Title
Methodologies for Evaluating Memory Models in gem5

Author
Samani, Mahyar

Publication Date
2021

Peer reviewed|Thesis/dissertation

METHODOLOGIES FOR EVALUATING MEMORY MODELS IN GEM5

By

MAHYAR SAMANI

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Electrical and Computer Engineering

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Jason Lowe-Power, Chair

Venkatesh Akella

John D. Owens

2021

Acknowledgement

I would like to thank my advisor professor Jason Lowe-Power for all of his help and guidance throughout this very challenging process. I would also like to thank my friend and colleague Marjan Fariborz for her help and support. I would not have been able to finish this thesis without the help of Jason and Marjan. Finally, I would like to thank my committee of advisors professors Venkatesh Akella and John D. Owens for their constructive feedback and advice.

Abstract

Recent trends in computer applications and the rate of data generation in the world has created a huge demand for high performance computing. Architecture simulators play a key role in improving the performance of computer hardware. Simulators should be validated by comparing their accuracy against real hardware to prevent faulty data from misleading researchers and engineers. Most cycle accurate simulators have been through such a process. However, full system simulators like gem5 have not been subject to such evaluations. Moreover, to speed up the simulation process gem5 is designed as an event based simulator abstracting away transient details of components in a system making the simulator susceptible to abstraction errors. In this work we propose methodologies and tools for evaluating the accuracy of gem5's models for components in the memory subsystem. Rather than focusing on details of each component by inspecting the microarchitectural statistics, we consider the full system effect of each component. In our methodology, we propose using synthetic traffic to factor out any inaccuracy that might originate from processor models. We concluded that gem5's models for memory subsystem components do not exhibit any unexpected behavior. We also observed a 2× difference between the latency readings from gem5 and DRAMSim3. We believe this difference is caused by the level of abstraction in gem5's memory controller design. Moreover, we model a complete memory subsystem by using publicly available information on the Intel Skylake architecture and use the RandomAccess benchmark to evaluate the accuracy of the memory subsystem. To implement the memory subsystem configuration in the gem5 simulator we use an instance of the ruby cache model that implements a MOESI coherency protocol for a two level hierarchy. We use the same size, latency, and associativity for the L1 cache as the real hardware. Due to the lack of an L3 cache in our cache model, we use an L2 cache to represent both the L2 and L3 caches. For this cache we used the same size and associativity from the L3 in the real hardware, and used a weighted average of L2 and L3 access latencies. We report a 10% error in our GUPS measurements in our simulations. The difference is partially caused by the differneces in the configuration of

caches. Moreover, gem5's lack of support for some of the microarchitectural components in the memory subsystem such as per-bank or per-rank memory queue structures adds to the difference between readings from simulation and real hardware. We believe studies that do not target changes in the memory subsystem could use this configuration as an evaluated set up for their experiments. Moreover, further fine tunings of the configuration could result in more accurate representation of the real hardware.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With Moore's law reaching an effective end, and with the evergrowing demand at the application level, it has become increasingly important for computer hardware to advance with as close agility as computer software [11]. According to Forbes magazine [23], every day 2.5 quintillion bytes of data is created. Moreover, 90% of this vast volume of data has been generated over the last two years. This astonishing speed in generation of data means that computer systems need to provide more and more performance for this raw data to be converted to useful information. Moreover, this trend is also apparent with the emergence of the RISC-V [2] architecture, and companies like SiFive that focus on open source hardware. Research in computer architecture is the most important driving factor in improving computer hardware performance, much of which relies on simulation. The gem5 simulator is one of the well known full system simulators both in academia and industry. Errors and inaccuracies in simulators if sufficiently large could mislead researchers in their decisions; therefore the requirement for validating and evaluating simulators is key. Motivated by making reproducible and realistic computer architecture research, this work targets validating gem5's memory subsystem. It would be an ideal goal to compare the simulator's models to real hardware. However, this approach is not scalable and not possible in certain cases due to the extreme variety of computer hardware. In addition, the simulators are almost as complex

as the systems they simulate. In our work, we use a bottom up approach to evaluate the accuracy of gem5's models, where we start by validating the most simple component in the memory subsystem and build up more complex systems in each stage. First, we validate the DRAM models included in the gem5 simulator by using synthetic traffic and comparing the models to their respective counterpart to those of DRAMSim3. DRAMSim3 is a cycle accurate memory simulator that has been validated by comparing to the timing results of Micron's Verilog workbench [18]. Moreover, we also tested the memory models with the RandomAccess benchmark from HPCC benchmarks [20].

Next, we test the ruby cache system in gem5, by setting up cache hierarchies close to real hardware and measuring their effect on memory access latency. Overall we do not report any unexpected behavior from the memory modules and controller; we observed a 2× difference in latency measurements for our memory tests which is the result of an abstraction error caused by the queue design in gem5's memory controller. Our observations on cache models were in line with our expectations based on real hardware and the concept of caches. Lastly we configured and simulated a memory subsystem based on an Intel Skylake architecture that achieved a 0.043 GUPS measurement comparing to 0.039 for the real hardware [8].

1.1 Motivation

Becuase it is infeasible to prototype computer hardware, the computer architecture research community has been heavily relying on simulation to test novel ideas. Unlike computer software, computer hardware prototyping is expensive, and complicated to the point that hardware manufacturers use a pipeline for their products, as in when a generation of product might be at its latest stages of production, the next one or two generations might be at their earlier stages. Furthermore, computer architecture researchers start with predicting the computational requirements of future applications and require tools to profile their design for such requirements. This complexity in research and development has been a motivating

reason for both academia and industry to use simulation in research. SimpleScalar [4], gem5 [19], RSim [12], SimOS [24], and CACTI [7] are examples of simulators that are widely used both in academia and industry. While some simulators have been validated by comparing against real hardware performance, no microarchitecture simulator has been subject to such scrutiny. The continued use and modification of unverified simulators may consistently be adding errors into experimental results and data, which could be large enough to mislead researchers in their conclusions.

The fast growth of performance demand at the application level has led to hardware designs that exploit parallelism to improve performance, AMD introduced their 64 core Ryzen Threadripper processors in 2019 and the smallest Nvidia Ampere GPU has 4352 FP32 functional units. The increasing parallel compute units require a lot of data movement to be fed, meaning memory performance now plays a much more important role in overall system performance [15, 26, 3, 21, 14].

Overall, computer architecture research could benefit heavily from a validated model for the memory subsystem. To that end, there are many memory simulators such as DRAMSim3 [18], DRAMSys4 [25], and Ramulator [16] that model the interaction of DRAM with the memory controller at the accuracy of clock cycles. For instance, DRAMSim3 has been verified against hardware by comparing its timing results for different memory models against Micron Verilog models [18]. However, none of these simulators are capable of simulating computer systems at a higher levels of fidelity. Simulators like DRAMSim3 do not simulate any component past the memory controller. In order to study the effect of memory systems on the overall performance of a computer system, researchers need to use full system simulators. Full system simulators such as gem5 [19] do not simulate hardware to the fidelity of clock cycles; rather gem5 is a cycle-level simulator that uses event-based simulation. These features of gem5 allow for faster simulation and performance and vast flexibility in its models. The tradeoff of flexibility and performance with accuracy needs to be accounted for when using such simulators. To this end, this work aims at validating the memory components

in the gem5 simulator by studying the high-level effect of memories on performance. The contribution of this work is as follows:

- Proposing a methodology for evaluating the accuracy of performance models for memory subsystem in the gem5 simulator.
- Developing tools for memory performance evaluation.
- Evaluating the accuracy of models for the memory subsystem in the gem5 simulator, using the tools and methodology.
- Pointing out sources of error in the studied models.

Chapter 2 reviews the background knowledge that helps better understand the terms used in this work, Chapter 3 studies and compares previous and similar work to this thesis, Chapter 4 gives details on the proposed methodology and developed tools, Chapter 5 explains the process of evaluating different performance models in the gem5 simulator, and finally Chapter 6 concludes this thesis and lays out objectives for future improvement to this work and the gem5 simulator.

Chapter 2

Background

2.1 gem5

gem5 is an open-source community-based full system simulator. It is widely used for computer architecture research both in academia and industry. The infrastructure provided by gem5 allows users to model computer hardware at the cycle level with the fidelity of booting up a Linux-based operating system. It provides support for various architectures, the most important of which are x86, Arm, and RISC-V. The simulator provides models for the memory subsystem components such as memory modules and caches. The original design of the memory controller model was proposed by Hansson et al. [10]. The proposed architecture has separate read and write queues to queue pending requests on a per controller basis, in contrast to other simulators with per-bank or per-rank queuing structures. The original design of the memory controller in gem5 was able to keep track of every bank connected to it. However, to allow connection for both DRAM and NVM modules to the same memory controller, the bank state information is decoupled from the memory controller and stored in a new object called the memory interface.

The simulator also provides models for caches. gem5 offers two types of caches. The classic cache model provides a faster, less detailed model for the caches. It is not intended

to be used by studies investigating cache coherency protocols; rather, it is meant to be a solution for researchers studying other aspects of a computer system. This model trades off a little bit of accuracy with vaster configurability and higher simulation speed. The ruby cache model is designed to model the cache coherency protocol in detail. In addition, gem5 has an infrastructure for the user to implement proposed coherency designs using its domain-specific language, SLICC. Overall, all the memory subsystem components measure statistical data that represent their performance at a high-level; for instance memory modules measure read/write bandwidth, and caches measure hit/miss rates. Each model is also capable of measuring detailed measurements such as row conflicts in the memory banks.

2.2 DRAMSim3

DRAMSim3 is a cycle-accurate simulator for computer memory. It is constructed with a modular design that allows DRAMSim3 to model major DRAM technologies and the features that commonly come with the memory controllers attached to the different DRAM technologies. These features include capabilities such as dual command issue for the HBM memories and implementing a t32AW instead of tFAW limitation for the GDDR5 memories. The modular design of the simulator allows for the memory controller features to be separated from the DRAM technologies. The simulator configurator file includes information on DRAM technology and the bank organization such as tCL, tCK, number of bank groups, and number of banks per bank group. In addition, it also includes a parameterized list of features and functional parameters for the memory controller, such as memory refresh policy, paging policy, and address mapping. DRAMsim3 uses Micron's DRAM power model [18] to calculate the power consumption on the fly, or it can generate a command trace that can be used as inputs for DRAMPower [6]. DRAMSim3 has been validated against hardware by comparing its timing results against Micron's Verilog bench. Moreover, DRAMSim3 provides an interface for connecting a processing core model or using a trace of memory

accesses to stimulate the memory. Using this feature from DRAMSim3, we have integrated the simulator into the gem5 code base. gem5 and DRAMSim3 use a wrapper object that represents the DRAMSim3 memory controller in the gem5 simulator. This wrapper is capable of translating gem5 memory packets into DRAMSim3 understandable inputs. For each request to the memory, DRAMSim3 sends back the response for that request and the timing information of that request's service. The wrapper then schedules a response event for the request according to the timing information. This event will initiate a sequence of events moving the data back from the memory to its requester. Due to the complexity and variety of computer hardware, it is not feasible to compare the results of a simulator and the real hardware. For instance, to compare the timing results of memory models, the experiment setup might need several different FPGA boards with different onboard memories. Therefore, in order to evaluate the accuracy of memory subsystem models in the gem5 simulator, we used DRAMSim3 as the reference for comparison.

2.3 Sources of Error

Similar to any design process, performance models are also susceptible to many sources of error, including but not limited to modeling errors, specification errors, and abstraction errors. A modeling error is a result of incorrectly implementing the functionality of the system to model. Specification error happens when a misinformed developer models a functionality based on an incorrect understanding of the specification of the modeled system. Lastly, abstraction errors occur when the developer, aiming to gain some simulation performance, models the system at a higher level of abstraction but fails to account for the equivalent timing effect. Moreover, they could result from unimplemented features that have a significant impact on performance. Much like how the hardware is validated, performance models are also validated by inspection. The process of validation by inspection involves analyzing the simulation process and performing sanity checks on the simulation results. It may also

include stepping through the small sequences of the model's execution while observing the model's state. Sanity checking uses the array of statistical counters that are part of the model. These counters gather statistical data such as the IPC or the cache hit rate. It becomes the developer's responsibility to devise simple benchmarks that exercise the boundary conditions of resources in the model. Simulating these benchmarks allows the developers to gain insight into the cause of inaccuracies in the model. The more the developer observes expected behavior from the model, the more confident they could invest in the model. In case of unexpected behavior that could not be explained, the developer will have to search the model for possible errors.

Chapter 3

Related Work

Hansson et al. [10] proposed an event-based simulation model for the memory controller to be used with the gem5 simulator. The choice of event-based simulation is a valid choice for gaining simulation performance. However, this design choice could give rise to inaccuracy in their models; as discussed in the background section, modeling systems at a higher level of abstraction could result in the introduction of errors. In order to validate their design they used synthetic traffic generation to compare the proposed model with its DRAMSim3 counterpart, using bandwidth utilization and latency distribution as their metrics for comparison. They report no unexpected behavior based on their comparison, while the proposed model achieved a 7x speedup in simulation time. Compared to this work, we use the same methodology by using achieved bandwidth and latency to compare the models in gem5 to their counterparts in DRAMSim3. It should be noted that the memory controller has been subject to change ever since its introduction by Hansson et al. [10]. Moreover, to better evaluate the models in gem5, we conduct a series of sanity checks to study the compliance of statistical counters in the models with each other. For instance, we expect to see an average latency reduction if the page hit rate is increased.

In their work, Akram and Sawalha [1] validated the x86 model in the gem5 simulator by comparing the results obtained from gem5 to information gathered from Hardware Monitor

Counters. In their real hardware experiments, they used the perf tool to profile different aspects of the system. In addition, they used available information on Intel's Haswell microarchitecture from a Core i7-4770 processor to match the simulator model to real hardware as close as possible. Finally, they used microbenchmarks that focus on specific subsystems of a processor to target different parts of a model. Using IPC as the metric for comparison, they measured an average error of 39% for control benchmarks, an average error of 8.5% for dependency benchmarks, an average error of 458% for execution benchmarks, and an average error of 38.72% for memory benchmarks. In comparison, rather than focusing on a whole system to evaluate, our focus is to validate different components in the memory system by singling each component and studying its overall full system effect by using metrics such as bandwidth and latency.

Gutierrez et al. and Butko et al. validated the accuracy of gem5 by modeling real systems based on ARM [9, 5]. After making some modifications to the simulator, apart from configuring it to match the experimental board (ARM Cortex A15 based), they were able to achieve a mean percentage runtime error of 5% and a mean absolute percentage runtime error of 13% for SPEC CPU2006 benchmarks. Butko et al. have analyzed the accuracy of gem5 for simulation of a multicore embedded system (ARM Cortex A9 based) [5]. Various benchmarks are related to scientific workloads (SPLASH-2 [28]), media applications (ALPBench [17]), and memory bandwidth (STREAM [22]) were used for validation. The results show that the accuracy varies from 1.39% to 17.94%. We have not been able to find any validation effort for x86 based targets for the gem5 simulator.

Chapter 4

Proposed Methodology and Tools

In this work, we focus on validating the memory subsystem components in gem5: the DRAM Models, and the cache models, at a system level. We use metrics that describe the high-level behavior (performance) rather than using detailed measurements of internal design parameters. These metrics include bandwidth, latency, etc. Also, we consider a bottom-up approach in order to achieve this goal. We start by validating the simplest component that could be validated in a standalone setup (DRAM models) and create a more complex system and validate them using validated components from previous steps.

To this end, we used the gem5 simulator's components that allow for creating synthetic traffic that enable us to test different components in the memory subsystem without concerns for the inaccuracies of processor models in the simulator. We also used DRAMSim3, which is widely known as the go-to simulator for simulating different memory designs, as our reference for comparison. Moreover, as a part of this project, we integrated DRAMSim3 into the gem5 simulator to conduct controlled experiments by only changing the unit under test, the memory controller and the DRAM module in this case, between the test and control experiments. This is made possible by creating a simobject in the gem5 simulator that corresponds to the memory controller in the DRAMSim3 simulator. This way, DRAMSim3, and gem5 can communicate the timing results through the DRAMSim3 simobject. Thus

enabling us to compare the timing results of the DRAM models and remove the effect of other components in the system.

Moreover, to further test our memory system, we implemented a simobject that will run the RandomAccess benchmark as specified by HPCC benchmarks [20]. RandomAccess is a key-value store program designed to stress the memory subsystem by accessing an array's random indices. Furthermore, it uses Giga Updates Per Second (GUPS) as a metric. Hence, the respective simobject is also called GUPSGen.

This Chapter elaborates on the methodology we used for our evaluation. Section 4.1 and section 4.2 describe the specifications and details of PyTrafficGen and GUPSGen, respectively. Section 4.3 provides details on how the memory models were evaluated. Section 4.4 describes the methods used to assess the cache models in gem5. Lastly, section 4.5 discusses the metric used for our evaluation.

4.1 PyTrafficGen

The PyTrafficGen is one of gem5's simobjects that creates synthetic traffic that can act as stimuli for the memory subsystem. It is designed in a way that can replace any component of a full system configuration that is considered to be a data requestor. A requestor is any component that can request services from other components in the system, such as the processor core requesting data to be read. These synthetic traffics are based on a collection of python generators that generate parameters for requests for a memory request. These generators could be either probabilistic or trace-based. This design allows PyTrafficGen to function as a block box replacement for modules that are not implemented, such as a video engine. Despite its flexibility in tuning the timing relationship between a series of requests, a PyTrafficGen cannot model the consistency dependencies between two requests. For example, it does not provide an interface for the user to create a combination of read/write requests to the same address in the memory. Except for the trace-based generators that use

a memory trace to generate the requests, every probabilistic mode of the PyTrafficGen uses certain parameters to define the characteristics of the traffic it generates. These parameters are described below:

- duration: The duration of generating requests in ticks (quantum of time in gem5)
- start address: The lower bound for addresses that the synthetic traffic will access
- end address: The upper bound for addresses that the synthetic traffic will access
- minimum period: The minimum timing difference between two consecutive requests in ticks.
- maximum period: The maximum timing difference between two consecutive requests in ticks.
- request size: The number of bytes that are read/written by each request.
- read percentage: The percentage of reads among all the requests, the rest of requests are write requests.

4.2 GUPSGen

As mentioned previously, the PyTrafficGen lacks capabilities to create synthetic traffic with dependencies between two different requests to the memory. Therefore, we implemented the GUPSGen object. It is a simobject that can imitate a processing core, executing the RandomAccess benchmark [20] by translating indices from an array to their physical address in the memory. The To control the characteristics of the generated traffic, the user can fine-tune the following parameters to their requirements:

- start address: The address at which the array is allocated. This parameter should be a multiple of 64, equivalent to most memory models' atom size. This requirement is not specific to gem5 but the real-world memory models.

- memory size: The amount of memory to allocate for the update table. Should be a power of 2 as specified by the HPCC.
- update limit: The number of updates to be issued. This parameter is used to cut down on simulation time. It should be set considering enough requests to reach a steady state.
- request queue size: The maximum number of outstanding requests to the memory. By resizing the queue, a user can tune the amount of pressure put on the memory system by the generator.

The HPCC [20] guidelines specify setting the memory size to half of the physical memory size at maximum. In case of running multiple instances of this benchmark in parallel, the same restriction applies to the aggregated sum of all the used memories. We implemented the simobject to make requests to the memory subsystem until the memory can not accept any more requests. At this point further requests are queued up in the GUPSGen until the internal queue is full. After this point there are not requests generated until there is room opened in the queue for new requests. This process of applying pressure from the GUPSGen and applying back pressure from the memory subsystem continues until all the updates are made. The GUPSGen creates read requests with a random distribution of addresses. The arrival of responses for every read request will then trigger a write request to the same location in the memory. Threfore the ordering of read and write requests depends on the order at which read responses arrive in the GUPGen and the order at which original read requests are created.

4.3 Memory Models

The existing memory models in the gem5 simulator are designed to focus on how the memory affects the full system performance rather than modeling the details of the interactions of

the memory controller and the DRAM banks. This design consideration enables the event-based simulation that achieves higher simulation performance by not simulating unnecessary details. These details include the transient states that occur between a request for data arriving at the memory controller and the corresponding response being sent to the requester that do not affect the performance of the memory as long as they are accounted for in the memory's timing response. Unfortunately, this means that the memory model in the gem5 simulator cannot model the power consumption of the used models.

Moreover, this design consideration allows for flexibility in designing the memory system. The current memory controller design allows for both DRAM and NVM modules with different timing parameters to be connected to the same memory controller. The memory controller uses separate read and write queues to amortize the cost of bus turnaround between reads and writes. It prioritizes reads over writes since reads are more important to the progress of computer programs. Therefore, writes could be done in bulk to minimize the time required to change the direction of the data bus. In addition, it can also service requests that can be serviced by the pending requests in the queues; for example, it can send back the data for a read if there is a pending write request to the same address.

Regarding scheduling policy, the memory controller supports the First Ready First Come First Served (FRFCFS) policy by default which targets reducing the response latency. However, the user could change the policy as an input parameter to a MemCtrl simobject. Currently, First Come First Served (FCFS) is the alternative to FRFCFS along with any policy the user wishes to implement themselves.

Overall, to define the characteristics of a memory controller, a user can change a collection of parameters, a few of which are explained below:

- read queue size: the maximum number of read requests that can be queued at the memory controller before it starts applying backpressure.
- write queue size: the maximum number of write requests that can be queued at the memory controller before it starts applying backpressure.

- scheduling policy: the policy by which the next request to be serviced is chosen. Currently, it can be either FRFCFS or FCFS by default.

- minimum write threshold: the smallest number of pending writes in case of existing read requests. The memory controller will stop issuing write commands if this threshold is met.

- maximum write threshold: biggest number of pending writes in case of existing read requests. The memory controller will start issuing write commands if this threshold is met.

To validate the high-level behavior of the memory models in the gem5 simulator, we used synthetic traffic generation in a simple setup consisting of one memory controller and a traffic generator. We used the respective models from DRAMSim3 as our reference for comparison. We used bandwidth and latency as metrics for our comparison. Moreover, we compared the effect of demand bandwidth on latency to depict the relation between latency and bandwidth. We expect as the demand bandwidth approaches the specified bandwidth of the memory, the measured latency will increase, forming a hockey stick graph.

4.4 Cache Models

The gem5 simulator provides two classes of models for the cache components in the memory system. The classic cache model is intended to be used by researchers who inspect aspects of the system other than the coherency mechanism. It requires minimal configuration and enables faster simulation. The Ruby cache model implements the details of different cache coherency protocols and requires more configuration by the user. This model is better geared toward research studies that target the coherency mechanism.

4.4.1 Classic Caches

The classic cache model implements a snooping MOESI protocol. The caches in this model are by default non-blocking caches with Miss Status Handling Registers (MSHR) and write buffers and could also be equipped with prefetchers. This model also allows for almost any arbitrary cache configuration with any number of caches at any level. In addition, the classic cache model requires minimal configurability by the user and the user is only responsible for describing the overall configuration of caches, such as the number and size of caches at each level and the number of levels. Each request for data propagates in the expected fashion, where a miss in the L1 results in a snoop on the local L1 to L2 bus. In case of no response, the same process will happen at the lower levels of the hierarchy until the request is either serviced by a cache or by a memory. Due to the requirements for this model for caches, it does not store information about transient states in the caches, which could result in stale values being returned to the processing cores. To get around this problem, and by sacrificing a little bit of timing accuracy on snooping request, the simulator uses express snoops for classic caches to service snoop requests on the L1 to the last level cache path atomically.

4.4.2 Ruby Caches

The Ruby cache model implements a detailed model for the cache hierarchy. It includes models for cache controllers, directory controllers, etc. In addition, it can model inclusive/exclusive cache hierarchies with different replacement policies and cache coherency protocols [19]. These models are modular, flexible, and configurable; as an example, users can specify the size of the line fill buffer in each cache or choose the replacement policy. The Ruby cache model was designed with the following requirements in mind.

- separation of concerns: The model is designed in a way where different scopes of the design could be changed without changing other scopes. For example, the coherence protocol details are separated from the cache line size, associativity, and Network on

Chip topology.

- rich configurability: the model allows for almost any architectural parameter affecting the functionality and timing of the cache hierarchy to be changed.
- rapid prototyping: through the domain-specific language called SLICC, a user can specify the functionality of various controllers.

gem5 also comes with a few cache coherency protocols natively implemented. Below is a list of supported protocols.

- MI example: a basic protocol that supports only one level of cache.
- MESI Two Level: models a single chip, 2-level, strictly-inclusive hierarchy.
- MOESI CMP directory: models a multiple chips, 2-level, non-inclusive (neither strictly inclusive nor exclusive) hierarchy.
- MOESI hammer: models a single chip, 2-level private, strictly-exclusive hierarchy.
- MESI Three Level: 3-level caches, strictly-inclusive hierarchy. Based on MESI Two Level with an extra L0 cache.
- CHI: a flexible protocol that implements Arm's AMBA5 CHI transactions. Supports configurable cache hierarchy with both MESI or MOESI coherency.

Due to its limited support for coherence protocols, we focus our evaluation exclusively on the Ruby cache models. They can better configure a high-performance cache hierarchy and better model the timing model of our caches. As we have already mentioned, the focus of this work is to validate the components of the memory subsystem by their high-level attributes that affect the overall performance of a full computing system. To validate the cache models, we use validated models from the memory validation step and measure the achieved latency and bandwidth of a system comprised of synthetic traffic generators (PyTrafficGen or GUPSGen), cache hierarchies, and memory models.

4.5 Metrics

To evaluate different components of the memory subsystem at a high level, we chose bandwidth and latency as metrics to evaluate the simulator models. We believe bandwidth and latency are the most important metrics that define the performance of the memory subsystem at a high level. Moreover, current memory designs target trading off bandwidth and latency. For example, while HBM memory can deliver 256 GB/s of bandwidth with latencies as high as 100 ns, DDR4 memory can only deliver 25.6 GB/s of bandwidth with latencies as low as 10 ns. Therefore, understanding this trade off and how the models in gem5 account for this trade off is key.

4.5.1 Bandwidth

Memory bandwidth could affect the performance of highly parallel applications significantly. For instance, a Graphics Processing Unit (GPU) hides high memory access latency by making rapid context switches, which impose a significant bandwidth requirement on the memory. Therefore, we use bandwidth as one of our metrics to measure memory system performance. Moreover, this metric is useful for capturing the behavior of our components in different conditions. In addition, it would be easy to make a comparison between different setups. For example, one would expect any memory to perform at its peak performance when accessed with a sequential pattern of accesses since memories are designed in a way that exploits spatial and temporal localities that may exist in a computer program. Moreover, using bandwidth as a metric, we could make a comparison between different models of memory. For example, we know that HBM can deliver more bandwidth than LPDDR3 under the same demand bandwidth.

4.5.2 Latency

While bandwidth has a significant impact on the overall performance of highly parallel applications, latency could play a significant role in achieving high performance in single-threaded applications or multi-threaded applications that require a considerable amount of data to be shared. Using this rationale, we chose latency as another metric for evaluating the high-level behavior of different memory subsystem components. In addition, similar to bandwidth, we can make a cross-comparison between different setups. For example, we can expect that using an open page policy in the memory controller would highly benefit those applications that exhibit a sequential access pattern to the memory. On the contrary, using a closed page policy would incur a less significant penalty for an application that exhibits random accesses in the memory controller.

Chapter 5

Evaluation

The goal of this work is to evaluate the accuracy of different models of gem5's memory subsystem at a level which is important to the overall performance of a complete computing system. In order to achieve this goal, the ideal solution is to run real world applications in gem5's full system simulation mode and measure the performance metrics of our memory subsystem. However, this is not possible at the time of writing this work, due to the inaccurate processor models that can not accurately model those features of a modern process that pressure the memory subsystem. To circumvent this limitation, we use synthetic traffic generation to create traffic that matches the general profile of computer applications. Then we use bandwidth and latency as metrics for our evaluation as they capture the overall behavior of components of the memory subsystem that concerns the full system performance of a computer system. This chapter present the results of our experiments and is organized as follows: section 5.1 describes the experiments used to evaluate the memory models in the gem5 simulator and presents the results of those experiments, and section 5.2 describes the experiments used to evaluate the ruby cache model in the gem5 simulator and presents the results of those experiments. Lastly, section 5.3 describes the evaluation of a complete memory subsystem along with its comparison to real hardware.

5.1 Memory Tests

5.1.1 Synthetic Traffic

In order to validate the accuracy of the memory controller in the gem5 simulator we used synthetic traffic to measure the average bandwidth and latency of requests to the memory models. We used each model's respective counterpart in DRAMSim3 as the reference, We used two types of stream and random traffic to validate our models. Figure 5.1 describes the system simulated in gem5.

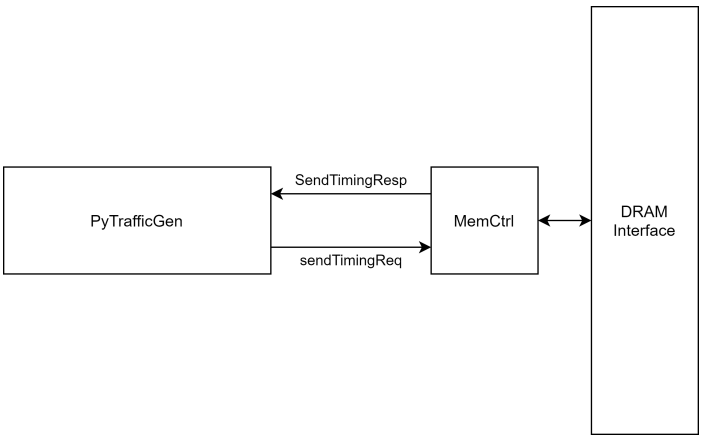


Figure 5.1: Diagram of the system used to test memory models.

Both stream and random modes use user-defined parameters that describe the timing behavior and access range and intensity of the series of requests sent to the memory. These parameters are listed below:

- minimum period: determines the lower bound for the timing difference between two consecutive requests.
- maximum period: determines the upper bound for the timing difference between two consecutive requests.
- block size: determines the number of bytes that is read/written with each request.

- minimum address: determines the smallest address that is accessed by the synthetic traffic.
- maximum address: determines the largest address that is accessed by the synthetic traffic.

Based on what we described above in order to determine the demand bandwidth generated by the traffic generator, we first set minimum period and maximum period to be equal to each other and rename them as injection period, then we could use an equation like 5.1:

$$\text{bandwidth}_{\text{demand}} = \frac{\text{size}_{\text{block}}}{\text{period}_{\text{injection}}} \rightarrow \text{period}_{\text{injection}} = \frac{\text{size}_{\text{block}}}{\text{bandwidth}_{\text{demand}}} \tag{5.1}$$

Looking at the above equation we can determine the injection period using block size and demand bandwidth, this way we can tune the demand bandwidth the traffic generator generates. As previously mentioned, simulation models in gem5 are very flexible in their behavior. For example memory models can respond to any request of any size. In the real world design of memories there are only one or two possible sizes of requests that could be serviced by the memory which is referred to as the memory's atom size. However, DRAMSim3 models only accept requests equal to the atom size of the memory. Therefore, we chose block size to be equal to the memory's atom size, for example DDR4 has an atom size of 64. Using that block size, we were able to determine injection period based on our desired demand bandwidth.

The PyTrafficGen, in its stream (linear) mode, will start creating requests starting from the minimum address, then it will increase the address by the block size to generate the address for the next request, which will be issued at a random number of ticks later. This random number is generated from a uniform distribution between the minimum period and maximum period. By equating the minimum and maximum period, the generator will make requests at a constant rate. In addition, a PyTrafficGen working in random mode will generate accesses that are almost independent of each other and are chosen from a uniform

distribution of addresses between the minimum and maximum address.

We chose three of the common memory models in gem5 for comparison to DRAMSim3. DDR3 is still used by many IoT devices and also some accelerators such as the first version of the Google TPU [13], DDR4 is the memory standard for consumer computing products such as Desktops and Laptops, and HBM has recently gained traction for use in GPUs such as AMD Radeon VII and Nvidia Titan V. Table 5.1 describes the specifications for each of the mentioned memories along with the injection period for PyTrafficGen to create demand bandwidth equivalent to the memory's peak theoretical bandwidth.

Due to our high-level approach toward evaluating the accuracy of each model, we also used configurable parameters from each model that would have a discriminatory effect on the performance of applications. Therefore, we chose parameters that if changed would enhance the performance for some applications and diminish the performance of others. In order to achieve the best performance, software developers try to exploit the existing locality in their algorithms to improve the memory access latency; the optimal order with which the addresses in the memory should be accessed depends on the address mappings and the paging policy of the memory controller. Below is a list of the different address mapping that we used as options (it should be noted that we used all of the different address mappings that are implemented in the gem5 simulator):

- RoRaBaChCo: Row:Rank:Bank:Channel:Column
- RoRaBaCoCh: Row:Rank:Bank:Column:Channel
- RoCoRaBaCh: Row:Column:Rank:Bank:Channel

We used Open Page and Close Page as our options for paging policy. Moreover, we laid out

Memory	BUS Clock (MHz)	BUS Width (bits)	Peak Bandwidth (GB/s)	Atom Size	Injection Period for PyTrafficGen (ticks)
DDR3	1600	64	12.8	64	4768
DDR4	2400	64	19.2	64	3104
HBM	1000	128	16	64	3725

Table 5.1: Memory specifications and configurations.

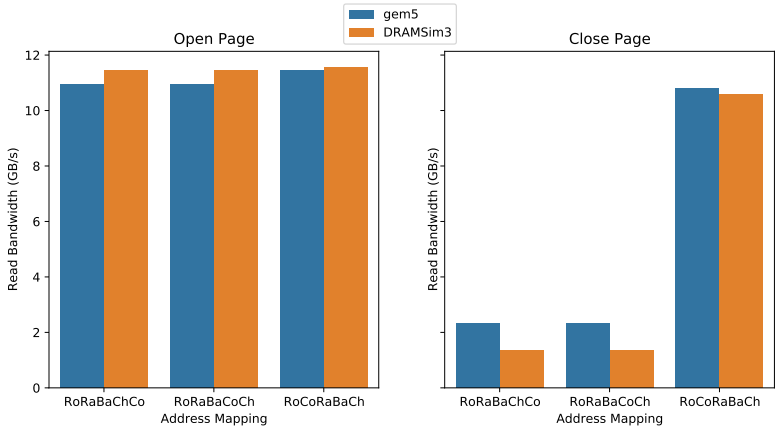
a set of assumptions based on the different combination of inputs that we will try to attest for each memory model. Below is a description of these assumptions:

- When using linear traffic, a memory with an open page policy should provide more bandwidth than a memory with close page policy.
- When using linear traffic RoCoRaBaCh should provide the most bandwidth and achieve the least latency. This is due to the fact that with this address mapping, bits associated to the corresponding bank change more rapidly and expose more parallelism in the sequence of linear accesses.
- When using random traffic, bandwidth and latency measurement should not be affected by the change of address mapping and/or paging policy, this is due to the lack of locality in an ideal random access pattern.
- When using linear traffic, a memory with open page policy should achieve lower latency than a memory with close page policy.
- When using random traffic, a memory with close page policy should achieve lower latency than a memory with open page policy.

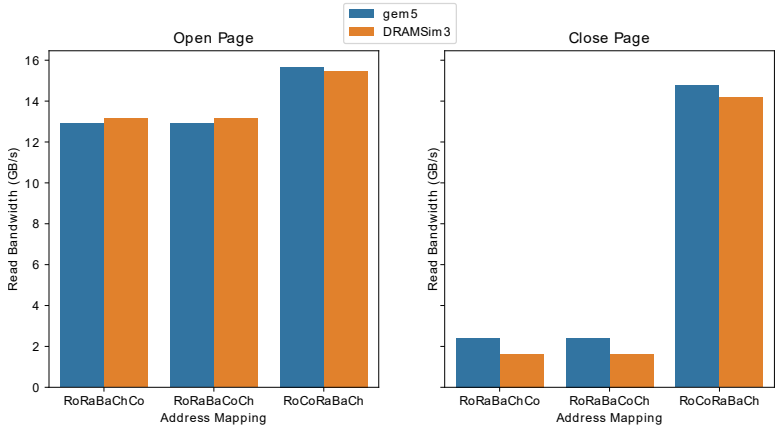
5.1.2 Comparison to DRAMSim3

Figure 5.2 shows a comparison of measured bandwidth between the DDR3, DDR4, and HBM models in the gem5 simulator and their counterparts in DRAMSim3 under linear traffic. The results show little difference between the measured bandwidths from gem5 models and DRAMSim3 models. For the RoCoRaBaCh address mapping the paging policy does not affect the measured bandwidth as much as other address mappings. This is due to the fact that with a linear access pattern the lower bits change with every request. Also since we are using a single channel setup, no bits are allocated for the channel, meaning that every request ends up in a different bank from its predecessor. In regards to our expectations with linear traffic:

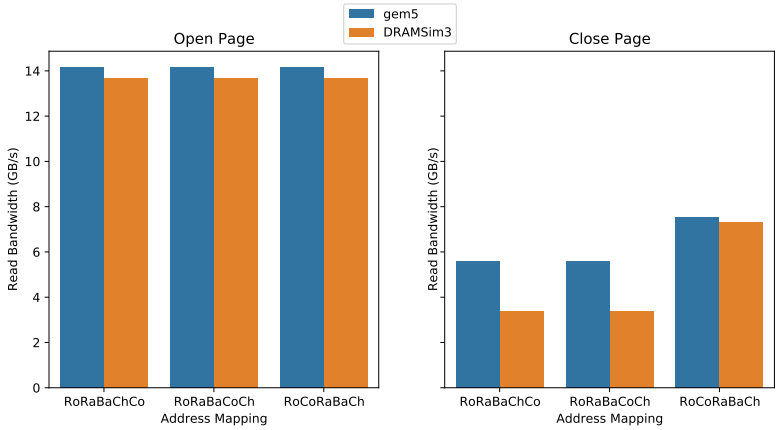
- The results show that while using a linear traffic the memories with an open page policy provide more bandwidth.
- The results also show that a memory with RoCoRaBaCh provides the most bandwidth under a linear traffic regardless of the paging policy.



(a) DDR3



(b) DDR4



(c) HBM

Figure 5.2: Comparison of measured bandwidth different memories under a linear traffic equivalent to their peak theoretical bandwidth.

Figure 5.3 shows a comparison of read latency between the simulation models for DDR3, DDR4, and HBM in gem5 and DRAMSim3 for linear traffic. There is almost a 2× difference between latencies in gem5 and DRAMSim3. We believe this is an abstraction error caused by the following:

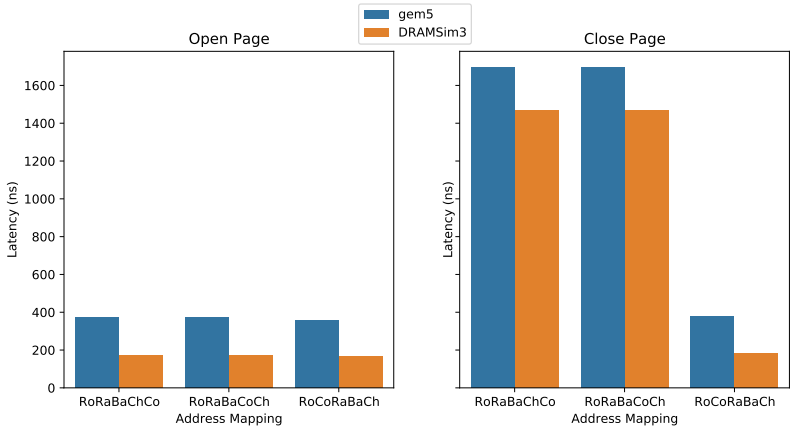
- The gem5 memory controller does not implement the memory queue in structure in detail and only implements a shared queue for every channel in the memory controller, whereas DRAMSim3 uses a per-rank structure. Since all the memories have 2 ranks, in order to match the size of the queue in gem5 with DRAMSim3, we used 2× the number of entries in gem5. This will result in higher latency in comparison to DRAMSim3.

- The gem5 memory controller does not fully implement the bank group feature in memories, where consecutive accesses to banks from different banks incur a smaller serialization time (t_{CCD_S} instead of t_{CCD_L}).

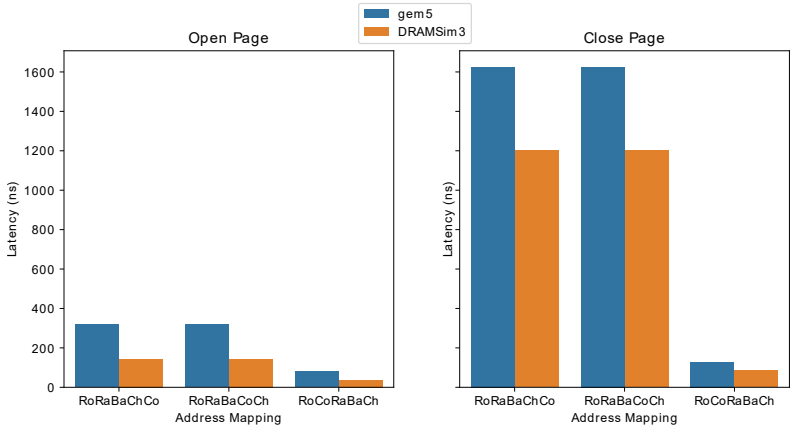
In addition, as discussed previously, the RoCoRaBaCh address mapping can hide the access latency even under a closed page policy in a memory channel. Moreover, in regards to our preset list of assumptions, here is a overview of this model's compliance with those.

- The results show that using open page policy would result in lower latency under linear traffic.

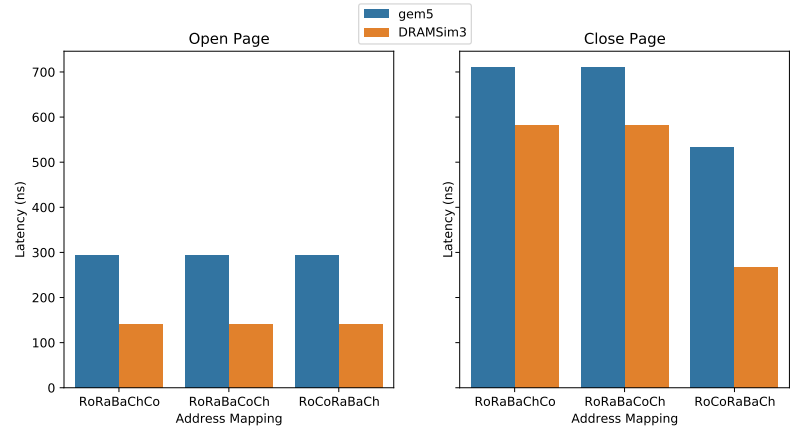
- The results show that among all the address mappings using RoCoRaBaCh address mapping results in the lowest measured latency.



(a) DDR3



(b) DDR4

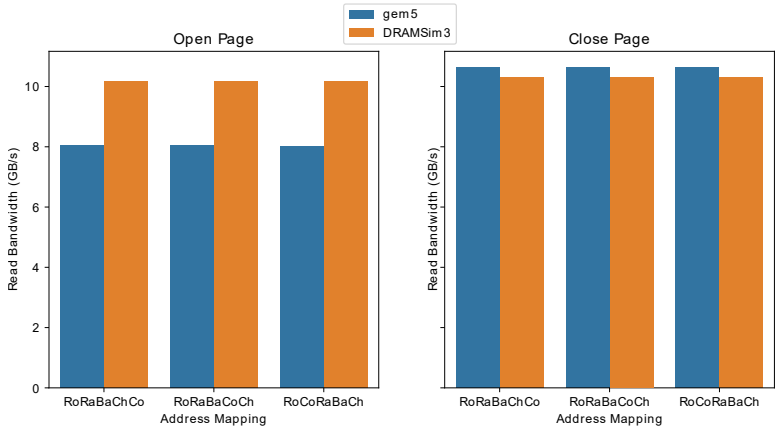


(c) HBM

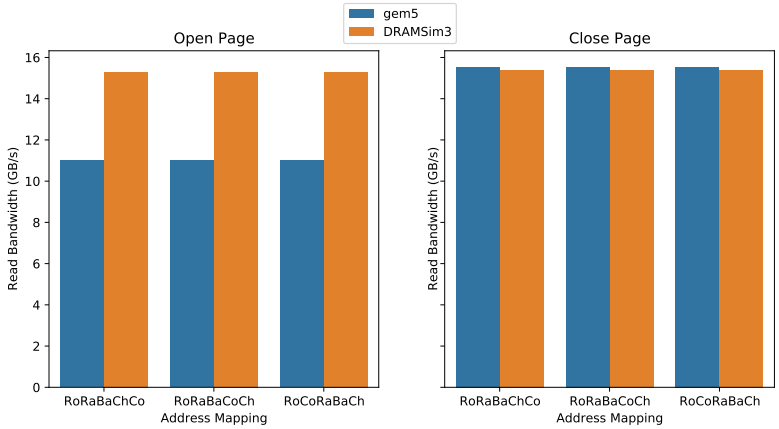
Figure 5.3: Comparison of measured latency different memories under a linear traffic equivalent to their peak theoretical bandwidth.

Figure 5.4 shows a comparison of measured bandwidth between gem5's models of the DDR3, DDR4, and HBM and their DRAMSim3 implementations for random traffic. The results show little difference between the measured bandwidths from gem5 models and DRAMSim3 models. The results show that under random traffic the performance metrics do not change with the the change of the memory which is in line with our expectations; Different address mappings target exploiting the locality in memory accesses to deliver a higher bandwidth or lower latency which a random access pattern lacks. We also observe a small increase in measured bandwidth when using close page instead of open page policy, which is due to the small time saved by precharging the row after every access and parallelising the precharge cycles with other banks' activations. In regards to our expectations with random traffic:

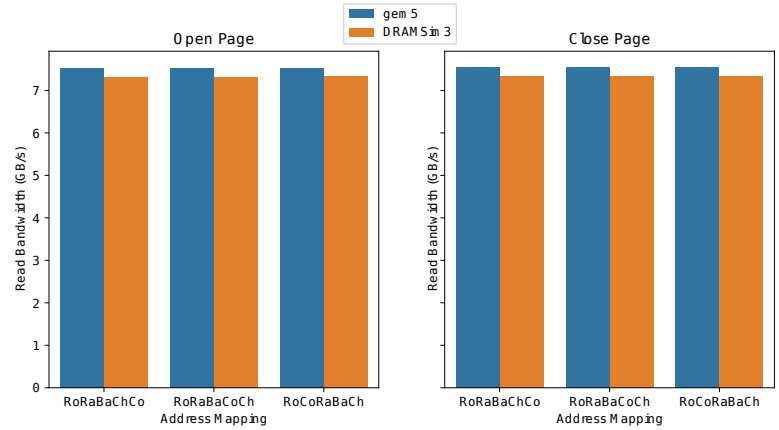
- The results show that under random traffic, bandwidth measurements are address mapping agnostic.



(a) DDR3



(b) DDR4

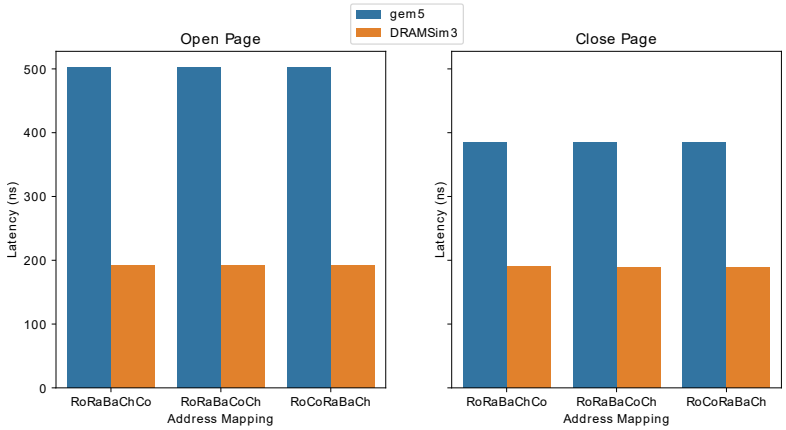


(c) HBM

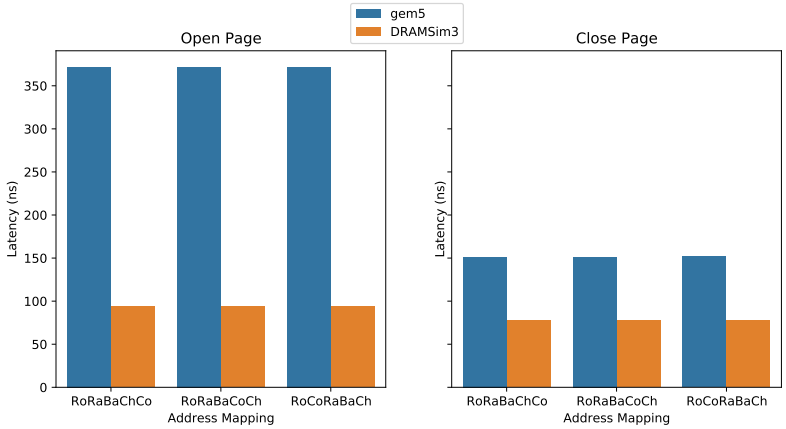
Figure 5.4: Comparison of measured bandwidth different memories under a random traffic equivalent to their peak theoretical bandwidth.

Figure 5.5 shows a comparison of read latency between the simulation models for DDR3, DDR4, and HBM in gem5 and DRAMSim3 for random traffic. There is almost a 2.5× difference between the measured latencies in gem5 and DRAMSim3 with open page policy and a 2× difference with close page policy. The errors originate from the same source as explained before for linear traffic. Moreover, in regards to our preset list of assumptions here is a overview of this model's compliance with those.

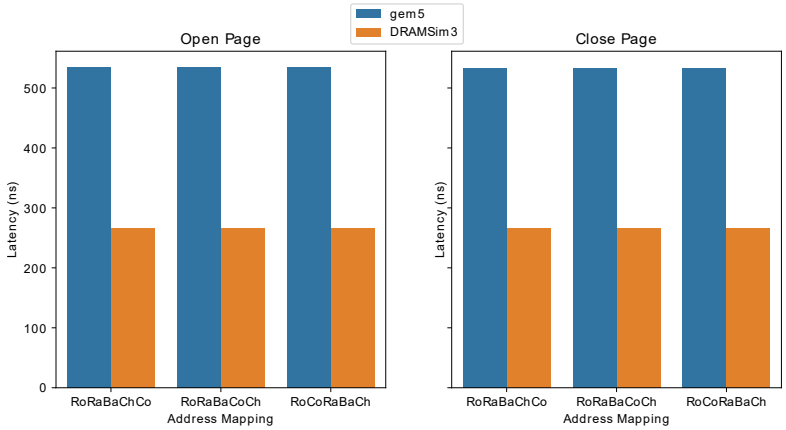
- The results show that using close page policy would result in lower latency under random traffic.
- The results show that under random traffic, latency readings are not affected by the change of address mapping.



(a) DDR3



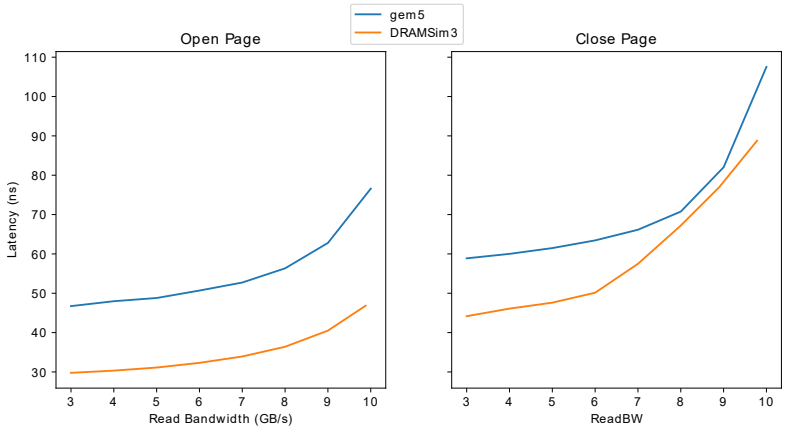
(b) DDR4



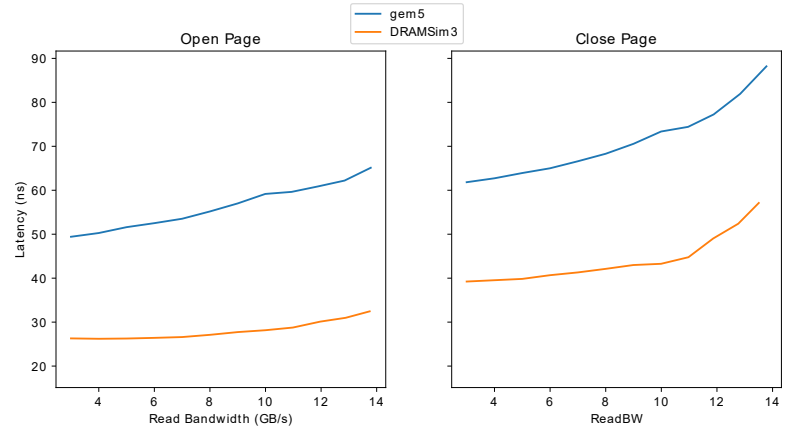
(c) HBM

Figure 5.5: Comparison of measured latencies different memories under a random traffic equivalent to their peak theoretical bandwidth.

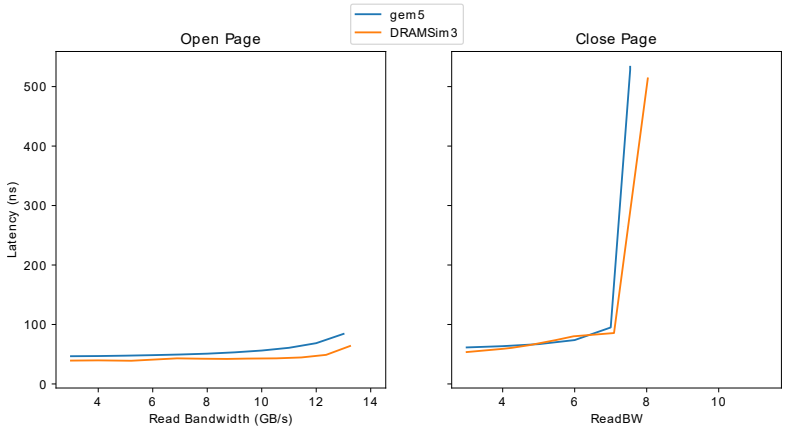
Figure 5.6 shows the relation between measured bandwidth and latency for DDR3, DDR4, and HBM in both gem5 and DRAMSim3 under linear traffic, and Figure 5.7 shows the relation between measured bandwidth and latency for DDR3, DDR4, and HBM in both gem5 and DRAMSim3 under random traffic. Comparing the two metrics and the trend of changes in each of the variables, we could expose the overall behavior of the memory model. We would expect as the bandwidth readings increase, the latency readings increase as well, which is aligned with what the results show in the figures. Moreover, based on our previous latency tests we saw a $2\times$ difference in the measured latency. However, that source of inaccuracy in the absolute values could be disregarded since based on the observation in the trends, the difference in latency measurements do not show any random behavior. In contrast, they exhibit an almost constant difference between gem5 and DRAMSim3. Therefore, in experiments that do not target memory performance rather use memory to evaluate novel ideas in other aspects of the system the relative performance of their different configurations should not be affected by our measured difference in latency.



(a) DDR3

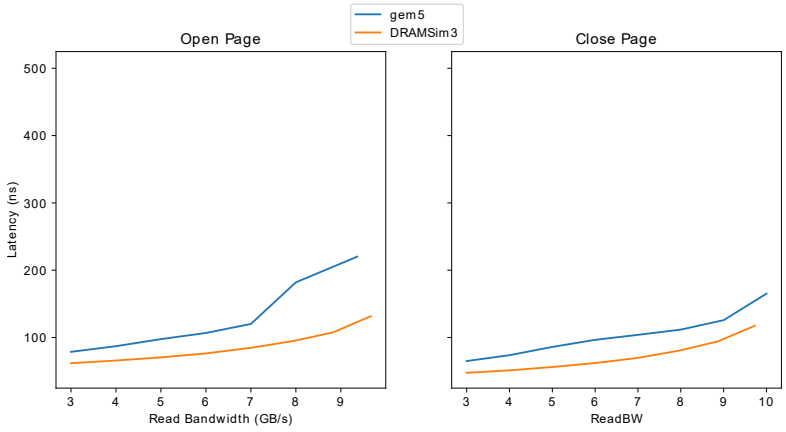


(b) DDR4

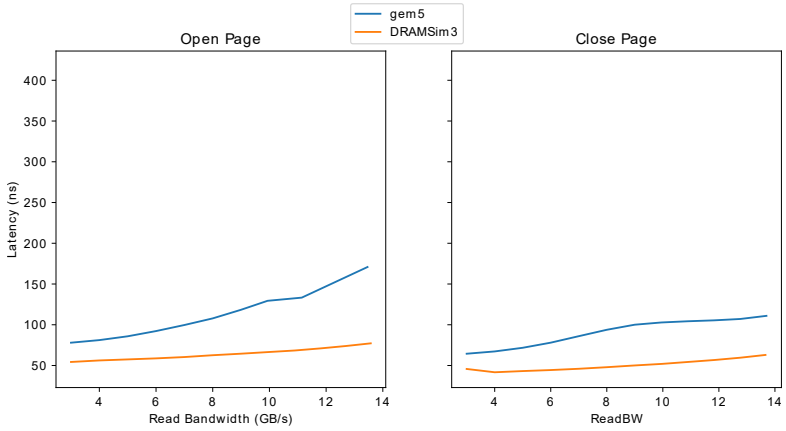


(c) HBM

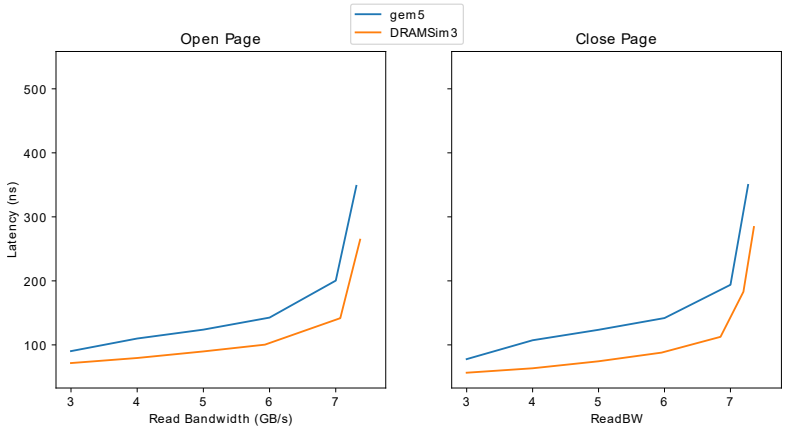
Figure 5.6: Tradeoff of bandwidth and latency under linear traffic for different memories.



(a) DDR3



(b) DDR4



(c) HBM

Figure 5.7: Tradeoff of bandwidth and latency under random traffic for different memories.

5.2 Cache Tests

Caches are used in computing systems to provide low latency, high bandwidth access to the memory subsystem. Modern computer systems use multiple levels of caches to further improve the performance of the memory subsystems. However, the achieved latency and bandwidth is dependent on the frequency of accesses serviced by each layer of the memory, as the access reaches to the lower levels of memory it achieves less bandwidth and higher latency. To attest this fact we used synthetic traffic with a two level cache hierarchy. The L1 cache is an 64 KiB 8-way set associative cache connected to a 512 KiB 4-way set associative L2 cache. Diagram 5.8 describes the test bench system used for this experiment.

In this test we measured the average read latency of accesses in a scenario described below:

First we warmed up the L2 cache by reading addresses from 0 to 512 KiB. Next, we fill the L1 cache by reading addresses from 0 to 64 KiB. After this phase we reset the statistics gathered up until this point and simulate the main phase. The main phase consists of reading addresses from 0 to a variable range. As this range increases, the access addresses start moving down the memory hierarchy, resulting in an increase in the access latency and a decrease in the measured bandwidth. Figure 5.9 shows the effect of the accessed range on the measure latency and bandwidth.

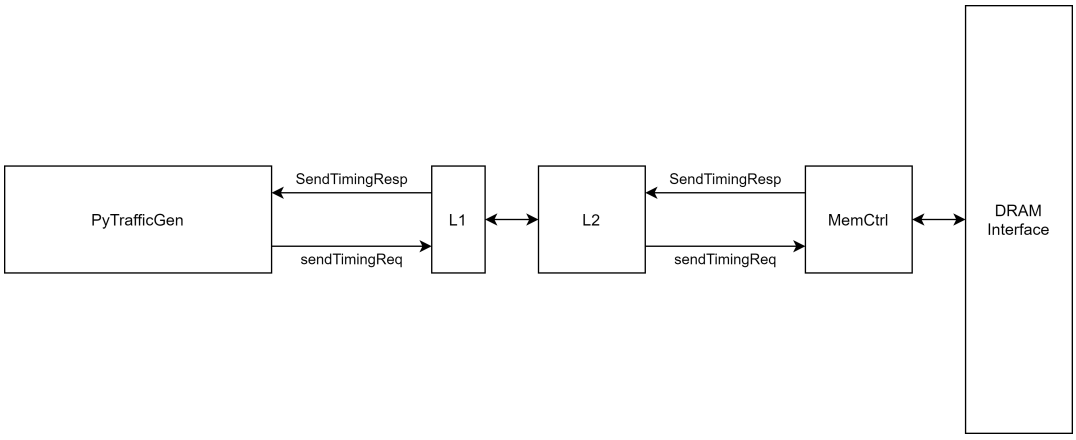


Figure 5.8: Diagram of the system used to test cache models.

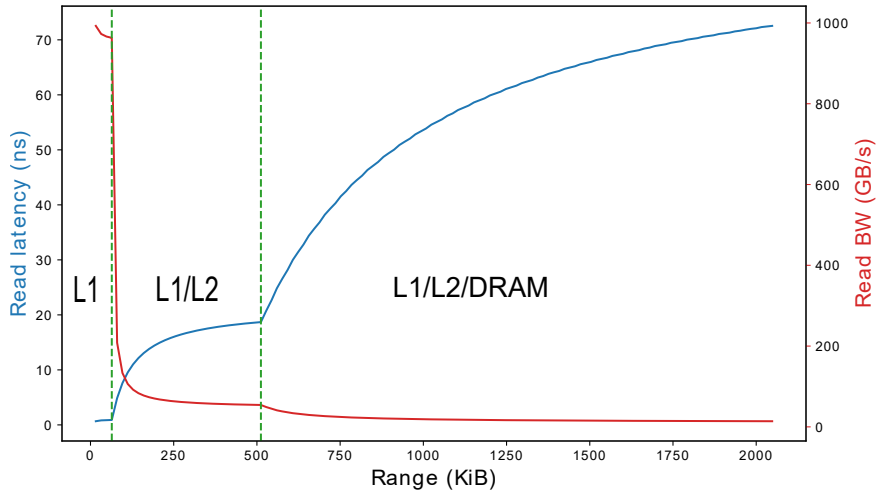


Figure 5.9: Effect of hierarchical memory on access latency and bandwidth, investigating the behavior of cache models in gem5.

Specification		Intel Skylake	gem5 model
L1 Cache	Size	32 KiB	32 KiB
	Associativity	8	8
	MSHRs	10	10
	Access Latency	4 cycles	4 cycles
L2 Cache	Size	256 KiB	16 MiB
	Associativity	4	16
	Access Latency	12 cycles	40 cycles
L3 Cache	Size	16 MiB	N/A
	Associativity	16	N/A
	Access Latency	42	N/A
	GUPS	0.39	0.43

Table 5.2: Comparison of real hardware with gem model for GUPS Test.

The data presented in figure 5.9 shows that going down in the memory hierarchy will result in an increase in the latency and a decrease in the bandwidth, which is in line with our expectation of how a cache hierarchy influences the performance of memory subsystem.

5.3 GUPS Tests

In order to evaluate the whole memory subsystem consisting of caches and memory modules, we used GUPSGen in a system described by figure 5.10. As we previously mentioned, GUPS is a performance measure defined by the RandomAccess benchmark [20]. It exhibits a

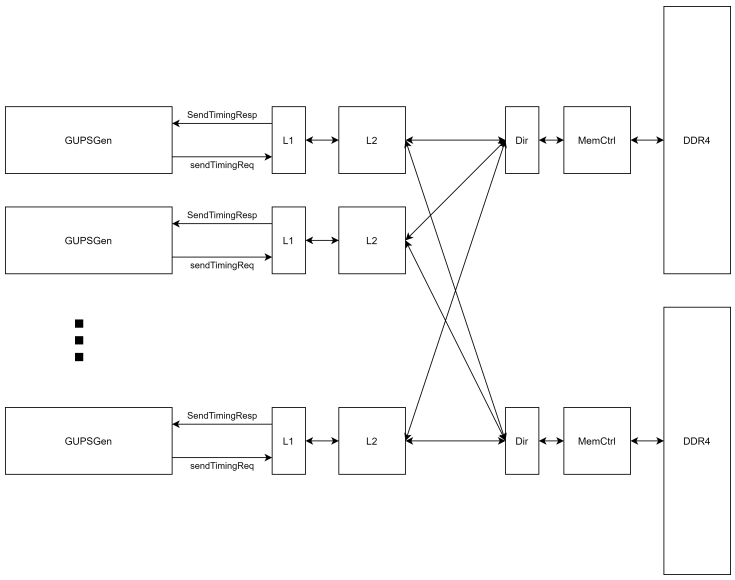


Figure 5.10: Diagram of the system used to test memory subsystem.

uniformly distributed pattern of memory accesses with both read and write operations. Due to the lack of locality in its accesses, the benchmark is a suitable test for pressuring the memory subsystem. In order to configure our caches, we used publicly available information on the Intel Skylake architecture [27]. Table 5.2 lays out the details and the comparison of GUPS measurements between the gem5 model and real hardware. We used the ruby cache implementation of the MOESI protocol to model our caches. However, gem5 currently does not support a 3 level cache hierarchy with the MOESI protocol; therefore, in order to model the L3 cache, we used an weighted an average of access latency from the L2 cache and L3 cache from the Skylake processor to configure the L2 cache in our model. The results show that while the real hardware running the GUPS benchmark in parallel mode achieves a 0.039 GUPS measurement, the configured model achieved 0.043 GUPS measurement [8].

Chapter 6

Conclusion and Future Work

In this work, we studied the accuracy of memory and cache models in gem5. We used a bottom up approach to set up systems that single out a component in the memory subsystem for testing. This approach enables us to validate each components separately. In our evaluation, we only considered the system effect of these components, and how different configurations of each model influences the performance at a higher level. In our experiments we did not encounter any unexpected behavior from any model. However, the memory controller model shows a 2× latency difference with our reference for comparison, we related this difference to an abstraction error resulted from the design of the memory controller. The memory controller does not implement queue structure in detail and lacks support for DRAM specific features such as bank groups. However, our further studies showed that this difference could be disregarded in regards to relative performance. Studies not targeting the memory design will not be affected by this difference as the difference will be factored out, since it is evenly applied to all cases, in the users' comparison. Moreover, we validated the cache models in the gem5 simulator by measuring their effect on memory subsystem performance using latency and bandwidth as metrics. Finally, we set up a complete memory subsystem based on publicly available information on an Intel Skylake architecture. Our results showed a 10% difference between the models in gem5 and real hardware on the GUPS measurements.

The major focus of this work is to validate the memory controller and DRAM modules in the memory subsystem, in future we will focus on validating cache models with further detail as listed below:

- Validate the Network on Chip topologies by comparing timing results between different topologies.
- Validate different coherency protocols by devising experiments that represent Single Producer, Multiple Consumer scenarios.
- Use validated designs to implement known good configurations for the memory subsystem.

Moreover, based on our measurements and observations we believe the following improvements could be done to increase the accuracy of gem5's components in the memory subsystem:

- Support could be added for per-bank and per-rank queue structures in the memory controller design. We believe implementing queue structures in detail could significantly improve the measured memory access latency. Moreover, the proposed improvement does not require significant change to the code base.
- Bank grouping feature could be fully implemented as it now a common feature among modern memory modules such as HBM, and GDDR. We believe this has considerable impact in latency measurements from memory modules specifically when using highly parallel programs or architectures. This change could be implemented by checking the number of bank group the request would map to when servicing and scheduling requests. Therefore, we do estimate that little change to the code base would accomplish this task.
- Modern coherency protocols along with common cache hierarchies could be implemented using SLICC. We believe this change makes it possible to better configure

memory subsystems that represent the real hardware. However, this change requires significant amount of change and time to be implemented correctly, both functionally and accurately.

The above list targets improvements in terms of the accuracy of the simulator. However, the simulator could also be improved in terms of usability and configurability. Below is a list of proposed improvements that could improve the usability of the gem5 simulator:

- Similar to SLICC, a domain specific language could be developed to describe the behavior of the memory controller. The goal of this improvement would be to allow the users to implement their designs by describing the memory controller as a state machine. This change would specifically benefit those who study the possibility of using DRAM as caches for NVM or using DRAM alongside NVM.

- The infrastructure for NoC simulation could be updated to support probing inside the NoC. At the time of writing this thesis, the links between the routers and controllers in the memory subsystem do not support attaching monitors. We believe adding this capability would allow users to gather more accurate information on memory requests such as compiling a trace of physicall memory addresses during full system simulations.

Bibliography

[1] Akram, Ayaz and Sawalha, Lina. Validation of the gem5 simulator for x86 architectures. In *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 53–58. IEEE, November 2019.

[2] Asanović, Krste and Patterson, David A. Instruction Sets Should Be Free: The Case For RISC-V. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, August 2014.

[3] Bojnordi, Mahdi Nazm and Ipek, Engin. PARDIS: A Programmable Memory Controller for the DDRx Interfacing Standards. *ACM SIGARCH Computer Architecture News*, 40(3):13–24, July 2012.

[4] Burger, Doug and Austin, Todd M. The SimpleScalar Tool Set, Version 2.0. *ACM SIGARCH Computer Architecture News*, 25(3):13–25, June 1997.

[5] Butko, Anastasiia and Garibotti, Rafael and Ost, Luciano and Sassatelli, Gilles. Accuracy Evaluation of GEM5 Simulator System. In *7th International Workshop on Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC)*, pages 1–7. IEEE, October 2012.

[6] Chandrasekar, Karthik and Weis, Christian and Li, Yonghui and Akesson, Benny and Wehn, Norbert and Goossens, Kees. DRAMPower: Open-source DRAM Power & Energy Estimation Tool, 2012.

[7] Chen, Ke and Li, Sheng and Muralimanohar, Naveen and Ahn, Jung Ho and Brockman, Jay B. and Jouppi, Norman P. CACTI-3DD: Architecture-level Modeling for 3D Die-stacked DRAM Main Memory. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 33–38. IEEE, April 2012.

[8] Cuma, Martin. *AMD Rome review*. Available on: https://www.chpc.utah.edu/documentation/white_papers/rome.pdf, Accessed on: 2021-07-30.

[9] Gutierrez, Anthony and Pusdesris, Joseph and Dreslinski, Ronald G. and Mudge, Trevor and Sudanthi, Chander and Emmons, Christopher D. and Hayenga, Mitchell and Paver, Nigel. Sources of Error in Full-system Simulation. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 13–22. IEEE, June 2014.

[10] Hansson, Andreas and Agarwal, Neha and Kolli, Aasheesh and Wenisch, Thomas and Udipi, Aniruddha N. Simulating DRAM controllers for future system architecture exploration. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 201–210. IEEE, June 2014.

[11] Hennessy, John L. and Patterson, David A. A New Golden Age for Computer Architecture. *Communications of the ACM*, 62(2):48–60, February 2019.

[12] Hughes, Christopher J. and Pai, Vijay S. and Ranganathan, Parthasarathy and Adve, Sarita V. Rsim: Simulating Shared-Memory Multiprocessors with ILP Processors. *Computer*, 35(2):40–49, August 2002.

[13] Jouppi, Norman P. and Young, Cliff and Patil, Nishant and Patterson, David and Agrawal, Gaurav and Bajwa, Raminder and Bates, Sarah and Bhatia, Suresh and Boden, Nan and Borchers, Al and others. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, June 2017.

[14] Jung, Matthias and Weis, Christian and Wehn, Norbert and Chandrasekar, Karthik. TLM Modelling of 3D Stacked Wide I/O DRAM Subsystems: A Virtual Platform for Memory Controller Design Space Exploration. In *Proceedings of the 2013 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, pages 1–6, January 2013.

[15] Kahng, Andrew B. and Srinivas, Vaishnav. Mobile System Considerations for SDRAM Interface Trends. In *International Workshop on System Level Interconnect Prediction*, pages 1–8. IEEE, January 2011.

[16] Kim, Yoongu and Yang, Weikun and Mutlu, Onur. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE Computer Architecture Letters*, 15(1):45–49, March 2015.

[17] Li, Man-Lap and Sasanka, Ruchira and Adve, Sarita V. and Chen, Yen-Kuang and Debes, Eric. The ALPBench Benchmark Suite for Complex Multimedia Applications. In *IEEE International Workshop/Symposium on Workload Characterization*, pages 34–45. IEEE, November 2005.

[18] Li, Shang and Yang, Zhiyuan and Reddy, Dhiraj and Srivastava, Ankur and Jacob, Bruce. DRAMsim3: A Cycle-Accurate, Thermal-Capable DRAM Simulator. *IEEE Computer Architecture Letters*, 19(2):106–109, February 2020.

[19] Lowe-Power, Jason and Ahmad, Abdul Mutaal and Akram, Ayaz and Alian, Mohammad and Amslinger, Rico and Andreozzi, Matteo and Armejach, Adrià and Asmussen, Nils and Beckmann, Brad and Bharadwaj, Srikant and others. The gem5 Simulator: Version 20.0+. *arXiv preprint arXiv:2007.03152*, September 2020.

[20] Luszczek, Piotr R. and Bailey, David H. and Dongarra, Jack J. and Kepner, Jeremy and Lucas, Robert F. and Rabenseifner, Rolf and Takahashi, Daisuke. The HPC Challenge (HPCC) benchmark suite. In *Proceedings of the 2006 ACM/IEEE Conference*

on Supercomputing, SC '06, page 213–es, New York, NY, USA, 2006. Association for Computing Machinery.

[21] Malladi, Krishna T. and Nothaft, Frank A. and Periyathambi, Karthika and Lee, Benjamin C. and Kozyrakis, Christos and Horowitz, Mark. Towards Energy-Proportional Datacenter Memory with Mobile DRAM. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 37–48. IEEE, July 2012.

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

[23] Press, Gill. *54 Predictions About The State Of Data In 2021.* Available on: https://www.forbes.com/sites/gilpress/2021/12/30/54-predictions-about-the-state-of-data-in-2021/, Accessed on: 2021-07-16.

[24] Rosenblum, Mendel and Bugnion, Edouard and Devine, Scott and Herrod, Stephen A. Using the SimOS Machine Simulator to Study Complex Computer Systems. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 7(1):78–103, January 1997.

[25] Steiner, Lukas and Jung, Matthias and Prado, Felipe S. and Bykov, Kirill and Wehn, Norbert. DRAMSys4. 0: A Fast and Cycle-Accurate SystemC/TLM-Based DRAM Simulator. In *International Conference on Embedded Computer Systems*, pages 110–126. Springer, October 2020.

[26] Stevens, Jim and Tschirhart, Paul and Chang, Mu-Tien and Bhati, Ishwar and Enns, Peter and Greensky, James and Chisti, Zeshan and Lu, Shih-Lien and Jacob, Bruce. AN INTEGRATED SIMULATION INFRASTRUCTURE FOR THE ENTIRE MEMORY HIERARCHY: CACHE, DRAM, NONVOLATILE MEMORY, AND DISK. *Intel Technology Journal*, 17(1):184–200, 2013.

[27] WikiChip. *Skylake (client) - Microarchitectures - Intel - WikiChip.* Available on: https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client), Accessed on: 2021-07-30.

[28] Woo, Steven Cameron and Ohara, Moriyoshi and Torrie, Evan and Singh, Jaswinder Pal and Gupta, Anoop. The SPLASH-2 Programs: Characterization and Methodological Considerations. *ACM SIGARCH Computer Architecture News*, 23(2):24–36, May 1995.