MemorySim: An RTL-level, timing accurate simulator model for the Chisel ecosystem

Ansh Chaurasia

ACM Reference Format:
Ansh Chaurasia. 2025. MemorySim: An RTL-level, timing accurate simulator model for the Chisel ecosystem . In . ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 ABSTRACT

AI applications have seen a surge in demand, leading to an uptick in attention towards AI chips. These applications have created a myriad of opportunities in hardware space. One area of opportunity for all hardware systems is the memory subsystem, which often, through bandwidth and performance, dictates system performance of high-demand applications like large language models (LLMs).

Currently there is a dire limitation in register-transfer level (RTL) memory subsystem models with support for obtaining profiling data. High-level memory models like DRAMSim2 [20] and DRAMSim3 [19] exist for simulations, but implementations like these often do not support both *timing* and *correctness*.

We introduce MemorySim, a RTL-level memory simulator that strives to provide accurate timing simulations of memory systems while retaining correctness. The simulator is designed to be integrated into existing RTL-level simulations written in Chisel or Verilog. It's also compatible with the Chisel [3] / Chipyard [18] ecosystem, allowing users to obtain highly accurate representations of performance and power through direct integration downstream leverage of simulation tools such as FireSim [21].

2 INTRODUCTION

The advent of the Transformer architecture has revolutionized sequence modeling by replacing recurrent and convolutional networks with self-attention mechanisms, enabling highly parallelizable models that achieve state-of-the-art results across a wide range of tasks [17]. Building upon this foundation, the paradigm of *foundation models*—large pretrained models adaptable to diverse downstream tasks—has emerged, exemplified by GPT-3's 175 billion-parameter few-shot capabilities [5] and the broad taxonomy of foundational systems spanning language, vision, and multimodal reasoning [4]. Scalability studies further characterize the emergent capabilities and performance trends of these models under increasing compute and data budgets [8].

Despite algorithmic innovations, the memory subsystem of modern accelerators often becomes the primary bottleneck in both training and inference of large language models (LLMs). Techniques such as the ZeRO optimizer stages mitigate memory redundancies and extend feasible model sizes by partitioning optimizer state, gradients, and parameters across devices [13]. Yet recent GPU-level analyses reveal that inference—especially under large batch sizes—remains predominantly memory-bound due to DRAM bandwidth saturation, hindering compute utilization even on high-end hardware [14].

During training, large-batch regimes and activation checkpointing for model parallelism elevate peak memory demands, driving complex combinations of data, tensor, and pipeline parallelism to avoid out-of-memory failures [10]. For inference, the quadratic complexity of self-attention and auto-regressive generation exacerbate cache and bandwidth pressures, prompting system-level optimizations such as kernel fusion and heterogeneous offloading [12]. These observations underscore the need for RTL-level memory simulation and profiling tools that can deliver accurate timing and correctness information to guide hardware-aware architecture exploration.

3 CURRENT LIMITATIONS

While high-level DRAM models such as DRAMSim2 [19] and DRAMSim3 [20] have become de facto tools for architectural exploration, they face a core triad of limitations when applied to AI-accelerator memory subsystem analysis:

(1) *Timing fidelity*: Both DRAMSim2 and DRAMSim3 use simplified cycle abstractions that approximate row-buffer hits, precharges, and refresh behavior, but omit critical microarchitectural effects—such as bank-slice contention and internal command reordering—which can skew effective bandwidth and latency estimates

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

under heavy, burst-oriented traffic common to transformer inference [9].

(2) *Resource constraints*: Without RTL-accurate description of resources, simulators struggle to capture limitations beyond a behavioral abstraction. Abstractions such as *backpressure*, and cycle based limitations cannot guarantee bit-true data integrity or capture corner-case hazards (e.g., tRRD violations under concurrent multi-bank accesses), making them unsuitable for early integration tests of new memory-centric accelerator designs [2].

(3) *Power–performance coupling*: Standalone power-estimation frameworks such as DRAMPower [15] and VAMPIRE [1] often rely on cycle-stack traces produced by DRAMSim2 or gem5; yet the loose coupling between timing and energy models precludes accurate estimation of dynamic power under throttling or command prioritization schemes optimized for AI workloads [16].

More cycle-accurate DRAM frameworks—Ramulator [9], USIMM [11], and DRAMSys [7]—offer enhanced configurability and support for emerging DDR standards, but remain difficult to embed within register-transfer-level (RTL) simulations. Their C++ incarnations introduce foreign-language interfacing overheads when integrated with Chisel or SystemVerilog based designs, resulting in slow co-simulation or the need for custom wrappers that further erode timing validity [6].

Consequently, there exists a pressing need for an *RTL-native* memory simulator that: (a) exposes fine-grained timing callbacks and decoupled command channels, (b) preserves data correctness via cycle-accurate handshake protocols, and (c) seamlessly interoperates with hardware generators and FPGA-accelerated emulation flows. By operating entirely within the Chisel [3] / Chipyard [18] ecosystem—leveraging the same FIRRTL intermediate representation and MIDAS-based back-ends [6, 21]—such a simulator can accurately profile bandwidth, latency, and power consumption without sacrificing integration ease or scale.

4 PROJECT TARGETS

Implement an RTL-level memory model in Chisel that can be accessed via Chipyard and FireSim emulations to estimate timing and power. As mentioned later in this paper, we plan to evaluate the fidelity of the model through rough comparisons with ideal memory systems.

5 SIMULATOR ARCHITECTURE

5.1 Top-Level Architecture

Figure 1 showcases a high level overview of the various system components, and potential avenues for the user to test the memory model in a standalone fashion. MemorySim

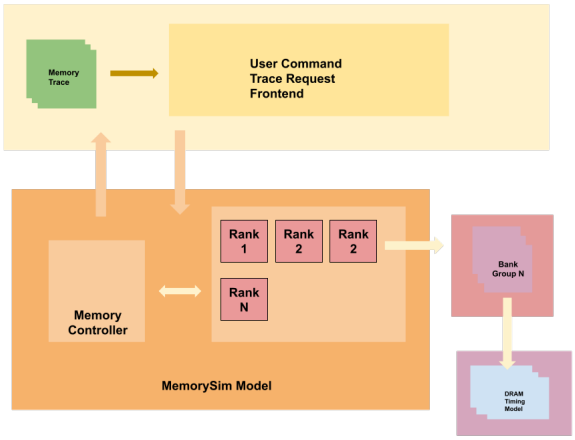

Figure 1: High-level architecture overview of the MemorySim simulation architecture.

was designed to be interoperable with existing Chisel hardware, while also providing useful value through standalone, trace-based simulations. We provide verified options for the module to be integrated into existing Chisel repositories, but focus on the standalone executable for isolated experimentation and understanding.

The core MemorySim architecture consists of the command front-end to receive standalone memory traces, which is received by a backend consisting of a memory controller and a memory channel. A typical path of a memory request would look like the following:

(1) Memory trace lists request $R = \{\textbf{addr}, t\}$, where t is the cycle at which the request must be issued.

(2) At cycle t, the request is enqueued into the global `reqQueue`. If the controller is not back-pressured, the request is dispatched in the very next cycle.

(3) The `MemoryController` classifies the request by rank and bank, forwards it into the appropriate Bank Scheduler's local queue, and begins the ACTIVATE – READ/WRITE – PRECHARGE handshake.

(4) Each Bank Scheduler enforces closed-page policy and refresh deadlines by driving the downstream DRAM Timing Model, which holds the request in timing-parameter states (e.g. t_{RCD}, t_{RP}, t_{RFC}).

(5) Upon completion, the Bank Scheduler emits a ⟨data/ack⟩ token back to the controller, which collects it in `respQueue` and returns a final acknowledgment to the frontend.

5.2 Bank Scheduler Design Architecture

A critical unit of the memory subsystem is the bank scheduler. This module interacts with the physical memory module. Each bank scheduler is designed to operate and manage a

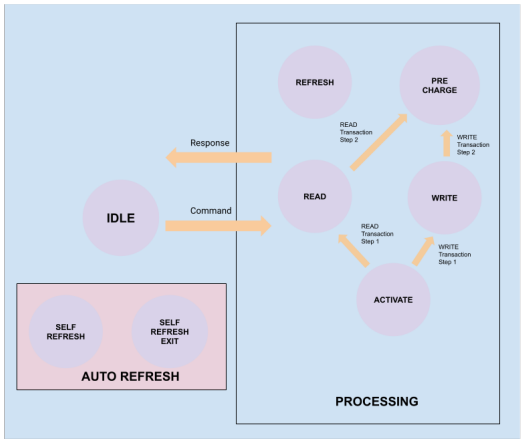

Figure 2: A diagram of finite-state-machine (FSM) orchestration of the bank scheduler.

bank within the memory unit. Addresses are directly mapped to banks via a fixed address mapping. The address mapping scheme is shown below:

address ⟵ {**remainingbits**, **rankidx**, **bankgroupidx**, **bankidx**}

By extracting the the **rank**, **bankgroup**, and **bank** bits, we can map an address to a specific DRAM bank scheduler.

Our DRAM Bank follows a *closed-page* policy, which means every request, irrespective of its successors, are preceded by `ACTIVATE` and succeeded by `PRECHARGE`. As will be noted in our results in subsection 8.1, we discover that the ideal reference simulator *always* uses an open page policy (despite attempts to change the configuration to use a closed policy), contributing to the total steady state cycle-difference.

Figure 2 shows a model of the various states employed by the closed-page bank finite-state machine at a high level. The states are divided into `IDLE`, `PROCESSING`, and self refresh related states used to park the DRAM bank during `IDLE` activity. The life-cycle transition paths for the scheduler are listed below.

5.2.1 *READ REQUEST.* :

(1) Request initially is in `IDLE`
(2) If not within the refresh deadline, the scheduler issues an `ACTIVATE` request, powering on the DRAM row
(3) After an `ACTIVATE` completion acknowledgment (`ACTIVATE`-ack) is returned from the DRAM, the scheduler issues a READ request.
(4) After the `READ`-ack is returned, the scheduler issues a `PRECHARGE` command
(5) Post `PRECHARGE`-ack, scheduler returns to IDLE, awaiting the next request.

5.2.2 *WRITE REQUEST.* :

(1) Request initially is in IDLE.
(2) If not within the refresh deadline, the scheduler issues an ACTIVATE request, powering on the DRAM row
(3) After an `ACTIVATE` completion acknowledgment (`ACTIVATE`-ack) is returned from the DRAM, the scheduler issues a WRITE request.
(4) After the `WRITE`-ack is returned, the scheduler issues a `PRECHARGE` command
(5) Post `PRECHARGE`-ack, scheduler returns to IDLE, awaiting the next request.

5.2.3 *REFRESH REQUEST.* :

(1) Request initially is in `IDLE`
(2) If within the refresh deadline, a REFRESH request is issued.
(3) After a `REFRESH`-ack is returned from the DRAM, the scheduler returns to IDLE.

The refresh request flow also supports self refreshes - if the bank has been idle for 1,000 cycles, a self refresh request is issued, and upon an acknowledgement, moves into the `SREF ENTER` state. Upon a valid request arriving, it remains in that state. It then issues an `SREF EXIT` command to exit self refresh mode, and processes the next request.

5.3 Memory Controller Architecture Design

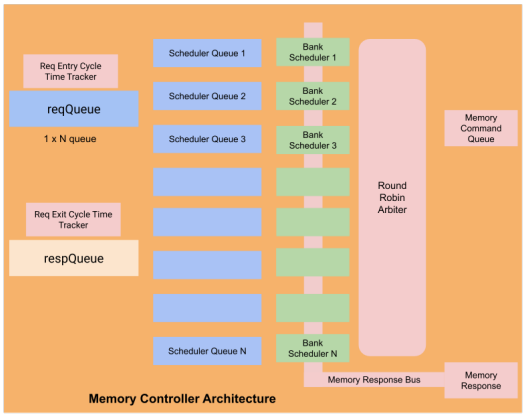

Figure 3: Diagram of the memory controller at a high level. The reqQueue data structure comprises of all the blue strips, and splits incoming requests into the respective scheduler queues. Once the scheduler receives responses from physical memory, the data is carried forth towards the response queue.

The memory controller is designed to issues requests with parallelism at the bank level. As shown in Figure 3, requests

enter through a request queue *reqQueue*, and are issued back to the user through a response queue *respQueue*.

Intermediate memory commands from bank schedulers are sent from the bank scheduler to a round robin arbiter, which collects the requests, and enqueues them into the memory command queue. Responses from the physical memory channel are sent back to the controller, which broadcasts the response to all bank schedulers. The bank scheduler who's address mapping and request-id meet the response's will accept the request.

5.4 Physical Memory Model Architecture

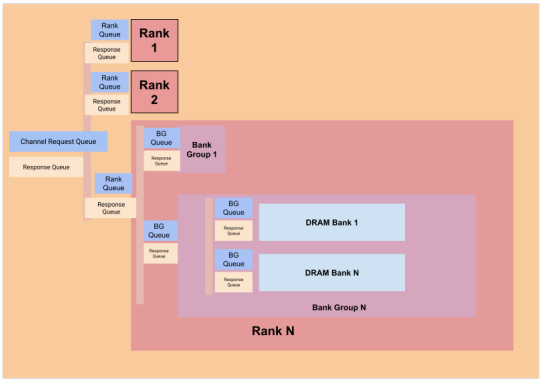

Figure 4: Top-level diagram of the memory hierarchy, which goes from (1) channels, (2) to ranks, (3) to bank groups, (4) to banks. The red bars denote RR arbiters, with the inputs solely being the light response queues. The responses are coalesced into the the broader response queue. No request queues interact with the round-robin arbitration.

The controller interacts with a memory channel, a diagram of which is shown above in Figure 4. Each level of the hierarchy is separated by request and round-robin arbitrated response queues. The request queues leverage a similar multi-dequeue data structure to that discussed in the memory controller design. Functionally, the channel, rank, and bank group are indistinguishable, representing layers of the memory hierarchy. Our DRAM timing model breaks this tradition, representing the fundamental unit of memory storage, and as such requires more complex logic.

5.5 DRAM Timing Model

The DRAM model bears many similarities, and can thought of as a mirror to the implementation of bank-scheduler FSM.

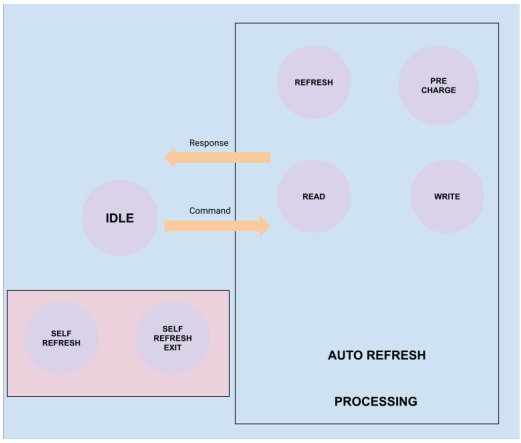

Figure 5: Diagram of the timing model used for the DRAM memory bank. The top-level diagram, although similar to that of the bank scheduler is different in function - no processing states are explicitly linked (i.e READ comes after PRECHARGE), rather these states are entered through control from the bank states.

This is by design. However, it is designed to be controlled by the bank scheduler and track row information, rather than make significant independent logic. Given a set of timing parameters, the model enters a state of `READ`, `WRITE`, `REFRESH`, `PRECHARGE`, `SELF REFRESH`, `SELF REFRESH EXIT`, waits for duration specified by a timing parameter, and then issues a response.

A list of timing parameters that are supported have been listed here - Table 1 .

6 EXPERIMENTATION METHODOLOGY

6.1 Introduction to DRAMSim3

DRAMSim3 is a cycle-accurate, thermal-capable DRAM simulator supporting a wide range of DRAM protocols—including DDR3, DDR4, LPDDR3, LPDDR4, GDDR5, GDDR6, HBM, HMC, and emerging non-volatile memories—designed for both trace-driven and full-system simulation environments. It is implemented in modern C++ as an object-oriented framework, featuring parameterized DRAM bank models, modular memory controllers, command queues, and flexible system-level interfaces for integration with CPU simulators such as gem5 and ZSim or for standalone trace workloads. The core simulation kernel is a discrete event-driven scheduler that processes DRAM commands (`ACTIVATE`, `READ`, `WRITE`, `PRECHARGE`, `REFRESH`) according to timing parameters derived from JEDEC standards, ensuring cycle-accurate fidelity.

Parameter Name	Stage	Purpose	Value
tRP	PRECHARGE	Total time it takes for a row precharge before activation	14
tFAW	ACTIVATE	Four activation window length – capped at 4 activations within this duration	30
tRRDL	ACTIVATE	Minimum cycles between two consecutive activates	6
tRCDRD	READ	Delay from activate to read	14
tCCDL	READ	Gap between consecutive reads	2
tWTR	READ	Turnaround time between write and subsequent read	8
tRP	READ	Time since last precharge	14
tRCDWR	WRITE	Delay from activate to write	14
tCCDL	WRITE	Gap between consecutive writes (depends on previous op)	2
tRP	WRITE	Time from precharge to write	14
tRFC	REFRESH	Deadline to start a refresh	260
tREFI	REFRESH	Interval between periodic refreshes	3600

Table 1: Timing parameters implemented by the bank model.

DRAMSim3 optionally incorporates thermal and power modeling by coupling its event traces with energy-estimation engines, enabling detailed power–performance trade-off studies under throttling or scheduling policies optimized for AI workloads. The simulator is engineered for portability and parallelism: it compiles on Linux, macOS, and Windows, and it can leverage multiple threads to accelerate large trace-driven analyses while preserving accuracy.

6.2 DRAMSim3 Key Features

- **Modular Hierarchy**: Channels, ranks, bank groups, and banks are each represented by dedicated C++ classes, with timing parameters (e.g., `tRCD`, `tRP`, `tRAS`) loaded from JSON configuration files to match specific DRAM standards.
- **Command Scheduling**: The built-in memory controller supports configurable scheduling policies (e.g., FCFS, FRFCFS), with per-bank command queues and arbitration logic to model realistic contention and bank-level parallelism.
- **Trace and Full-System Modes**: Users can drive DRAMSim3 with simple address-trace files (request cycle, address, read/write opcode) or via hot-plug interfaces to full-system simulators, enabling evaluation from microbenchmarks to complex OS workloads.
- **Refresh and Self-Refresh**: Periodic and on-idle refresh operations are modeled according to user-specified `tREFI` and `tRFC` parameters, including power-down self-refresh modes for low-power analysis.
- **Parallel Execution**: A thread-pool approach partitions large trace files across worker threads, reducing wall-clock simulation time while maintaining correct event ordering.

7 TARGET BENCHMARKS

These microbenchmarks were created to generate real-world memory access traces (via Valgrind) for evaluating MemorySim's timing accuracy and correctness under representative workloads.

- `conv2d.c`: A small 2D convolution kernel to evaluate spatial locality and burst access patterns in sliding-window operations.
- `multihead_attention.c`: A toy multi-head attention workload that exercises dot-product computes and softmax-induced memory reuse typical of transformer models.
- `trace_example.c`: A minimal read/write trace generator used to validate basic request sequencing and correct data return in the simulator.
- `vector_similarity.c`: A cosine-similarity search across a small vector database to test irregular access patterns and reduction operations.

8 RESULTS

8.1 Overview

As one may reasonably expect, the RTL-level model consumes more cycles per read or write operation. On average for traces simulations runs set to 100, 000 cycles, the read-operation consumes $\frac{102+114+117+110}{4} \approx 111$ cycles, while write

Benchmark Name	Read Diff Avg	Read StdDev	Write Diff Avg	Write StdDev
conv2d.c	102	59	171	154
multihead-attention.c	114	67	110	38
trace-example.c	117	70	111	38
exp-vector-similarity.c	110	66	109	38

Table 2: Average cycle differences in read/write requests between DRAMSim3 (ideal software simulator) and MemorySim. Values computed as MemSimCycles – DRAMSimCycles. These results are obtained by setting the queueSize parameter to 128.

operations take $\frac{171+110+111+109}{4} \approx 125$ cycles. This data, listed in Table 2, alongside the standard deviations, suggest large room for improvement in the precision of the accelerator. An important characteristic to keep in mind here is the parameter **queueSize**, which controls the depth of all queues within the controller system. As we will characterize in time, toggling the queue depth significantly alters the read and write difference averages, bringing the differences down to the order of 50 and 80 cycles respectively, also reducing the standard deviation to 50 and 10 cycles respectively. To sum it up, the simulator's memory subsystem is highly sensitive to *backpressure* from **reqQueue**, which largely increases as **queueSize** rises.

For further analysis, we demonstrate how critical of an impact back-pressure can play (1) the mean cycles MemorySim takes over different cycle periods, and (2) varying the queue size by powers of two and observing the cycle differences.

We also acknowledge feature-based limitations, which we've listed below:

- **OPEN PAGE vs CLOSED PAGE policies**: There appears to be some systematic bugs in DRAMSim3 software, where schedulers would issue only READ requests, even after being placed in closed page policy, which would require the model to issue pre-charges after READ/WRITE requests to close rows, along with activates to open the row. In other words, DRAMSim3 enacted a default OPEN PAGE policy, in comparison to our CLOSED PAGE policy.
- **Lack of caching / request re-ordering**: The existing implementation currently doesn't support caching rows, which is implemented in the ideal simulator for read / write operations to bypass scheduling. As discussed in our section on next steps, these are set to be part of a later host of features to be released.

9 ABLATIONS

9.1 Cycles over Various Epochs

In order to understand how the simulator is reacting to the given benchmark traces and whether peak traffic had an impact overall, we plot the average latency vs the input cycle

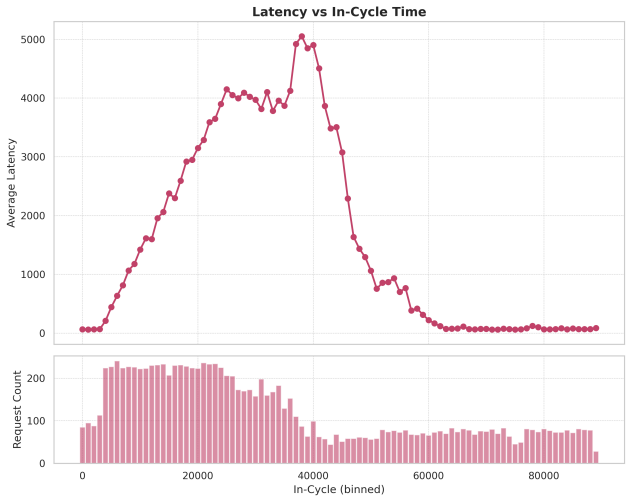

Figure 6: Simulator average latency profile, with a window of 1000 cycles. This was taken on the conv2d benchmark.

range, i.e, the average latency of all the requests within 1000 cycle bins. It's noticeable that request latency initially, for the first 500 cycles, remains stable. However, as the number of cycles progress and the system faces a higher number of traffic requests, we see a rise in latency. This shows that for some initial state, the system is capable of performing, but processively is unable to tackle the load.

9.2 Latency vs Queue Size

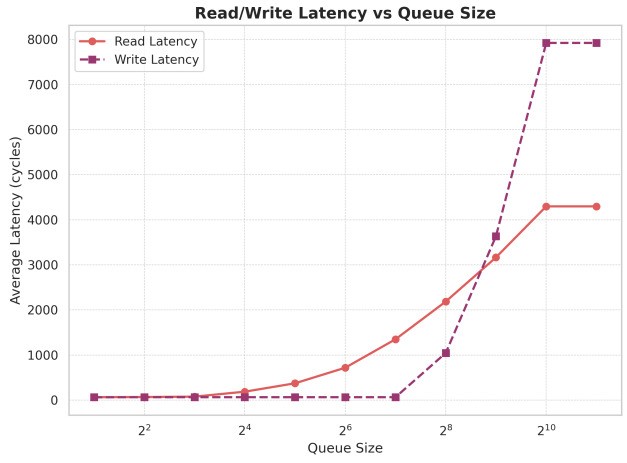

Figure 7: Read and write latency curves vs. queue size, performed on the conv2d benchmark.

As demonstrated previously in Figure 6, traffic seems to periodically increase. That leads us to check **queueSize**, the

parameter that checks the size of **reqQueue** (see Figure 3 for a reminder). We vary **queueSize** between 2 and 1024 to examine its impact on latency. It turns out, as shown in Figure 7, that the latency does indeed scale with request queue size, exponentially. Decreasing the request queue size brings the latency significantly lower, while extended peak traffic activity on larger queues lead to overall larger latency.

9.3 Simulator Cycle Breakdown

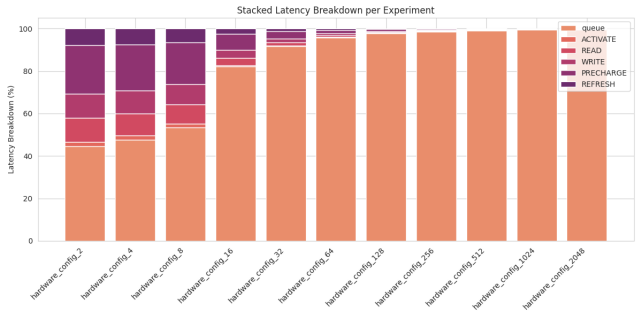

Figure 8: Breakdown of the total latencies by percentage on the `conv2d` benchmark. We vary the queue size of the controller, systematically increasing from 2 to 2048 in size.

To prove the statement that "request queue size is the single-most critical factor that determines average latency in the system", we perform a further breakdown of the average cycle latencies into their average constituents. Figure 8 yields an explanation into the phenomenon - Higher **queueSize** leads to degraded performance caused by increased backpressure on *reqQueue*. As the size of the queues scale, the backpressure becomes all-consuming, accounting for $\approx 100\%$ of the cycle delay.

9.4 Tradeoff: Latency vs. Number of Requests

Based on the previous ablation the solution appears simple - just reduce **queueSize**! Unfortunately, this fails to take into account another problem - requests that don't enter the system become *blocking* for all future requests. Consider the case where multiple bank schedulers are operating at peak capacity (full individual queues, dictated by a smaller queue size). Due to a small queue size, backpressure on the larger, request-facing queue can reach a state with requests in the queue are all backfill for a bank scheduler already operating at full capacity. As a result, all the other schedulers are starved of requests, leading to lower number of requests.

Figure 9 demonstrates the Pareto tradeoff between the number of requests served on the `conv2d` benchmark, vs. latency. Notably, the lowest average latency systems face this

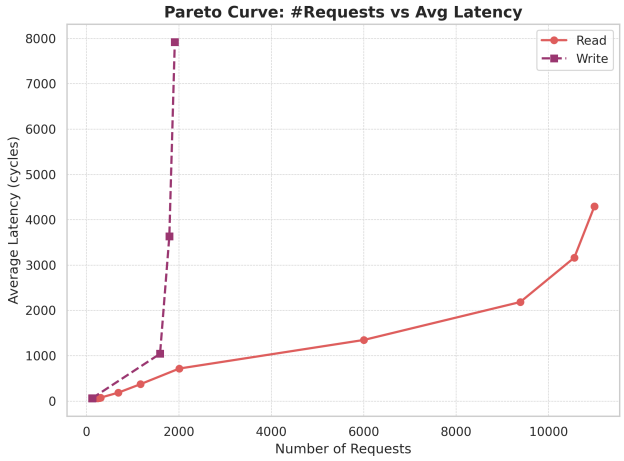

Figure 9: Pareto curve of the number of requests completed and the average latency. Ideally, we want to be at an operating point where we have high numbers of requests, and low average latency.

"starvation" problem more often, leading to lower throughput. On the other hand, higher throughput systems are more capable of unblocking themselves from this problem, at the cost at higher perceived average latency from **reqQueue** backpressure.

10 CONCLUSION

In this work, we have introduced *MemorySim*, an RTL-native DRAM simulator fully embedded within the Chisel/Chipyard ecosystem. By implementing bank-level FSMs, a centralized `reqQueue` with multiple dequeue support, and a cycle-accurate DRAM timing model entirely in hardware, MemorySim eliminates synthesis challenges with high level behavioural memory models.

Our quantitative evaluation against DRAMSim3 reveals a consistent cycle-count overhead across four AI-accelerator microbenchmarks. As shown in Table 2, MemorySim incurs an average read-cycle penalty of 111 cycles and a write-cycle penalty of 125 cycles over 100,000 cycle traces. Notably, `conv2d.c` exhibits the highest write overhead (171 cycles), while `multihead_attention.c` shows the lowest write variance (StdDev 38) (Table 2).

A detailed latency profiling (Figure 6) demonstrates that average request latency remains near 110 cycles for the first 500 cycles, but climbs to over 200 cycles under sustained traffic bursts—highlighting sensitivity to transient backpressure. Varying the `queueSize` parameter between 2 and 1024 (Figure 7) yields an exponential latency increase: a queue size of 2 achieves sub-80 cycle average latency, whereas a size of 1024 pushes latency beyond 250 cycles.

Figure 8 breaks down the sources of latency and confirms that backpressure in the centralized `reqQueue` accounts for up to 100% of additional cycles as queue depth increases. The Pareto trade-off between throughput and latency (Figure 9) further illustrates that minimizing `queueSize` reduces latency but can starve bank schedulers—dropping completed requests from over 10,000 to under 6,000 on the `conv2d` benchmark.

In summary, MemorySim delivers a high-fidelity, modular platform for RTL-level memory subsystem exploration. By correlating specific latency overheads with system parameters (Table 2, Figure 7), designers can quantitatively evaluate and tune memory-controller policies. Future work will extend these findings with dynamic backpressure controls, per-bank read caching, and rank-level self-refresh optimizations to further close the performance gap to software models while retaining hardware correctness.

11 CODE

All reference code used can be found at this GitHub repository: https://github.com/AnshKetchum/hbm-controller.

12 ACKNOWLEDGEMENTS

We thank Professor Chistopher Fletcher from UC Berkeley and Ph.D student Tianrui Wei for their generous, insightful counsel.

REFERENCES

[1] Hadi Ardakani, Yuke Yang, Guangyu Dong, James Drake, and Michael J. Irwin. 2017. VAMPIRE: A DRAM Power and Performance Modeling Framework. In *Proceedings of the 2017 IEEE International Symposium on Performance Analysis of Systems and Software*. 1–12.

[2] Klaus Arnold, Jin-Soo Jeong, and David Chau. 2015. Validating DRAM Timing and Functionality in an FPGA-Based Emulation Platform. In *Proceedings of the 2015 IEEE International Symposium on Performance Analysis of Systems and Software*. 15–25.

[3] Jonathan Bachrach, Jonathan Vo, Charles Richards, Andrew Lee, Andrew Waterman, David Patterson, and Krste Asanović. 2012. Chisel: Constructing Hardware in a Scala Embedded Language. In *Proceedings of the 49th Annual Design Automation Conference*. 1216–1225.

[4] Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, et al. 2021. On the Opportunities and Risks of Foundation Models. *arXiv preprint arXiv:2108.07258* (2021). https://arxiv.org/abs/2108.07258

[5] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language Models are Few-Shot Learners. *Advances in Neural Information Processing Systems (NeurIPS)* 33 (2020), 1877–1901. https://arxiv.org/abs/2005.14165

[6] Al Harrison, Maria Vegdahl, Nayan Parihar, et al. 2020. MIDAS: FPGA-Accelerated Full-System Simulation at Near-Native Speed. In *Proceedings of the 2020 IEEE International Symposium on Performance Analysis of Systems and Software*. 1–12.

[7] William Kai, Wim Heirman, and Dirk Stroobandt. 2018. DRAMSys: Flexible and Extensible Open-Source Memory Subsystem Modeling in SystemC. In *Proceedings of the 2018 IEEE International Symposium on Performance Analysis of Systems and Software*. 1–11.

[8] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling Laws for Neural Language Models. *arXiv preprint arXiv:2001.08361* (2020). https://arxiv.org/abs/2001.08361

[9] Yunji Kim, Jayadev Das, Onur Mutlu, and Yale N. Patt. 2015. Ramulator: A Fast and Extensible DRAM Simulator. In *IEEE Computer Architecture Letters*, Vol. 15. 45–49.

[10] Joyjit Kundu, Wenzhe Guo, Ali BanaGozar, Udari De Alwis, Sourav Sengupta, Puneet Gupta, and Arindam Mallik. 2024. Performance Modeling and Workload Analysis of Distributed Large Language Model Training and Inference. *arXiv preprint arXiv:2407.14645* (2024). https://arxiv.org/abs/2407.14645

[11] Jakob Mosbeck, Guillaume Helion, Robert Dreslinski, and et al. 2015. USIMM: A Stateful DRAM System Simulation Infrastructure. In *Proceedings of the 2015 IEEE International Symposium on Performance Analysis of Systems and Software*. 1–10.

[12] Xuefei Ning, Yu Wang, Guohao Dai, Jie Xu, X.-P. Zhang, Yuhan Dong, Zulin Yuan, Shujie Yan, and Xiaoyang Li. 2024. A Survey on Efficient Inference for Large Language Models. *arXiv preprint arXiv:2404.14294* (2024). https://arxiv.org/abs/2404.14294

[13] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2019. ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. *arXiv preprint arXiv:1910.02054* (2019). https://arxiv.org/abs/1910.02054

[14] Pol G. Recasens, Ferran Agullo, Yue Zhu, Chen Wang, Eun Kyung Lee, Olivier Tardieu, Jordi Torres, and Josep Ll. Berral. 2025. Mind the Memory Gap: Unveiling GPU Bottlenecks in Large-Batch LLM Inference. *arXiv preprint arXiv:2503.08311* (2025). https://arxiv.org/abs/2503.08311

[15] Wim Schellekens, Sam Swinnen, and Dirk Stroobandt. 2014. DRAMPower: Open-Source DRAM Power Energy Estimation Tool. In *Proceedings of the 2014 IEEE International Symposium on Performance Analysis of Systems and Software*. 1–12.

[16] Santhosh Shevgoor, Onur Mutlu, and Alok Choudhary. 2016. Understanding the Impact of DRAM Latency on Memory System Performance. In *Proceedings of the 2016 ACM International Conference on Measurement and Modeling of Computer Systems*. 390–401.

[17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. In *Advances in Neural Information Processing Systems (NeurIPS)*. 5998–6008. https://arxiv.org/abs/1706.03762

[18] Andrew Waterman, Krste Asanović, David Patterson, and Jonathan Bachrach. 2020. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *arXiv preprint arXiv:2001.00046* (2020).

[19] Henry Weber, Bavani Weber, and Per Stenstrom. 2010. DRAMSim2: A Cycle Accurate Memory System Simulator. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. 2–13.

[20] Chen Yu, Aparna Chandrasekar, Vivek Seshadri, et al. 2021. DRAMSim3: A Memory System Simulator. *arXiv preprint arXiv:2102.05607* (2021).

[21] Andrew Zhang, Jacob Izraelevitz, Onur Guney, et al. 2020. FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud. In *Proceedings of the 2020 USENIX Annual Technical Conference*. 1–15.