2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) | 978-1-6654-5954-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/ISPASS55109.2022.00045

DRAM Bandwidth and Latency Stacks: Visualizing DRAM Bottlenecks

Stijn Eyerman
Intel Corporation
Belgium
stijn.eyerman@intel.com

Wim Heirman
Intel Corporation
Belgium
wim.heirman@intel.com

Ibrahim Hur
Intel Corporation
USA
ibrahim.hur@intel.com

***Abstract*—For memory-bound applications, memory bandwidth utilization and memory access latency determine performance. DRAM specifications mention the maximum peak bandwidth and uncontended read latency, but this number is never achieved in practice. Many factors impact the actually achieved bandwidth, and it is often not obvious to hardware architects or software developers how higher bandwidth usage, and thus higher performance, can be achieved. Similarly, latency is impacted by numerous technology constraints and queueing in the memory controller.**

DRAM bandwidth stacks intuitively visualize the memory bandwidth consumption of an application and indicate where potential bandwidth is lost. The top of the stack is the peak bandwidth, while the bottom component shows the actually achieved bandwidth. The other components show how much bandwidth is wasted on DRAM refresh, precharge and activate commands, or because of (parts of) the DRAM chip being idle when there are no memory operations available. DRAM latency stacks show the average latency of a memory read operation, divided into base read time, row conflict, and multiple queue components. DRAM bandwidth and latency stacks are complementary to CPI stacks and speedup stacks, providing additional insight to optimize the performance of an application or to improve the hardware.

I. Introduction

Memory bandwidth is one of the major performance parameters of a processor, next to core count, chip frequency and cache size. The performance of memory-bound applications is often determined by how much bandwidth they can use—think of the slope at the left in the roofline model [19]. Memory specifications refer to the peak bandwidth a certain memory chip can obtain; for example, a common DDR4-2400 module has a peak bandwidth of 19.2 GB/s (2400 MT/s $\times$ 8 B memory channel width). However, this bandwidth assumes perfect circumstances and cannot be achieved in practice. Because performance is determined by bandwidth usage, it is helpful to determine the causes of suboptimal bandwidth usage and how they can be resolved.

Although memory latency is often a secondary parameter, it is equally important for performance. The processor cores and memory form a closed loop: cores generate requests to memory, and will eventually stall and not generate new requests until the access is done. If the access time is too high, the cores will not be able to generate enough requests to fully utilize the available bandwidth. On the other hand, the closer bandwidth usage comes to the peak bandwidth, the more queueing latency will be added to the access latency, again limiting the request rate.

We propose *bandwidth and latency stacks* as a way to intuitively visualize the bandwidth usage, memory latency and bottlenecks of an application. A stacked representation is an established way to represent performance bottlenecks, e.g., CPI stacks [10] and speedup stacks [9]. The top of the *bandwidth stack* represents the peak bandwidth. The bottom component shows the achieved bandwidth (split into read and write bandwidth), meaning that the rest of the stack represents the "lost" bandwidth. This bandwidth loss can have many causes: DRAM refresh cycles block the full chip, a page miss in a bank causes a precharge-activate cycle, or timing restrictions at the channel, rank and bank level delay an operation. It might also be that the core(s) do not provide enough memory requests to saturate the bandwidth, or that the addresses requested are not uniformly distributed across the parallel memory banks. Each of these causes is represented as a component in the stack, sized according to its impact on bandwidth loss.

The top of the *latency stack* equals the average latency of a read request. The bottom component equals the minimal read time in optimal circumstances: no queueing and an access to an already open page. The other components quantify the extra latency due to page misses, write bursts and queueing.

Measuring bandwidth and latency stack components and sizing them meaningfully is not evident: memory specifications contain dozens of timing restrictions at multiple levels (channel, rank, bank group, bank) and the organization is highly parallel. Operations occur in parallel and timing restrictions overlap, making it hard to unambiguously account a cause to suboptimal memory usage. We clarify in this paper how to construct bandwidth and latency stacks in a meaningful and useful way. We also evaluate them using simulation and show how they can be used to analyze and optimize memory usage and performance. Additionally, we show how bandwidth stacks can be used to more accurately extrapolate bandwidth usage when scaling up the core count.

We begin by discussing related work and a high-level overview of how DRAM is organized. Next, we discuss the construction of bandwidth and latency stacks. After explaining our experimental setup, we validate the intuitiveness and meaningfulness of the newly proposed stacks using synthetic

978-1-6654-5954-9/22/$31.00 ©2022 IEEE
DOI 10.1109/ISPASS55109.2022.00045

benchmarks. We then collect stacks for the graph-oriented GAP benchmarks, and show how they contribute to performance analysis and extrapolation, followed by our conclusions.

II. Related Work

Performance analysis of processor systems is challenging, because of the complexity of (out-of-order) cores, on-chip network and memory, and the interplay of these components. Performance stacks are a visually insightful way to represent the components that impact performance. For example, a CPI or cycle stack [10] represents all execution cycles of an application, and its components show how much time is spent in active execution cycles, waiting for memory operations to finish, resolving mispredicted branches, etc. FLOPS stacks [11] identify, for compute-bound applications, why the maximum floating-point operation throughput of a machine is not achieved due to memory, limited instruction-level parallelism, insufficient vectorization, etc. Speedup stacks [9] show the ideal speedup of a parallel application (equal to thread count), and its components reflect the causes why this ideal speedup is not reached: sequential parts, synchronization, conflicts in cache and network, etc.

A stacked representation is in most cases a simplified view of performance. It suggests that the performance limiting components can be isolated and added. In reality, many events occur in parallel and have an impact on each other. As an example, cycle stacks can be measured at different stages in the processor [8], resulting in different stacks that are all representing the same application. Stacks are a first order indication, and are popular because of their intuitiveness.

The roofline model [15], [19] is an intuitive model to characterize a system and analyze performance. It is a graph showing compute intensity (operations per byte) on the X-axis and performance on the Y-axis. An application whose performance is beneath that of the roofline model and that is memory intensive (left slope of the roofline), does not reach the peak memory bandwidth usage and has suboptimal performance. Increasing bandwidth usage increases performance for these applications.

Performance profiling tools, such as Intel VTune [5], can detect application phases where bandwidth usage is low and/or memory latency is high. Eklov et al. [7] propose "Bandwidth Bandit", a tool to quantify how sensitive an application is to bandwidth contention. Xu et al. [20] use machine learning to detect whether an application or specific data structure suffers from (remote) bandwidth contention. Helm and Taura [13] developed a hardware profiling tool that measures memory latency and show that the relative latency compared to uncontended latency is a better indicator for bandwidth contention than bandwidth usage. These tools profile and quantify bandwidth contention and the corresponding latency increase, but do not attribute the contention and delay to the DRAM specific root causes, which is the goal of bandwidth and latency stacks. Cho et al. [4] define data bus busy time, bank busy time and inter-bank interference time as metrics to analyze DRAM performance.

Processor and memory simulation [2], [3], [17], [18] is a commonly used tool to determine and analyze performance. Simulation can model future designs that are not yet available, and it enables detailed performance analysis because each internal event can be monitored. A memory simulator can produce a trace of DRAM commands, which can be used to construct bandwidth stacks. Well-known memory simulators are DRAMSim [18] and Ramulator [17]. DRAMSim3 includes a visualization tool to plot bandwidth usage, latency and power consumption over time, without measuring the contributions of the different components like in the proposed bandwidth and latency stacks. Alternatively, command traces can be collected during program execution on the hardware, e.g., by inserting an (FPGA) profiler between the memory controller and the memory chip [14].

III. Memory Organization and Timing Constraints

Modern DRAM memories (e.g., DDR4 [16]) are organized hierarchically. The *channel* is the connection between the DRAM memory and the memory controller, located on the processor chip. It comprises the command bus and the data bus. A channel is connected to one or more *ranks*, which are independent memory packages. Multiple ranks per channel increase the channel usage and achieved bandwidth, but also complicates the timing of commands.

A rank consists of a number of *banks*. Banks operate in parallel to increase the performance and peak bandwidth. Banks consist of multiple SDRAM chips that operate together to provide one row of data to a page buffer (8 KB is a common size), from which data is read and transferred to the memory controller. The growing number of banks per rank, to increase bandwidth, has led to an intermediate level, the *bank group*. Requests to the same bank group have more timing restrictions than accesses to different bank groups.

DRAM is accessed by the memory controller by sending DRAM commands along the command bus. Read and write operations from the processor are translated to commands by the memory controller. If a read operation is to a bank that has the matching row open in the page buffer, the memory controller just needs to issue a read command. However, if the wrong row is in the page buffer, a precharge command should be sent first to write back the current row and precharge the bitlines. Then an activate command loads the row into the page buffer, followed by a read command to obtain the requested data. Similar commands are required for write operations. Because the capacitors that store the data are leaking, rows need to be regularly refreshed by issuing a refresh command, during which the chip is inaccessible.

Commands can only be issued when strict timing requirements are met. These timings are documented in the vendor provided specifications of the chip. For example, it takes a while to load a row into the page buffer on an activate command, so the read command to that row should not be issued before the activate is ready. Refresh should happen at

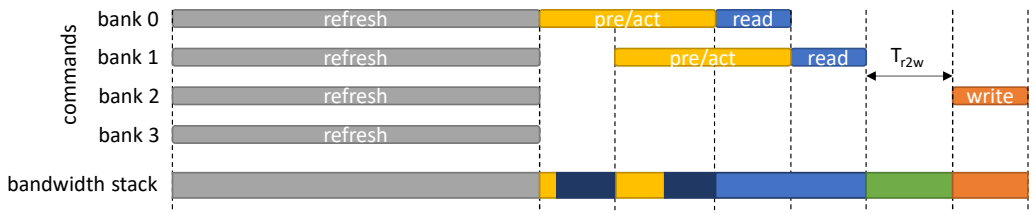

Fig. 1. Bandwidth stack accounting example. Commands for four banks and resulting stack components. The read/write operation is the time the channel is occupied to transfer the data, these cannot be done in parallel. Precharge/activate can be done in parallel in the banks, which is why we add bank-idle (dark blue) components for the idle banks. T_{r2w} is the read to write delay, generating a constraints component.

prescribed times to avoid loss of data, and no other commands can be issued during refresh. Read commands require an address, and the data is put on the channel after some time, while for write commands, the data is put on the channel together with the command. This induces extra overhead when switching between read and write commands. Writes are buffered in the memory controller until there are idle cycles or the buffer fills up. At that point, the write buffer is drained, performing a burst of writes to minimize the read-to-write overhead. There are dozens of timing restrictions, which makes an intuitive analysis of DRAM bandwidth and latency not straightforward.

IV. Bandwidth Stack Accounting Mechanism

We construct one stack per memory controller/channel, which can be aggregated afterwards. The ideal case, achieving peak bandwidth, is when the channel is completely busy, i.e., sending data at its maximum rate. Therefore, we account useful cycles when data is sent across the channel, either data that is read from memory or data that needs to be written to memory. On cycles where no data is transferred, we need to figure out why no data is sent, and account these cycles to the respective loss component. It is important to not double count cycles to multiple components, otherwise we lose the intuitive meaning of the stacked representation, i.e., the sum of all components equals total time (or total bandwidth).

To avoid double counting, we propose a hierarchical accounting approach, giving the highest priority to the most meaningful reason. We first check if the DRAM is currently refreshing, which means that no operation can be started. These cycles are accounted to the *refresh* component.

If the DRAM is not refreshing, we check all the banks. If one or more of them is executing a precharge or activate command, a new page is opened and a data transfer is pending until the page is open. We account these cycles to the *precharge/activate* component. However, if one or a few banks are active, and the others are not performing a command, potential bandwidth by exploiting bank parallelism is not used. So assigning these cycles fully to the respective commands is not representative: if more banks were busy, the precharge/activate latency could be hidden by read/write commands on the other banks. Therefore, if there are n banks and at least one is busy, we assign $1/n$ cycles to each bank[1], either to the precharge/activate or *bank-idle* component. The bank-idle component reflects cycles that could be used by additional requests, provided that they are distributed across the banks and not accessing the busy banks. If there is a large bank-idle component, and it does not disappear by increasing the request rate, there is an issue with the distribution of requests across the banks.

If all banks are idle, we do not assign bank-idle cycles, we first check if there is another (rank or bank group) constraint preventing the start of a read or write command, such as the read/write to read/write timing constraints, and account to the *constraints* component if that is the case. If not, the DRAM chip is completely idle, and we assign the cycle to the *idle* component. Figure 1 shows an example where commands are issued to different banks in parallel, and which stack components are incremented on each cycle.

In summary, the lost bandwidth components mean and can be addressed by:

- *Idle*: The full DRAM chip is idle; increase the request rate (more threads, more memory-level parallelism).
- *Bank-idle*: Some banks are idle while others are active; increase the request rate, and if that does not work, make the distribution across the banks more uniform.
- *Precharge/activate*: Time is spent in closing and opening pages in banks; increase page hit rate by optimizing locality.
- *Constraint*: Different timing constraints limit throughput; try to avoid constant switching between reads and writes.
- *Refresh*: DRAM rows are refreshed; intrinsic to DRAM operation, nothing to do about.

Note that it is not always possible to address these issues. The memory address stream is often a characteristic of the application that cannot be changed. The bandwidth stack then shows that this is the case, and that there is no further margin to improve bandwidth utilization.

Due to the complexity of the accounting, bandwidth stacks are targeted to be collected during processor and/or memory simulation. Even for simulation, complexity and speed needs to be considered, to not impractically slow down simulation. Instead of accounting cycle by cycle, as explained in the

[1]To simplify the accounting, we add 1 to each counter, and divide these specific counters by n during postprocessing.

conceptual description above and which would be very time-consuming, we collect the traces of memory operations on the different levels. We then analyze them on the fly (we do not keep the full trace), by looking at the first read or write on the channel and analyzing the commands before that first channel transfer to find the events that prevented a transfer. Because each command takes a few cycles, we account multiple cycles in one step, which is much faster than a cycle-by-cycle approach. As an alternative to integrated simulation, a command trace (including timings) can be collected from the hardware [14] or a DRAM simulator [17], [18], and the bandwidth stack can be constructed offline from this trace using the accounting mechanism described in this section.

After collecting the components as cycle counts, a post-processing step transforms them into bandwidth components expressed in GB/s. Peak memory bandwidth equals the channel bandwidth, so each cycle corresponds to the amount of data the channel can transfer in a cycle. For example, if the data bus is 8 byte, and 2 transfers can be done per cycle (double data rate), each cycle corresponds to 16 bytes of data. Next, we divide by the total simulation time to obtain a bandwidth number in GB/s. For example, if we simulate 1 million memory cycles at 1.2 GHz, and the precharge component is accounted as 100,000 cycles, its bandwidth component equals

$$\frac{100{,}000 \text{ cycles} \times 16 \text{ B}}{1{,}000{,}000 \text{ cycles}} \times 1.2 \text{ GHz} = 1.92 \text{ GB/s}$$

V. Latency Stacks

To complement the bandwidth stacks, we also collect latency stacks. For each load operation that accesses main memory, we collect the DRAM latency, divided in multiple components. The *base* component is the minimum latency to perform a read without any contention or constraints. *Precharge/activate* (pre/act) is the extra latency to precharge and activate a row in case of a page miss. The remaining latency is queueing time, i.e., the time a request has to wait until it is started. To provide more insight, we subdivide the waiting time further. The time a request was delayed because the DRAM was refreshing is accounted to the *refresh* component. During a write burst, no reads can be scheduled, so we account this time to the *writeburst* component. The remaining queueing latency is due to bandwidth and/or timing constraints, and is added to the *queue* component.

Latency stacks are more straightforward to measure than bandwidth stacks, because we do not need to take into account overlap effects: the components are measured for each individual read operation, and then averaged over all read operations. We only consider read operations, because they have a direct impact on core performance: loads that need to access main memory stall the core until they are finished. Writes, on the other hand, usually do not stall a core.

Although latency and bandwidth are correlated, bandwidth stacks and latency stacks provide different information. High bandwidth usage often incurs high queueing times, and thus higher latency. However, sometimes latency is high when bandwidth usage is still far from the peak bandwidth. In this case, the latency stack can provide an explanation. The higher latency is then often the cause of low bandwidth consumption: cores are stalled and generate memory requests at a lower rate.

Another example of the complementarity between bandwidth and latency stacks is the interpretation of the bank-idle bandwidth component. If the bank-idle component is high, it can either be because of a low request rate or because there are many accesses to the same bank at the same time. In the first case, there is no significant queueing latency in the latency stack, while the second case will show a high queueing delay. To increase bandwidth usage in the first case, request rate should be increased (e.g., more threads), while in the second case, bank interleaving should be improved.

VI. Experimental Setup

We have implemented the bandwidth stack and latency stack accounting mechanism in an in-house version of the Sniper multicore simulator [3], extended with a DRAM model based on the Ramulator DRAM simulator [12], [17]. Because the Sniper core models are not synchronized cycle by cycle (for high simulation speed), memory accesses can occur out of order, so the first step is to reconstruct an in-order command trace for each memory controller (on the fly). Next, we apply the bandwidth stack algorithm to collect the components of the stack. The latency stack accounting is done per read operation. We extend the DRAM model to differentiate between the queueing components, and check whether a read was delayed because of refreshes or write bursts. Similar to the cycle stacks in Sniper, bandwidth and latency stacks can be collected aggregated over the full processor (all memory controllers), per memory controller and/or through time (showing a stack per time unit, which is useful for detecting phase behavior).

To show the usefulness of bandwidth and latency stacks, we simulate a single DDR4-2400 memory controller with FR-FCFS scheduling policy, attached to 1 to 8 Intel Skylake-like cores [6]. The cores are 4-wide out-of-order cores, with a 224-entry ROB. L1 instruction and data caches are 32 KB each, the private L2 cache is 1MB. To factor out caching effects, we keep the shared last-level cache the same for all core counts (8 NUCA slices for a total of 11 MB).

The memory module has one channel and one rank, 4 bank groups and 4 banks per bank group, for a total of 16 banks. The page buffer is 8 KB (128 64-byte cache lines per row). The frequency is 1.2 GHz, and the data bus width is 8 byte. With two transfers per cycle (double data rate), the transfer rate is 2400 MT/s, resulting in a peak bandwidth of 19.2 GB/s.

We simulate increasing traffic (i.e., attaching more cores), different bank indexing (impacting the distribution of accesses across cores), open and closed paging policy, and different read/write fractions. The goal is to show that the effect on the resulting bandwidth stacks is as expected from the bandwidth and latency stack component definition. As validation workload we use synthetic benchmarks with a sequential and a random memory access pattern and a configurable load/store fraction. Synthetic benchmarks are more straightforward to understand and analyze than real applications, which makes it

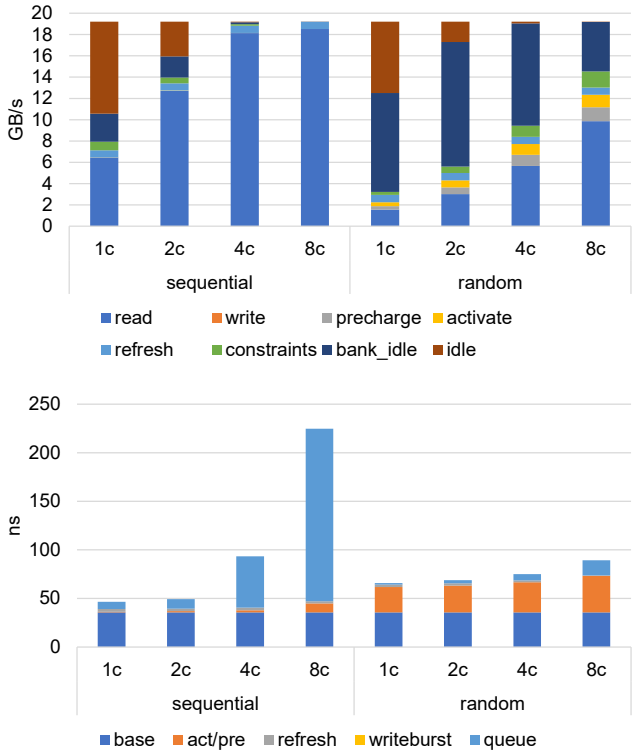

Fig. 2. Bandwidth (top) and latency (bottom) stacks for the sequential and random pattern with loads only, on 1 to 8 cores.

easier to show the intuitiveness and usefulness of bandwidth stacks. We also simulate the GAP benchmarks [1], which are representative for graph workloads and are known to be memory bound, to show how the stacks can be used to analyze real-world applications.

VII. Synthetic Benchmarks

A. Read-only

Figure 2 shows the bandwidth and latency stacks for the sequential and random access benchmark with only loads (no stores), on 1 to 8 cores attached to one memory controller with a peak bandwidth of 19.2 GB/s. The sequential pattern on one core reaches 6.4 GB/s read bandwidth. There is a large idle component, meaning that one core does not provide enough requests to saturate the bandwidth. The constraints component accounts for 0.8 GB/s lost bandwidth. The sequential pattern causes successive requests to the same bank: each row in a bank contains 8 KB of data, or 128 cache lines of 64 B. As a result, successive requests also access the same bank group, which has a lower bandwidth than the channel according to the specification of the modeled DDR4 configuration: a bank group can transfer one cache line in 6 memory cycles, while the channel only needs 4 cycles. This constraint on the bank group bandwidth is visualized by the constraints component. During the bank group constraint waiting time, the other banks are idle, which is visible through the bank-idle component (2.6 GB/s). The refresh component is constant for all stacks: DRAM is refreshed at a fixed rate. There is no precharge/activate component: page hit rate is 99% because of the sequential pattern. The latency stacks shows that the latency is close to the base read latency, with small refresh and queueing components.

Increasing the core count increases the request rate and thus also the achieved bandwidth. The increase is proportional to the number of cores, until the maximum bandwidth (minus the refresh rate) is achieved at 4 cores. At that point, queueing latency increases significantly: the request rate is larger than the bandwidth, leading to long waiting times. Note that the relative size of the bandwidth stack constraints and bank-idle components is lower for 2 cores and disappears for 4 and 8 cores. Each core accesses different parts of the sequential pattern, spreading the resulting requests over bank groups, which reduces the impact of the lower bank group bandwidth. The DDR4 chip we simulate has 4 bank groups, so at 4 cores, these components mostly disappear.

The random pattern has much lower bandwidth usage. The sequential pattern has perfect spatial locality and perfect predictability, meaning that caches and prefetchers are very effective in hiding the memory latency from the core, which increases the request rate. These structures cannot handle a random pattern, so the core sees the full memory latency and is thus often stalled on memory operations, which reduces the request rate and thus the achieved bandwidth. Due to the random pattern, page hit rate is 0%, which explains the precharge/activate components in both the bandwidth and latency stacks. There is also a large bank-idle component in the bandwidth stack: banks are busy longer because they need to precharge/activate for every access, meaning that there are more cycles where one or a few banks are busy and the others are idle. However, there is no large queueing component in the latency stack, indicating that the bank-idle component is caused by a low request rate. The constraints component is also large and increases with core count; this component now also includes extra constraints between pre/act commands.

As core count increases, the bandwidth usage also increases, but not fully proportional: the bandwidth usage at 8 cores is only 6.4 times higher than that of 1 core, despite the fact that the bandwidth usage at 8 cores uses only 51% of the 19.2 GB/s peak bandwidth. The bandwidth stack provides the explanation: at 8 cores, there is no idle component, meaning that the memory chip is active all of the time processing requests. Furthermore, the page misses cause bandwidth losses due to precharge/activate and the corresponding constraints, resulting in a relatively small bank-idle component (24% of the total bandwidth). This means that three quarters of the banks are busy at any time, resulting in a 75% chance of accessing a busy bank, which in turn causes queueing time, reflected by the increased queueing component in the latency stack. The extra latency makes the cores stall longer, which reduces the request rate and thus the achieved bandwidth.

It is interesting to compare the sequential pattern at 2 cores and the random pattern at 8 cores: the former has a

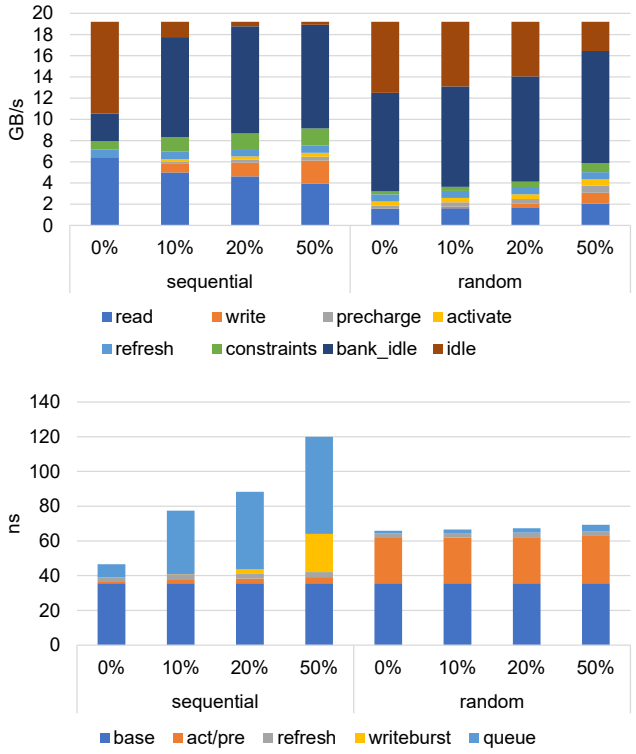

Fig. 3. Bandwidth (top) and latency (bottom) stacks for the sequential and random pattern with increasing store fraction on 1 core.

higher bandwidth usage *and* a lower queueing delay than the latter, which is contrary to the intuition that queueing latency increases with higher bandwidth usage. The bandwidth stack explains this behavior: due to page misses and constraints, there are fewer idle cycles in the random pattern than in the sequential pattern, leading to more queueing.

B. Impact of writes

In the next experiment, we add store operations, causing writes to memory. The cache organization with write-allocate policy induces both a memory read and a write on a store operation to a non-cached line: a read to load the cache line into cache, and a write somewhat later when the dirty cache line is evicted. Figure 3 shows the effect of increasing the store fraction from 0% to 50% for the sequential and random pattern on one core (the 0% stacks are the same as the 1c stacks in Figure 2).

Adding stores on the sequential pattern decreases the read bandwidth, which is not fully compensated by the additional write bandwidth (total bandwidth usage of 5.8 GB/s for 10% writes versus 6.4 GB/s read bandwidth for the 0% write pattern). This is contrary to expectation: because a store adds both a read and write request and cores do not stall on stores, we expect both read and write bandwidth to increase. The explanation is found in the large bank-idle component, combined with the large queueing component in the latency stack. This indicates that there is a problem with bank interleaving, as explained in Section V. The writes do not occur immediately at the execution of the store, but on the eviction of the cache line, which can be significantly later. This means that the ideal bank interleaving pattern of the sequential pattern is broken: writes can access the same bank as the current read pattern, but on a different page. This leads to queueing, and also to precharge/activate components because the read and write pattern access a different page. The queueing and bank conflict latency reduces the core request rate, explaining the lower overall bandwidth usage.

Increasing the store ratio to 20% and 50% further increases the queueing time, leading to lower read bandwidth (partly compensated by the additional write bandwidth). We also see an increase in the writeburst latency. Writes are buffered in the memory controller to prioritize reads, which are more critical for the core. If there are idle cycles or the buffer fills up, a burst of writes is issued, during which no reads are done. This burst is intentional to limit the overhead of switching between reads and writes. However, the sequential store pattern also causes a sequential write pattern at cache evict because of the LRU cache replacement policy. Writes in the write buffer therefore access the same bank, and are serialized on a write burst, because they cannot exploit bank parallelism. This causes a long time to issue the writes and empty the write buffer, during which all reads are waiting, causing the high queueing delay. Note that the writeburst delay can also have an indirect impact on the queueing delay: reads that are delayed because of a write burst create a read burst after the write burst is done, delaying the reads that arrive after the write burst. Because these reads arrive after the write burst, their delay is not accounted to the writeburst component, but to the queue component: they are delayed by other reads.

For the random access pattern, we do see a monotonic increase in the achieved bandwidth when increasing the store ratio: both the read and write bandwidth increase. There is no substantial increase in the queueing time, because writes are now better distributed across the banks, reducing the write burst and bank conflict time. There is also an increase of the precharge/activate and constraints bandwidth components.

C. Open versus closed page policy

The timing when pages are closed is determined by the page policy in the memory controller. An open page policy keeps pages open until another page on that bank is requested. To load another page, the current page has to be written back to the DRAM arrays (precharge command) and a new page is read (activate command). A closed page policy closes a page (by requesting a precharge) as soon as there are no pending accesses to that page anymore. If another page is requested later on, only an activate command needs to be issued, avoiding the latency of the precharge command. However, if the same page is accessed, there is an extra activate penalty compared to the open page policy. All results in the previous sections used an open page policy.

Figure 4 shows the bandwidth and latency stacks for the read-only sequential and random pattern on two cores, using

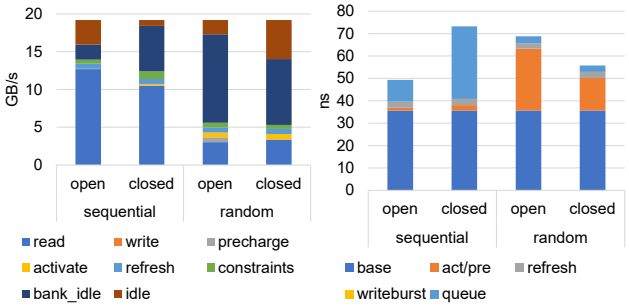

Fig. 4. Comparing bandwidth and latency stacks for open and closed page policy, for the read-only sequential and random pattern on 2 cores.

an open and closed page policy. As expected, the sequential pattern performs worse (lower bandwidth usage and higher latency) with a closed policy: most requests are to the currently open page, closing the page too early leads to longer latency. One might expect a larger increase in the precharge/activate component in the latency stack, but the largest increase is in the queueing component. This is because after closing a page, the first access has an extra activate latency, but the following accesses (to the same bank and page) have to wait for the precharge and read to end, which is accounted to the queueing latency component. This phenomenon is also visible in the bandwidth stack: there is a larger bank-idle component. Distributing subsequent accesses across banks would reduce queueing in this case.

For the random access pattern, bandwidth usage slightly improves for a closed policy (+11%) and latency reduces. This is also expected: each request accesses a different page, and closing pages earlier avoids the precharge latency, as is visible in the reduced pre/act latency component. The precharge component also disappears in the bandwidth stack, meaning that precharges are done in parallel with data transfers. The bank-idle component reduces because requests take less time, resulting in a larger full chip idle component.

D. Bank indexing

The bank indexing scheme determines which bank is accessed given a physical memory address. Figure 5(a) shows how the bank group, bank, row and column (cache line in a page) are indexed based on the physical address for our default setup. There is only one channel and one rank in our setup, so we do not need to index channel and rank. Cache lines are 64 byte, so the lower 6 bits determine the offset in a cache line. Pages contain 128 cache lines, so the next 7 bits determine the column in a DRAM page. In order to maximize the distribution of the requests across banks and bank groups, the next two bits select which of the 4 bank groups to address, and the next 2 determine which of the 4 banks within that bank group. Each bank consists of 32 Ki rows, so the upper 15 bits determine the row index. Note that this means that this memory controller can access 4 GB of data (32 address bits).

In a few cases in the previous sections, we noticed a large bank-idle component, combined with a large queueing

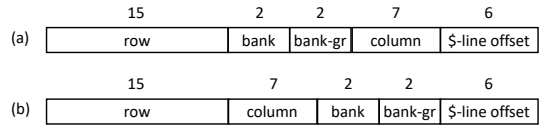

Fig. 5. Default (a) and cache-line interleaved (b) indexing schemes.

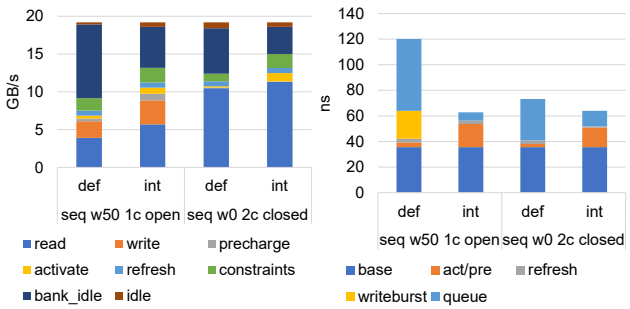

Fig. 6. Comparing default (def) and cache line interleaved (int) indexing for 2 use cases with high bank-idle and queueing components.

component, which indicates poor bank-level parallelism. This occurs in the sequential pattern, where successive requests access the same page on the same bank, causing contention and queueing. A solution to this issue is to distribute the sequential addresses across the banks, by interleaving cache lines across the banks. Figure 5(b) shows the alternative index scheme. The lower bits next to the cache line offset are now used to index the bank group and bank, moving the column index bits to higher address bits. We do not move the column bits higher, beyond the row bits, to retain page locality: once all banks are accessed, the stream returns to the first bank on the same page.

Figure 6 shows the bandwidth and latency stacks for the default (def) and cache line interleaved (int) indexing scheme for 2 cases where bank parallelism is a problem: a sequential stream with 50% stores and a sequential stream with a closed page policy on 2 cores. For both, bandwidth usage increases and latency decreases, which means our analysis based on the bandwidth and latency stacks was correct. The activate/precharge components increase: each sequential stream now opens a page on all banks, which causes more page misses because there are multiple streams, i.e., the read and write stream for the w50 case and the streams of the 2 independent cores for the 2c experiment. This is compensated by the decrease in the queueing and writeburst components: sequential accesses now don't queue up in the same bank. This indexing scheme, however, is in general not beneficial: for configurations where there is no large queueing component, the activate/precharge component still increases considerably, which is not compensated by a decrease in queueing, leading to a performance loss.

VIII. GAP BENCHMARKS

In the previous sections, we showed using simple synthetic streams that the proposed bandwidth and latency stacks reveal

meaningful information on why the expected bandwidth is not reached and what the potential solutions are. We now collect these stacks for realistic benchmarks. The GAP benchmarks [1] are basic graph kernels, representative for emerging graph workloads. Graph kernels are memory intensive, meaning that their performance is determined by memory behavior, and are therefore suited to analyze using bandwidth and latency stacks. Graph workloads typically have a mix of sequential and random patterns and cannot be categorized exclusively in one of the two patterns discussed previously.

Similar to the synthetic workloads, we simulate (and collect stacks of) the benchmarks for 1 to 8 cores, for an open and closed page policy. In general, the GAP benchmarks have better performance with a closed policy, due to the irregular access patterns caused by the irregular graph structure, so we only show results with the closed page policy. One exception is *triangle count* (tc), which mainly does sequential accesses and thus favors an open page policy. For the sake of brevity and because many results look very similar, we highlight some of the more interesting results.

A. Phase behavior and cycle stack correlation

Like many applications, the GAP benchmarks have different phases: different parts of the code and/or different data leading to different behavior. Because a single stack hides this phase behavior, our bandwidth and latency stack accounting mechanism records the data for small time samples, creating a through-time stack representation. Figure 7 shows the through-time processor cycle, memory bandwidth and memory latency stacks for breadth-first search (bfs) on 8 cores. There is highly varying behavior, depending on the type of algorithm (forward calculation until 35 ms, backward calculation afterwards) and the size of the working set (e.g., the first backward phase between 35 ms and 75 ms, and the second backward phase between 75 ms and 93 ms). There is a dip on all graphs around 30 ms, the cycle stack shows that this is because 7 of the 8 cores are idle (87.5% idle component), due to limited parallelism (which is why the algorithm switches to the highly parallel backward variant).

Bfs is highly memory bound, indicated by the large dram components in the cycle stack. The size of the dram components in the cycle stack correlates well with the bandwidth usage in the bandwidth stack and the queueing component in the latency stack. However, the cycle stack impact of memory operations in the first phase of the application is higher than in the phases after 35 ms, although the latter have higher bandwidth usage and higher latency. This means that the out-of-order core hides less of the memory latency, because there are too few independent instructions that can be done concurrently with the memory access. Improving memory performance on that phase will therefore improve performance more than on the other phases.

Looking at the first phase in particular, we notice a large bank-idle component in the bandwidth stack and a visible writeburst component in the latency stack (also noticeable in the other memory-bound phases). This indicates a potential

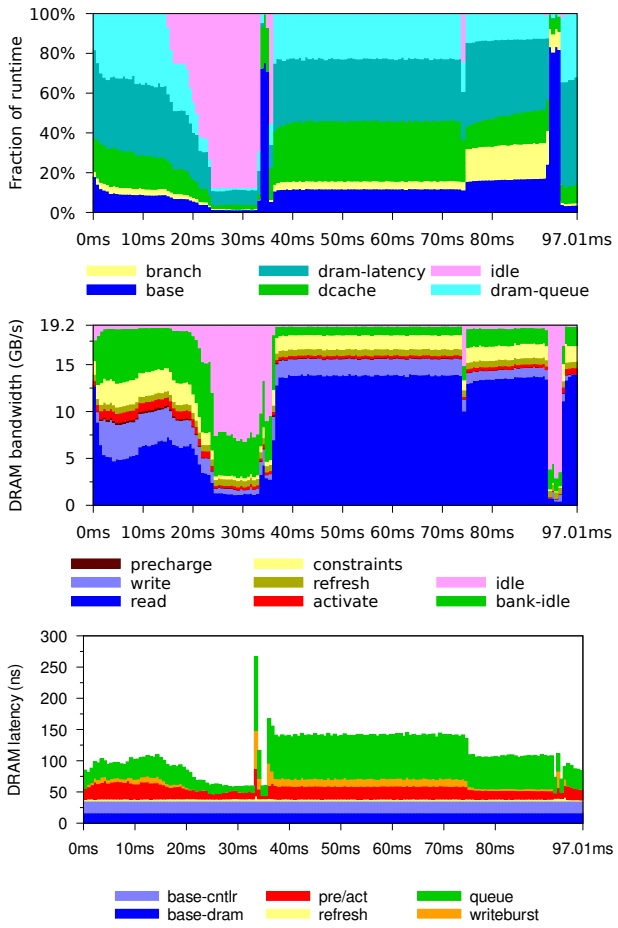

Fig. 7. Through-time cycle (top), bandwidth (middle) and latency (bottom) stack for bfs on 8 cores.

bank interleaving issue, as we also encountered with the synthetic benchmarks. Therefore, we simulate this benchmark with the cache-line interleaved indexing scheme, discussed in Section VII-D. Figure 8 (left) shows the aggregated latency stack for bfs using the default (def) and cache-line interleaved (int) indexing scheme. The queueing and writeburst components indeed reduce, at the cost of a larger precharge/activate component. The latter is caused by a lower page hit rate: from 41% for the default scheme to 8% for the interleaved scheme. As a result, total average latency, as well as bandwidth usage and performance, is almost the same for the two schemes.

Another way to reduce the writeburst component is to increase the write queue. A larger write queue causes fewer write bursts, and also increases the chances to issue writes on an idle period. Figure 8 also shows the latency stack for bfs with an 128-entry write queue, instead of the default 32 entries. The writeburst component is indeed reduced, along with a small reduction in the pre/act component, because read operations can be reordered more to minimize page misses. However, because bandwidth usage is already high and write

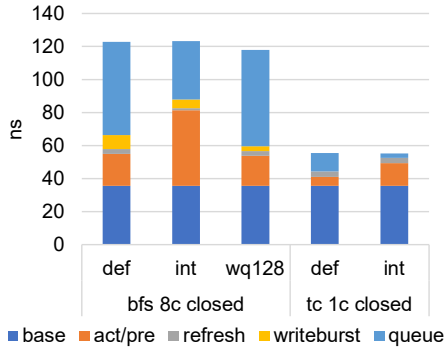

Fig. 8. Latency stack for bfs on 8 cores (left) and tc on one core (right) for the default configuation (def), the cache-line interleaved (int) indexing scheme and an 128-entry write queue (wq128).

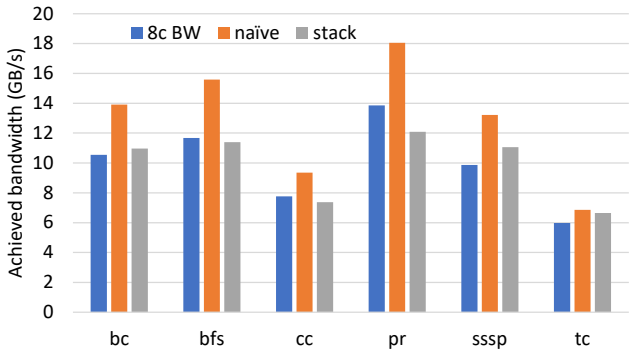

Fig. 9. Measured (8c BW) and extrapolated bandwidth usage at 8 cores, using the naive and stack-based method for the 6 GAP benchmarks.

bursts now take longer, the queueing component increases, undoing part of the gains.

Figure 8 (right) shows the latency stack for triangle count (tc) on one core with a closed page policy. Despite the very low bandwidth usage (0.9 GB/s), there is a considerable queueing component. Tc mainly performs sequential accesses, for which we found a similar issue in the synthetic sequential pattern: on a page miss, the activate latency makes successive accesses that go to the same bank wait, causing queueing. The cache-line interleaved indexing can also address this issue, by spreading successive accesses across banks. As seen on the figure, the queueuing component indeed reduces, but this is again completely offset by the increased page miss rate. Different from the synthetic sequential pattern, tc has multiple concurrent sequential streams, even with one thread. These interfere more if more pages are opened concurrently by one stream. Tc benefits more from an open page policy, reducing average latency to 44 ns.

These examples show that although the bandwidth and latency stacks provide more insight into the memory behavior and how it impacts performance, it is not always easy to improve performance by addressing bottlenecks. Optimizing for one component can impact other components, undoing the potential gain. Even at the software side, it is difficult to change access patterns to improve memory behavior. For example, graph applications are known to have irregular access patterns with low locality. Improving locality, e.g., using clustering and partitioning, is on itself NP-hard.

B. Extrapolating bandwidth usage

All of the benchmarks consume less than 1/8 of the peak bandwidth at 1 core. A straightforward extrapolation would be that the bandwidth, and thus the performance, scales linearly to 8 cores in the absence of other limiting factors. Our experiments show that none of the benchmarks scale perfectly linearly: the performance and bandwidth usage at 8 cores is lower than 8 times that of 1 core, despite a lower than peak bandwidth usage. The bandwidth stacks reveal other bandwidth limiting factors, such as precharge/activate, timing constraints and limited bank parallelism. Bandwidth stacks can be used to more accurately extrapolate bandwidth usage.

The main idea is that not only the achieved bandwidth scales with the number of cores, but also the other non-idle components: if traffic increases, more banks will be precharging/activating and more constraints need to be enforced because there is a denser execution of reads and writes. A first-order way to extrapolate the bandwidth stack is to multiply each component with the core count increase factor, except for the idle components (idle and bank-idle), which will obviously reduce. We also do not scale the refresh component, which remains constant. If the sum of the extrapolated components (without the idle components) exceeds the peak bandwidth, the performance becomes bandwidth bound. In this case, we scale down the components proportionally, such that the total stack equals the peak bandwidth. This leads to a reduced achieved bandwidth (read and write bandwidth component), which is the predicted bandwidth usage at the extrapolated core count.

To validate this approach, we start from the 1-core bandwidth stack and extrapolate the bandwidth usage to 8 cores, which we can compare to the actually achieved bandwidth in the 8-core simulation. We compare to a more naive approach that multiplies the achieved bandwidth at 1 core by 8 and saturates at the peak bandwidth (peak bandwidth minus the refresh rate, to be more accurate). Because the GAP benchmarks show phase behavior with varying bandwidth usage, and thus varying scaling behavior, we apply both methods for each measured sample and aggregate afterwards.

Figure 9 shows the achieved bandwidth at 8 cores (simulated) and the extrapolations using the two methods. It shows that the stack-based method is more accurate than the naive method; the average error is 27% for the naive method versus 8% for the stack-based method. Because there are other factors impacting bandwidth usage, such as synchronization overhead, cache interference, etc., this method cannot be absolutely accurate, but the results show that this relatively simple method already yields more than 3 times more accurate predictions than the naive method.

IX. Conclusions

Memory bandwidth usage is an important metric in performance analysis. DRAM operation is complex, which complicates analyzing bandwidth usage. Memory bandwidth and latency stacks are an intuitive way to visualize the bottlenecks preventing high bandwidth usage and what their impact is on performance. Using synthetic benchmarks, we show that the stacks are intuitive and meaningful. We evaluate graph applications to show that bandwidth and latency stacks can be used together with cycle stacks to get an in-depth view of an application's performance and to potentially improve its performance. Bandwidth stacks also enable more accurate extrapolations of bandwidth usage when scaling out to more cores.

References

[1] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP benchmark suite," *CoRR*, vol. abs/1508.03619, 2015. [Online]. Available: http://arxiv.org/abs/1508.03619

[2] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[3] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations," in *SC*, Nov. 2011.

[4] S.-J. Cho, J. Ahn, H. Choi, and W. Sung, "Performance analysis of multi-bank dram with increased clock frequency," in *2012 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2012, pp. 2477–2480.

[5] I. Corporation, "Vtune profiler: Memory access analysis," 2022. [Online]. Available: https://software.intel.com/en-us/vtune-help-memory-access-analysis

[6] J. Doweck, W.-F. Kao, A. K.-y. Lu, J. Mandelblat, A. Rahatekar, L. Rappoport, E. Rotem, A. Yasin, and A. Yoaz, "Inside 6th-generation Intel Core: New microarchitecture code-named Skylake," *IEEE Micro*, vol. 37, no. 2, pp. 52–62, 2017.

[7] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten, "Bandwidth bandit: Quantitative characterization of memory contention," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2013, pp. 1–10.

[8] S. Eyerman, W. Heirman, K. Du Bois, and I. Hur, "Multi-stage cpi stacks," *IEEE Computer Architecture Letters*, vol. 17, no. 1, pp. 55–58, 2018.

[9] S. Eyerman, K. Du Bois, and L. Eeckhout, "Speedup stacks: Identifying scaling bottlenecks in multi-threaded applications," in *2012 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*. IEEE, 2012, pp. 145–155.

[10] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A performance counter architecture for computing accurate CPI components," in *12th International conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII)*. Association for Computing Machinery (ACM), 2006, pp. 175–184.

[11] S. Eyerman, W. Heirman, K. Du Bois, and I. Hur, "Extending the performance analysis tool box: Multi-stage CPI stacks and FLOPS stacks," in *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2018, pp. 179–188.

[12] S. Eyerman, W. Heirman, and I. Hur, "Modeling DRAM timing in parallel simulators with immediate-response memory model," *IEEE Computer Architecture Letters*, vol. 20, no. 2, pp. 90–93, 2021.

[13] C. Helm and K. Taura, "Automatic identification and precise attribution of DRAM bandwidth contention," in *49th International Conference on Parallel Processing-ICPP*, 2020, pp. 1–11.

[14] Y. Huang, L. Chen, Z. Cui, Y. Ruan, Y. Bao, M. Chen, and N. Sun, "HMTT: A hybrid hardware/software tracing system for bridging the dram access trace's semantic gap," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, no. 1, pp. 1–25, 2014.

[15] A. Ilic, F. Pratas, and L. Sousa, "Cache-aware roofline model: Upgrading the loft," *IEEE Computer Architecture Letters*, vol. 13, no. 1, pp. 21–24, 2014.

[16] JEDEC, "DDR4 SDRAM standard (JESD79-4D)," 2021. [Online]. Available: https://www.jedec.org/standards-documents/docs/jesd79-4a

[17] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible DRAM simulator," *IEEE CAL*, vol. 15, no. 1, pp. 45–49, 2015.

[18] S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. Jacob, "DRAMsim3: a cycle-accurate, thermal-capable DRAM simulator," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 106–109, 2020.

[19] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[20] H. Xu, S. Wen, A. Gimenez, T. Gamblin, and X. Liu, "DR-BW: identifying bandwidth contention in NUMA architectures with supervised learning," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2017, pp. 367–376.