MemSeal: An MLP-Aware Leakage-Free Memory Controller

Abstract

Timing channels can be exploited to leak information between two virtual machines running on a shared server. Indeed, cache timing channels are important components in the Spectre attack. A shared memory controller can also be leveraged to establish a timing channel. Recent efforts have designed memory controllers that eliminate such timing channels, but that incur throughput penalties of over 2×. This paper advances the state-of-the-art by better matching the memory controller policies to the memory-level-parallelism (MLP) needs of the application. Our new memory controller, MemSeal, improves upon the performance of the state-of-the-art policy by 11%.

1. Introduction

Cloud platforms allow consumers to manage their computation and storage at low cost. These consumers also demand high levels of security and privacy for their sensitive data/computations, e.g., electronic health records. To a large extent, low cost is achieved because cloud operators dynamically schedule many workloads, belonging to different clients, on shared hardware. Each client or workload may be assigned its own Virtual Machine (VM), thus isolating and protecting each client. However, when VMs execute on shared hardware, information side channels do exist between the VMs. For example, if two VMs share a hardware cache, the latencies and miss rates experienced by one VM can betray the cache behavior of the other VM. Such a timing channel has been exploited by Ristenpart et al. [10] to crack passwords in an Amazon cloud infrastructure, in spite of various noise and real-world phenomena.

The recent Spectre attack [8] relied on cache timing channels to learn the secret-dependent footprints left behind by speculative execution. A cache timing channel can be eliminated with techniques like cache partitioning [14, 15], which exist in some commercial hardware [9]. But an application's secrets can be leaked through a variety of side channels, including through shared hardware structures such as the memory controller and branch predictor. Modern processors do not currently incorporate defenses against memory controller timing channels. This is an important vulnerability that can be exposed by attacks styled after Spectre.

A concrete attack through the memory controller was demonstrated by Wang et al. [13]. They show that the memory intensity of a VM can correlate with the number of 1s in the private key of an RSA decryption algorithm. Thus, by monitoring the memory intensity of a co-scheduled VM, an attacker VM can narrow the search space and quickly estimate private keys used by co-scheduled VMs.

Another important exploitation of memory-based timing channels is the establishment of a covert channel. Consider a cloud infrastructure set up by a hospital that includes electronic patient health records. The VMs used by the hospital may use state-of-the-art firewalls and encryption to protect their sensitive data. But they may use untrusted third-party software, e.g., a document reader, that has full access to unencrypted data and serves as a trojan. Because of the firewalls, the third-party trojan software can't easily send sensitive data outside the firewall. But by establishing a covert channel, it can leak information to another VM that is co-scheduled on the same hardware. In short, the third-party software can modulate the memory traffic rate, which is detected by the co-scheduled VM, thus communicating 1s (high traffic) and 0s (low traffic) to the co-scheduled VM. Recent work by Hunger et al. [5] has demonstrated memory-based covert channels that can leak information between threads at a high rate of 500 Kbps.

Therefore, to protect a sensitive cloud application, it is important to close the timing channel that exists within the memory system or memory controller. In other words, it is important to design a memory controller where the memory latencies experienced by one VM are independent of the memory activity of other co-scheduled VMs.

A few recent papers have designed such memory controllers. Wang et al. [13] introduced the first leakage-free memory controller. They leverage the concept of *Temporal Partitioning (TP)* that essentially allocates the memory controller to one VM at a time in round-robin fashion. During its *turn* to use the memory controller, a VM first issues memory requests, then enters a dead time when no new requests are issued. Thanks to the dead time, the VM's requests do not pose resource conflicts with requests issued by the next VM. While this eliminates leakage between VMs, it introduces significant performance slowdown. This is because of bandwidth underutilization and long queuing delays because each VM has to wait for its turn.

A follow-up work by Shafiee et al. [11] improved upon the performance of the TP design. It introduced a *Bank Triple Alternation (BTA)* mechanism that reduced the turn lengths for each VM. This approach imposes constraints on the banks that can be accessed in each turn, thus ensuring that accesses by one VM do not conflict with accesses by another VM.

Wang et al. [12] then introduced a solution, SecMC-NI, that represents the state-of-the-art in leakage-free memory controllers. SecMC-NI grows the turn length for each VM, but imposes scheduling constraints to maximize the memory requests issued in that turn. However, this design only achieves 46% of the throughput achieved by a non-secure memory controller, i.e., the slowdown is significant and the room for improvement is high.

In this work, we build upon the state-of-the-art in three ways. We first modify SecMC-NI with additional constraints similar to BTA. This helps reduce the turn length. We observe that an application's memory-level-parallelism (MLP) determines if this new scheduler is beneficial or not. We then introduce a simple policy that engages the new scheduler on a per VM basis. Finally, we observe that the new policy can yield higher benefits when combined with rank subsets. Our techniques only rely on hardware modifications and do not place a greater burden on the OS.

2. Background

2.1. Threat Model

As summarized in Section 1, we are targeting a cloud infrastructure where a single socket executes multiple different and unrelated VMs. These VMs may belong to different clients and do not trust each other. In fact, one of these VMs may be malicious and is attempting to extract information from other VMs. It does this by performing periodic memory accesses and recording average latencies for those accesses. Because of contention for the shared memory controller, these latencies reflect the memory intensity of co-scheduled VMs. This information leak either (i) betrays access patterns of co-scheduled threads, e.g., revealing the content of keys [10], or (ii) can be used to establish a covert channel between VMs. As described in Section 1, a covert channel can be used by a trojan application in one VM to betray secrets to a cohort in another VM [5].

Note that such timing channel attacks are performed without compromising the OS and without requiring physical access to the hardware. It can be performed by a remote user simply by asking the cloud operator to execute the user's VM.

Note that there may be several side channels in a system, e.g., timing channels in a shared cache, timing channels in a branch predictor, leakage through addresses visible on the memory bus, etc. To fully protect a VM, each of these side channels will have to be closed with piecemeal solutions. Therefore, the solutions in this paper that address the timing side channel in the memory controller must be combined with other orthogonal solutions, e.g., cache partitioning [14].

2.2. Memory Controller Basics

A memory controller manages the memory modules connected to a single memory channel. A channel is connected to multiple *ranks* and each rank is itself partitioned into multiple *banks*. Many timing parameters dictate when operations can be issued to each rank and bank. For a detailed rundown of relevant timing parameters, please refer to the descriptions in prior work [13, 11]. Here, we'll focus on the bottomline timing constraints imposed by those parameters.

The memory controller can issue a request to one rank, and follow it with a request to a different rank (so that both requests can be performed in parallel). For the DRAM simulation parameters described in Section 4, requests to different ranks can be issued with a minimum gap of 6 memory cycles. Shafiee et al. [11] show that if two accesses to different ranks are issued with a 7-cycle gap, they are guaranteed to not interfere with each other, regardless of whether the accesses are reads or writes.

In a similar vein, the memory controller can also issue requests to different banks (either in the same rank or in different ranks). Shafiee et al. consider all combinations of reads/writes and various timing parameters to show that requests to different banks will never interfere if they are separated by 15 cycles. Further, two requests to the same bank must be separated by 43 cycles so that the first access has enough time to complete (this assumes the worst case of a row buffer miss).

Based on the above discussion, we proceed with the following rules of thumb:

- Two accesses to different ranks must be separated by 7 cycles.
- Two accesses to different banks in a rank must be separated by 15 cycles.
- Two accesses to the same bank must be separated by 43 cycles.

By following these rules, the memory controller can create a schedule of memory accesses such that two VMs never conflict for resources and are not vulnerable to timing channels.

Note that most high-performance processors have multiple DDR3/DDR4 memory channels. Since each channel is independently controlled, we will focus on a single channel for most of our discussion.

2.3. Temporal Partitioning and Fixed Service

The first attempt at eliminating timing channels in the memory controller uses a policy referred to as Temporal Partitioning (TP) [13]. A VM is assigned a turn for a number of cycles (the turn length); memory requests are initiated in the early part of the turn; the VM does not initiate any requests in the last 43 cycles of the turn (referred to as *dead time*). This guarantees that all memory requests have been completed before the next VM begins its turn. As a result, the next VM views the memory system as a clean slate and its memory latencies are independent of the memory activity exhibited by the previous VM, i.e., no timing channels.

Shafiee et al. [11] create a deterministic triple-alternation schedule to reduce the dead time (bank triple alternation – BTA). They require that a VM, during a turn, issue requests to (say) bank-ids that are multiples of 3, i.e., bankid modulo 3 = 0. In the next turn, the next VM must issue requests to banks where bankid modulo 3 = 1. In the turn after that, the next VM issues requests to banks where bankid modulo 3 = 2, and so on. By grouping banks in this manner and imposing restrictions on which banks can be accessed in a turn, the memory controller guarantees that consecutive turns will never access the same banks. This makes it easier to create a non-conflicting schedule of memory accesses without a long dead time. Shafiee et al. show that dead time can be reduced to 14 cycles while ensuring that the second VM's accesses are independent of activity in the first VM.

Figure **??** shows an example schedule for both TP and BTA. Note that both of these works recommend issuing a single request per turn to keep the turn length short and not have long queuing delays. Both works also show that if the OS can map each VM to its own set of ranks or banks, it is easier to create non-conflicting schedules and improve performance. However, such approaches are not palatable because of the higher OS complexity, the impact on other cloud management policies, and the relatively small number of ranks per channel. We will therefore only consider techniques in this paper that can be implemented entirely in hardware.

2.4. SecMC-NI

In recent work, Wang et al. [12] introduced the state-of-the-art solution, SecMC-NI, with examples shown in Figure **??**. Wang et al. show that throughput can be increased by giving a VM a long enough turn to drain multiple requests, by using a short dead time, and by imposing some scheduling constraints that prevent resource conflicts across VMs.

Consider the example shown in Figure **??**. A VM is given a 43-cycle turn. Within that turn, the VM gets an opportunity to issue 6 memory requests – 2 requests each from 3 different ranks. All 6 requests are sent to different banks. A deterministic conflict-free schedule is set up by interleaving requests to the three ranks, as shown in Figure **??**. Following the rules of thumb described in Section 2.2, consecutive requests to different banks in the same rank are separated by more than 15 cycles; consecutive requests to different ranks are separated by at least 7 cycles[1].

In the first turn, VM-0 decides to issue requests to ranks R-3, R6, and R-7 (since those ranks have the most pending requests). Accordingly, two requests to R-3, two requests to R-6, and one request to R-7 are placed in the six issue slots (one slot is not utilized). In the second turn, VM-1 decides to issue requests to ranks R-1, R4, and R-2; it is able to fill all 6 slots. Note that in the two slots assigned to R-4, requests to banks B-3 and B-6 were issued. In the third turn, VM-2 decides to issue requests to ranks R-4, R-5, and R-6. The R-4 requests are to banks B-1 and B-6. Because of the schedule that was adopted for VM-1, VM-2 is forced to delay its access of B-6. This is done to preserve the 43-cycle gap between two accesses to the same bank. Because the banks being accessed in a turn are unique, we are guaranteed to find a correct conflict-free schedule for every turn, in spite of the constraints imposed by the schedule in the previous turn. Thus, a given VM is able to issue any requests it desires in a turn (while conforming to the rule that requests are to different banks in three ranks), regardless of what other VMs are doing.

With the above policy, the schedule may reveal what the other thread was doing. For example, if the previous VM repeatedly accessed rank R-5, the current VM's requests to R-5 typically happen in the later part of its turn. To eliminate this leakage, a VM should be oblivious to the schedule within its turn, i.e., it should not realize that R-5 accesses are later than others. This is enforced by waiting until the end of the VM's turn to return all its requests en masse.

To make the subsequent discussion more intuitive, we will refer to the SecMC-NI policy as *3R/2B*, i.e., a turn schedule that accesses 2 banks in 3 different ranks.

3. Proposed Memory Controller Policies

3.1. Drawbacks of SecMC-NI

As shown later in Section 5, the state-of-the-art SecMC-NI (or 3R/2B) solution achieves only 46% of the throughput achieved by a constraint-free non-secure baseline. If the VM does not have 2 accesses each to three different ranks, it will under-utilize the six available slots. Figure **??** quantifies this phenomenon. These results are for simulations with 8 VMs, 8 ranks, and a turn length of 43 cycles. Figure **??** shows a histogram to indicate the common scenarios encountered during a turn. The X-axis represents a number of scenarios, each with a given number of pending requests for a VM during a turn, and the number of requests that were actually issued. XX Say more here. We see that the amount of MLP in most applications is relatively low; in XX% of all turns, the number of pending requests is less than 4. This indicates that a turn length that accommodates six issue slots will often be under-utilized. When the number of pending requests is high, the number of issued requests is often (XX%) less than six – this is because the pending requests do not always exhibit a 3R/2B pattern. Overall, we observe that XX% of all issue slots in the 3R/2B scheduler go unutilized.

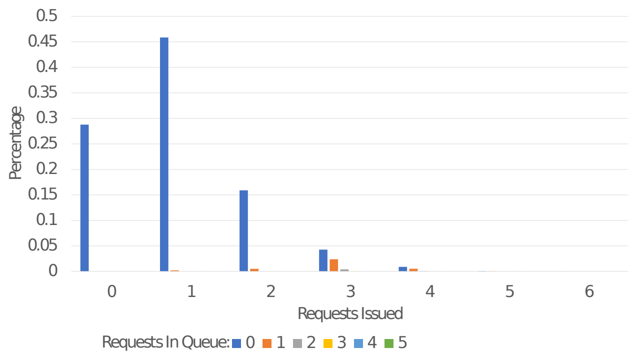

Figure 1: Histogram showing per turn stats for 3R/2B

[1] Some of the absolute numbers in our model are slightly different from that of Wang et al. [12] because we use slightly different DRAM timing parameters.

3.2. Ranked Triple Alternation (RTA)

The observations in the previous sub-section indicate that a 3R/2B baseline schedule leads to under-utilization because an application typically has low MLP. Two accesses to the same bank must be separated by 43 cycles; by having a 43-cycle turn length, the requests issued in one turn are not influenced by the requests issued in the previous turn. But a 43-cycle underutilized turn implies that applications have long wait times until they receive their next turn.

Our goal here is to shrink the turn length to improve utilization and reduce wait times. But if a shorter turn length is used, the banks touched in one turn are not free to receive new requests from the next turn, thus leaking information from one VM to the next. To prevent consecutive turns from touching the same banks, we introduce deterministic scheduling constraints that are application-independent. These scheduling constraints leverage the same concept used in BTA, but applied at the rank level.

We introduce *Rank Triple Alternation (RTA)*, which shrinks the turn length to 14 cycles and allows a VM to issue 2 requests to 2 different ranks in its turn, with the added constraint that *rank-id mod 3 = turn-id mod 3*. In other words, in turn 0, a VM is allowed to touch two different ranks with rank-ids that are both multiples of three; in turn 1, the next VM is allowed to touch two different ranks with rank-ids of the form 3N+1, and so on. After every 3 turns, a one-cycle bubble is introduced so that accesses to the same bank are separated by at least 43 cycles. An example of RTA is shown in Figure **??**. RTA is a much better match for applications that exhibit low MLP.

We also observe that RTA can issue two requests to the same rank in a turn as long as those two requests are of the same type (both reads or both writes) and are sent to different banks. RTA is also more effective than BTA because the gap between consecutive requests is usually 7 cycles in RTA and 15 cycles in BTA.

To ensure that a VM can touch different ranks in different turns, the total number of VMs should not be a multiple of 3. If the number of co-scheduled sensitive VMs is a multiple of 3, the OS can introduce a dummy VM or use *Rank Quad Alternation*. Also note that all of our schedulers employ an XOR address mapping policy to help evenly distribute requests across all ranks [16, 11].

3.3. Rank Subsetting

Rank Subsetting is used to increase the amount of parallelism by forming smaller parallel ranks. This increases the amount of banks and reduces queuing delays. This allows for the more turns to be scheduled at a time. However, forming smaller parallel ranks increases the data transfer time and reduces the size of the row buffer and cache line. The data transfer time, tBURST, increases from 4 cycles to 8 cycles with the addition of rank subsetting. Since tBURST has doubled, with our timing paramenters the restrictions placed on consecutive requests are now changed. The time between accesses to two different ranks is now 10 cycles, for two different banks it is now 25 cycles, and accesses to the same bank it is now 47 cycles. With these parameters the shortest possible turn that we can have without information leakage is 50 cycles. With turn lengths of 50 cycles, this allows for 5 requests to be scheduled a turn. The scheduled turns for 3R/2B must access atleast three different ranks and the ranks must have requests for atleast 2 different banks in that rank. This forces for the total length of the turn to extend from 50 cycles to 60 cycles.

RA can take advantage of the longer requests and reduce the number of requests in a turn to 1 with only a ranked alternation restriction of modulus 5. This greatly reduces the dead time from turns where cores did not have enough requests to fill their turn in RA and 3R/2B. As RA can take much more advantage of the rank subsetting than 3R/2B, RA outperforms 3R/2B in all workloads tested with ranked subsetting.

Figure X shows an example of a ranked alternation schedule with rank subsetting with 8 ranks and 8 banks. It uses a 10 cycle turn with a single 10 cycle request within each turn. Each of the requests access a different rank to ensure no information leakage.

4. Methodology

I have copied the methodology from our MICRO'15 paper to get you started.

For our simulations, we use Windriver Simics [4] interfaced with the USIMM memory simulator [3]. USIMM is configured to model a DDR3 memory system. While our target system is a 32-core processor with 4 channels, we limit simulation time by focusing on eight out-of-order processor cores and a single channel for most experiments. Simics and USIMM parameters are summarized in Table 1. Our baseline non-secure state-of-the-art scheduler is the best performing scheduler from the 2012 Memory Scheduling Championship [6].

We use a collection of multi-programmed workloads from SPEC2k6. Libquantum, milc, mcf, Gems-FDTD, astar, zeusmp and xalancbmk are run in rate mode (eight copies of the same program). SPEC programs are fast-forwarded for 50 billion instructions before detailed simulations are started. Simulations are terminated after a million memory reads are encountered. We used NPB workloads [2] CG and SP. We also consider the following workloads that mix benchmarks with varying memory requirements. Mix1 has two copies of xalancbmk, soplex, mcf and omnetpp. Mix2 has two copies of milc, lbm, xalancbmk and zeusmp. Each benchmark in these mixes is terminated after it executes the same number of instructions as in its baseline run. We assume that all co-scheduled programs receive an equal share of memory bandwidth and capacity.

For our memory energy analysis, we use the Micron power

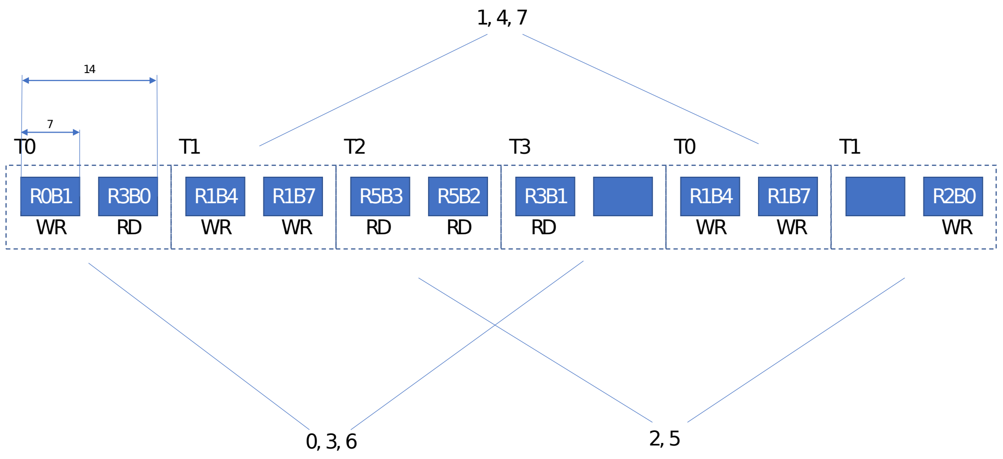

Figure 2: Example Schedule of RA with read-write bank reordering.

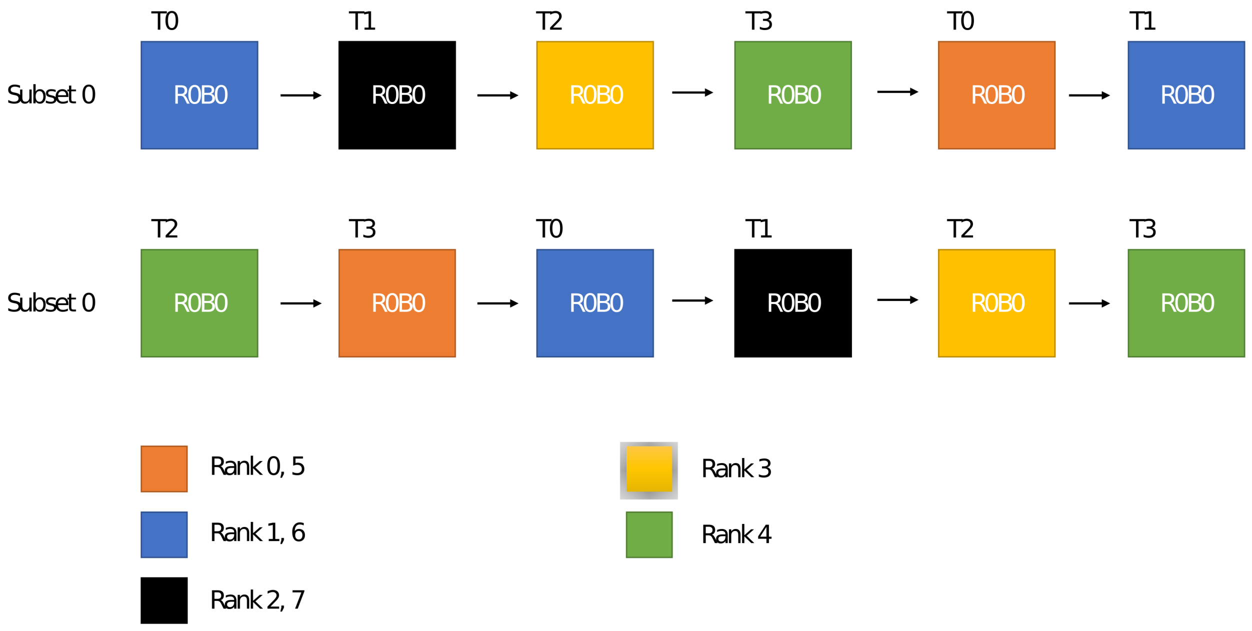

Figure 3: Example Schedule of Ranked Alternation with rank subsetting.

calculator for a DDR3 4 Gb part [1]. The calculator is fed with memory statistics collected during detailed Simics simulations. In the subsequent graphs we use the following abbreviations. TP_BP is Temporal partitioning with Bank Partitioning, TP_NP is Temporal partitioning with No spatial partitioning. FS_RP is Fixed-Sevice with Rank Partitioning, FS_Reordered_BP is Fixed-Service with Reordered Bank Partitioning, and FS_NP_Optimized is Fixed-Service with No Partitioning and the Triple Alternation optimization.

5. Results

5.1. Performance of RA

We first evaluate the effect of read-write bank reordering on the performance of RTA and compare their performance to the performance of RA. Figure X shows the results of RA with a modulus 7 restriction and RTA ran with and without read-write bank reordering using a XOR address mapping. Read-write reordering provides an average improvement of about 1%. It has a much larger affect on some workloads such as astar and xalancbmk due to the increased memory intensity. With a better address mapping to avoid similar banks, the ef-

Processor	
ISA	UltraSPARC III ISA
CMP size and Core Freq.	8-core, 3.2 GHz
ROB size per core	64 entry
Fetch, Dispatch, Execute, and Retire	Maximum 4 per cycle
Cache Hierarchy	
L1 I-cache	32KB/2-way, 1-cycle
L1 D-cache	32KB/2-way, 1-cycle
L2 Cache	4MB/8-way, shared,10-cyc
DRAM Parameters	
DRAM Frequency	1600 Mbps
Channels, ranks, banks	1 ch, 8 ranks/ch, 8 banks/rank
DRAM chips	4 Gb capacity
DRAM Timing Parameters (DRAM cycles) t_{RC} = 39, t_{RCD} = 11,t_{RAS} = 28, t_{FAW} = 24 t_{WR} = 12,t_{RP} = 11,t_{RTRS} = 2, t_{CAS} = 11 t_{RTP} = 6, t_{BURST} = 4,t_{CCD} = 4, t_{WTR} = 6 t_{RRD} = 5, t_{REFI} = 7.8μs, t_{RFC} = 260ns	

Table 1: Simulator and DRAM [7] parameters.

fects of read-write reordering become much less impactful. The performance for every other workload with read-write reordering is greater than or equal to compared to the without read-write reordering. The additional restritions placed in RA, RTA outperforms RA on average by 5%.

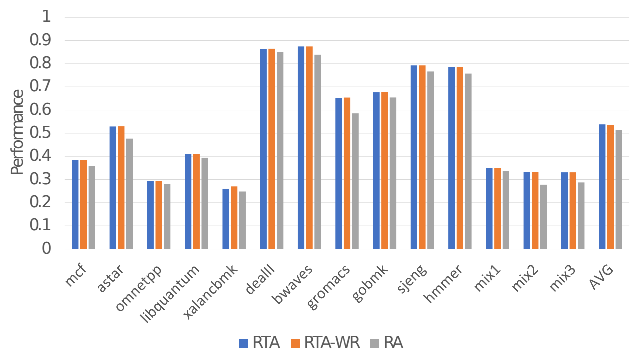

Figure 4: Performance comparision between RA and RTA with and without read-write reordering

5.2. Performance of Ranked Subsetting

The performance of ranked subsetting is shown in figure X. Figure X shows 3R/2B with rank subsetting (60 cycle turns with 6 requests per turn) and RA with rank subsetting (10 cycles turns with 1 request per turn). The performance increase from rank subsetting has a much larger impact on RA compared to SecMC-NI. This is because 3R/2B is able to lower the turn length down to 10 cycles while only requiring a mod 5 restriction on the ranks. 3R/2B must keep using a larger turn length to issue 6 requests which reduces performance as seen in figure X. 3R/2B shows a 9% performance decrease with rank subsetting and a 2% performance increase for RTA.

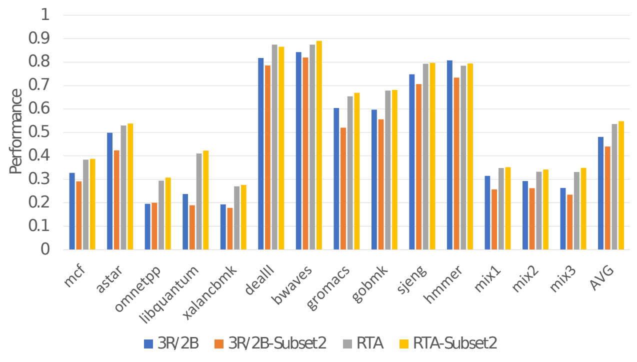

Figure 5: Performance comparision between 3R/2B and RTA with and without rank subsetting

Figure X shows the performance of rank subsetting using a 4-way rank subsetting and a 2-way rank subsetting with RA. For the 4-way rank subsetting, 18 cycle turns are given to each thread with 1 request per turn with a mod 3 rank alternation. The 4-way rank subsetting requires for the memory requests being issued to be extremely well mapped to be able to properly use the increased parallelism given by the 4-way ranked subsetting. They must be mapped to a subrank which can issue the request at the current rank and bank without any collisions for that threads turn in the subrank. Using a 4-way rank subsetting shows a 9% performance decrease over a 2-way rank subseting schedule.

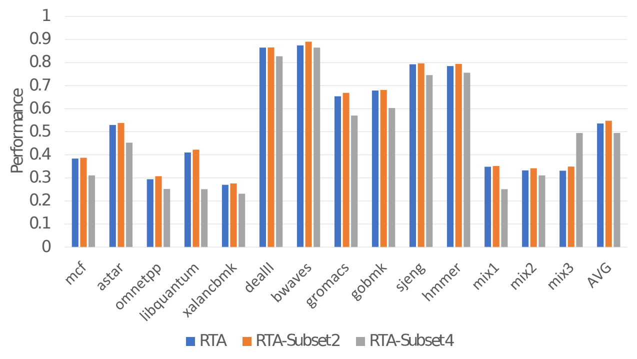

Figure 6: Performance comparision between 4-way and 2-way ranked subsetting for RA

5.3. Overall Performance

Figure X shows the overall performance for the ideas presented in this paper and previously proposed policies. The figure uses a XOR address mapping for all of the policies. The performance of RTA improves on the performance of 3R/2B

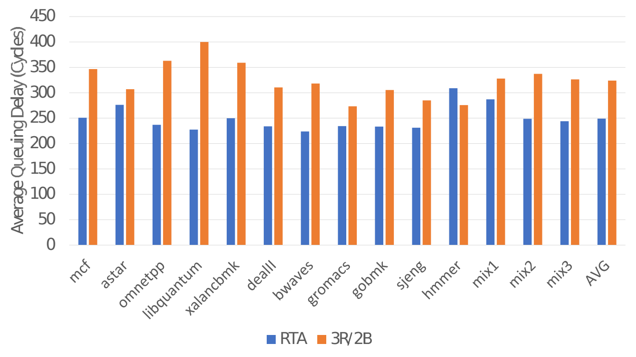

Figure 7: Performance comparision between 4-way and 2-way ranked subsetting for RA

without the use of rank subsetting by 11%. With rank subsetting the performance of RTA improves by 13% on 3R/2B while staying completely secure. It can be seen that in Figure X that RTA performs worse than 3R/2B for the hmmer workload. This is due to the low parallelism from the hmmer trace causing the additional restrictions placed by the alternation of RTA much more impactful. The XOR address mapping works to counteract this problem greatly lessening the impact. Figure Y, shows the average queuing delay of requests for RTA and 3R/2B. It can be seen that the queueing delay for hmmer with RTA is substantially larger than it is for 3R/2B.

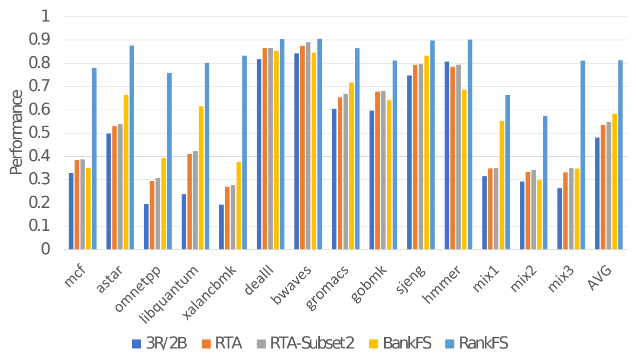

Figure 8: Overall performance of RTA, 3R/2B, and 2-way rank subsetting RTA, Bank FS, and Rank FS

Figure X shows the overall performance for RA with 2-way ranked subsetting, RTA, and previously proposed policies.

6. Related Work

Can skip this if or make it brief if you've already covered the major pieces in background.

7. Conclusions

References

[1] "Micron System Power Calculator," http://www.micron.com/products/support/power-calc.

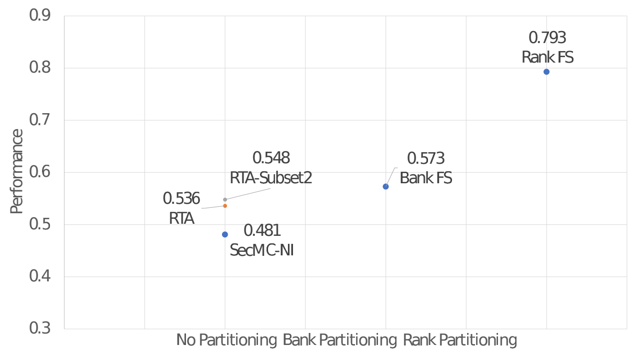

Figure 9: Overall performance of RTA, 2-way ranked subsetting RA, and previously proposed policies

[2] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS Parallel Benchmarks," *The International Journal of Supercomputer Applications*, vol. 5, no. 3, pp. 63–73, Fall 1994. [Online]. Available: http://www.nas.nasa.gov/Software/NPB/

[3] N. Chatterjee, R. Balasubramonian, M. Shevgoor, S. Pugsley, A. Udipi, A. Shafiee, K. Sudan, M. Awasthi, and Z. Chishti, "USIMM: the Utah SImulated Memory Module," University of Utah, Tech. Rep., 2012, UUCS-12-002.

[4] W. Company, "Wind River Simics Full System Simulator," 2007. [Online]. Available: http://www.windriver.com/products/simics/

[5] C. Hunger, M. Kazdagli, A. Rawat, S. Vishwanath, A. Dimakis, and M. Tiwari, "Understanding Contention-driven Covert Channels and Using Them for Defense," in *Proceedings of HPCA*, 2015.

[6] Y. Ishii, K. Hosokawa, M. Inaba, and K. Hiraki, "High Performance Memory Access Scheduling Using Compute-Phase Prediction and Writeback-Refresh Overlap," in *Memory Scheduling Championship*, 2012.

[7] JEDEC, *JESD79-4: JEDEC Standard DDR4 SDRAM*, 2012.

[8] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," 2018, https://spectreattack.com/spectre.pdf.

[9] K. Nguyen, "Introduction to Cache Allocation Technology in the Intel Xeon Processor E5 v4 Family," 2016, https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology.

[10] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds," in *Proceedings of the 16th ACM conference on Computer and Communications Security*, 2009, pp. 199–212.

[11] A. Shafiee, A. Gundu, M. Shevgoor, R. Balasubramonian, and M. Tiwari, "Avoiding Information Leakage in the Memory Controller with Fixed Service Policies," in *Proceedings of MICRO*, 2015.

[12] Y. Wang, B. Wu, and G. Suh, "Secure Dynamic Memory Scheduling Against Timing Channel Attacks," in *Proceedings of HPCA*, 2017.

[13] Y. Wang, A. Ferraiuolo, and G. E. Suh, "Timing Channel Protection for a Shared Memory Controller," in *Proceedings of HPCA*, 2014.

[14] Z. Wang and R. B. Lee, "New Cache Designs for Thwarting Software Cache-based Side Channel Attacks," in *Proceedings of ISCA*, 2007.

[15] Z. Wang and R. B. Lee, "A Novel Cache Architecture with Enhanced Performance and Security," in *Proceedings of MICRO*, 2008, pp. 83–93.

[16] Z. Zhang, Z. Zhu, and X. Zhand, "A Permutation-Based Page Interleaving Scheme to Reduce Row-Buffer Conflicts and Exploit Data Locality," in *Proceedings of MICRO*, 2000.