013.6531079