Avoiding Information Leakage in the Memory Controller with Fixed Service Policies

Ali Shafiee
University of Utah
Salt Lake City, UT, USA
shafiee@cs.utah.edu

Akhila Gundu
University of Utah
Salt Lake City, UT, USA
akhila.gundu@cs.utah.edu

Manjunath Shevgoor
University of Utah
Salt Lake City, UT, USA
shevgoor@cs.utah.edu

Rajeev Balasubramonian
University of Utah
Salt Lake City, UT, USA
rajeev@cs.utah.edu

Mohit Tiwari
University of Texas
Austin, TX, USA
tiwari@austin.utexas.edu

ABSTRACT

Trusted applications frequently execute in tandem with untrusted applications on personal devices and in cloud environments. Since these co-scheduled applications share hardware resources, the latencies encountered by the untrusted application betray information about whether the trusted applications are accessing shared resources or not. Prior studies have shown that such information leaks can be used by the untrusted application to decipher keys or launch covert-channel attacks. Prior work has also proposed techniques to eliminate information leakage in various shared resources. However, the best known solution to eliminate information leakage in the memory system incurs high performance penalties. This work develops a comprehensive approach to eliminate timing channels in the memory controller that has two key elements: (i) We shape the memory access behavior of every thread so that every thread appears identical to the memory system and to potential attackers. (ii) We show how efficient memory access pipelines can be constructed to process the resulting memory accesses without introducing any resource conflicts. We mathematically show that the proposed system yields zero information leakage. We then show that various page mapping policies can impact the throughput of our secure memory system. We also introduce techniques to re-order requests from different threads to boost performance without leaking information. Our best solution offers throughput that is 26% lower than that of an optimized non-secure baseline, and that is 70% higher than the best known competing scheme.

MICRO-48, December 05-09, 2015, Waikiki, HI, USA

DOI: http://dx.doi.org/10.1145/2830772.2830795

Keywords

Hardware Security

1. INTRODUCTION

Most modern computing platforms co-schedule multiple applications on shared hardware. In a cloud environment, a user application typically executes on a server with other untrusted applications. Also, an untrusted downloaded application typically executes on a user's computing device along with other trusted applications. These execution scenarios expose timing side-channels between the trusted and untrusted applications. By measuring delays to access shared resources, an attacking application can estimate resource usage patterns of the application being attacked. Such information leakage is then used to launch a more focused attack [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]. Ristenpart et al. [9] even demonstrate one possible attack on Amazon EC2 hardware that exploits cache timing channels to recover user passwords. Wang et al. [10] show that memory timing channels can be exploited in a similar way, or to establish covert channels.

When two or more threads run on a server, they share many on-chip and off-chip resources, such as L1/L2/L3 caches, the on-chip network, and the memory system. Many prior works have developed solutions to reduce information leakage in caches and on-chip networks [11, 12, 13, 14, 15, 16], but only one recent paper by Wang et al. [10] has examined information leakage in a shared memory system. Wang et al. propose *temporal partitioning (TP)* that allows only a single thread (or security domain) to issue memory requests in every time slice. This results in significant queuing delays and performance degradations that grow with thread count.

In this work, we consider a comprehensive approach to eliminate information leakage in the memory controller. Our solution uses a combination of memory access shaping, memory access pipelining with a mathematical model, and spatial partitioning. The memory controller shapes the requests emerging from a thread so that every thread has a constant injection rate, i.e., from the memory system's perspective, a thread's be-

havior does not change over time. To guarantee that the memory system can keep up with the worst-case pattern of memory accesses from threads, we design a novel memory access pipeline. In essence, if the input (the memory request pattern) to the memory controller is fixed, the output (commands issued to the memory system) can be fixed as well. We refer to this memory controller design as *Fixed Service (FS)*. Since a thread receives a fixed level of service, its behavior is not influenced by the memory access patterns of co-scheduled threads. The proposed memory access pipeline is designed to maximize throughput for the worst-case memory access pattern. We mathematically show that this combination of memory access shaping and the fixed service pipeline can eliminate timing-based information leakage in the memory controller.

We go on to show that our memory access pipeline is significantly more efficient if we assume some kind of spatial partitioning, i.e., if memory banks or ranks are partitioned among all threads. We also introduce other hardware techniques to improve performance when spatial partitioning is impractical. These hardware techniques re-order requests from different threads without betraying information about co-scheduled threads. Finally, we propose techniques to leverage a prefetcher and energy-efficiency modes.

We thus present a generalized approach for designing secure memory pipelines under a variety of assumptions. The prior TP work and optimized forms of TP are special cases of the models we describe. Our results show that with our proposed innovations, we out-perform the best TP model by 70% on average in an 8 core system.

2. BACKGROUND

In this section, we review DRAM basics, timing channel attacks, and the temporal partitioning policy of Wang et al. [10].

2.1 DRAM Basics

Readers familiar with DRAM architectures can skip this sub-section.

Memory Ranks and Banks. Typical high-end processors can have up to four DDR3/DDR4 memory channels, each managed by an on-chip memory controller. Each channel can support multiple *ranks*, where a rank is a collection of DRAM chips that work in unison to handle a cache line request. Each rank is itself partitioned into multiple *banks*. Each bank has a row buffer to store the last accessed row in that bank. Banks and ranks help support multiple outstanding transactions, thus enabling a high degree of parallelism in the memory system.

DRAM Commands. Read and write transactions are placed in per-channel *transaction queues*. A scheduling algorithm, e.g., FR-FCFS [17], is employed to pick the best candidate for issue. This transaction is decomposed into its constituent memory commands (Precharge, Activate, Column-read, or Column-write) that are then placed in logical per-bank in-order *command queues*. A read memory transaction first requires an Activate operation to populate the row buffer, followed by a Column-read to move the cache line from the row buffer to the memory controller. The Column-read can be issued with a special command that performs an automatic precharge immediately after the Column-read is complete. Alternatively, a separate Precharge command can be issued later – a Precharge closes the row and prepares the bank to read another row. If the required data is already present in the row buffer (a row buffer hit), the Activate operation can be avoided. A write memory transaction requires an optional Activate, followed by a Column-write, followed by an optional Precharge.

DRAM Timing Parameters. The scheduling of DRAM commands is governed by resource availabilities and a number of timing parameters. For example, two reads to different rows in a bank must be separated by time tRC, since the first access has to vacate the bank before another access can commence. A Column-read must happen at least $tRCD$ after its Activate to allow enough time to populate the row buffer. Time $tCAS$ after a Column-read, data begins to flow on the data bus. Two Activations to different banks in a rank must be separated by time $tRRD$; a single rank can only support up to four Activates within a sliding time window of length $tFAW$; both of these timing parameters are in place because the many banks in a rank share the same set of charge pumps and the power delivery network. Data transfers from two different ranks must be separated by time $tRTRS$ to accommodate the change in the bus driver. Two accesses to the same bank have to be performed sequentially, while accesses to different banks and ranks can be partially overlapped. In general, two requests to the same rank are governed by more timing parameters ($tFAW$, $tRRD$, tRC, $tWTR$, etc.) because the two requests have more resources in common. Two requests to different ranks are only constrained by their shared address/command/data buses and the $tRTRS$ parameter.

2.2 Threat Model and Security Policy

We target scenarios where processes from mutually distrustful *security domains* run concurrently on trusted hardware with a trusted OS/hypervisor. This scenario is typical of a cloud environment where an eavesdropper virtual machine (VM) can be co-located on a CPU with a victim VM and wants to infer the victim's secrets (termed as a *side*-channel attack). Alternately, the victim's VM could run a malicious application that (in addition to providing useful functionality) leaks secrets to the eavesdropper VM – this is termed a *covert*-channel attack. In addition to confidentiality, side- and covert-channel attacks can target the *integrity* of a trusted program's execution; for example, by affecting the victim's worst case execution time. Such attacks can be generalized to the problem of preventing illegal *information flows* given a lattice of security labels that represents the allowed flows of information [18, 19].

Specifically, this paper focuses on closing illegal information flows through the *memory controller* and enforcing strict non-interference [20, 21]. Memory-based side and covert channel attacks have been demonstrated in practice [10, 8, 22], where an attacker process mea-

sures its own overall execution time (a feature allowed on most hardware) and estimates memory access latencies. The memory access latency betrays queuing delays in the memory controller, which are a result of contention for shared resources (channel, rank, bank, etc.). The attacker can thus estimate its co-scheduled threads' memory intensity over time.

For example, Wang et al. [10] describe a side-channel attack on the RSA decryption algorithm – the victim RSA's memory accesses are correlated with the number of 1s in its private key. The attacker can gauge the victim thread's memory traffic, estimate the number of 1s, and thus narrow the search space to determine the private key.

Attacks on a production cloud system like Amazon EC2 are complicated by interference from the hypervisor or other guest VMs, cache and memory interleaving effects, etc. But such noise cannot eliminate information leakage through timing channels. For example, Wu et al. [8] show how to construct a 100 bits per second covert communication channel across two cores on an Amazon EC2 processor – far higher than the 1 bit per second suggested for high assurance systems [23] – using memory-bus locking instructions to contend for the bus. Recent work by Hunger et al. [22] shows that by synchronizing the covert channel sender and receiver, bandwidths of over 100Kbps can be achieved using the memory bus channel. In summary, designing a memory controller that can close off all timing channel leaks is crucial for a secure cloud.

While this work focuses on timing channels through the memory system, several other timing and side channels exist in real systems. As with any security proposal, the solution in this paper must be combined with a suite of side channel mitigation strategies (e.g., cache partitioning) to plug every possible system vulnerability. We anticipate that the proposed solution is orthogonal and compatible with most other side channel mitigation strategies. We later discuss interactions with side channels that are based on power measurements.

Our focus here is to design a secure scheduling protocol for memory controllers. A leaky *implementation* of our protocol can of course break its non-interference guarantees – the Heartbleed bug in the OpenSSL implementation of the Transport Layer Security (TLS) protocol is an example of such implementation-level vulnerabilities. Complementary research in gate-level information flow security [19, 21, 24, 25] can be applied to writing and verifying that an RTL implementation of our protocol does not introduce an information leak.

2.3 Temporal Partitioning (TP)

To alleviate timing channels in memory controllers, Wang et al. introduce Temporal Partitioning (TP) [10]. With TP, the memory controller and channel (address, command, data bus) are used exclusively by a single thread (or security domain) at a time. After a fixed time quantum (turn length [10]), the memory controller switches to a different thread. The lengths of the time quanta are determined beforehand by the operating system and/or the memory controller, based on priorities or memory demand. These lengths cannot change at run-time as a counter-measure against covert-channel attacks [10]. A suggested length for the time quantum is 96 ns.

To prevent a memory operation from spilling into the next time quantum and posing contention for the next thread, Wang et al. disallow the issue of new memory transactions near the end of a time quantum. This period at the end of the time quantum is referred to as the dead time. The dead time is about 65 ns and takes up most of the time quantum, i.e., only a small fraction of the time quantum is used to initiate memory transactions. With the TP policy, every bank must be precharged at the end of every time quantum. If this is not done, then subsequent row buffer hits/misses will reveal if other threads accessed that bank during their time quanta. To avoid the need for many precharges at the end of a time quantum, Wang et al. perform the Precharge immediately after a Column-read or Column-write.

Wang et al. also consider a bank-partitioned memory system, where threads are allocated to disjoint sets of banks. With such bank partitioning, a thread can overlap its data transfer with the start of a memory transaction from another thread [26]. This helps bring down the dead time to 12 ns. In later sections, we set TP in the context of our proposed designs.

3. PROPOSAL

In our proposed model, every thread makes a fixed demand on the memory controller and the memory controller provides a fixed level of service to each thread in return. We refer to this approach as *Fixed Service (FS)*.

3.1 FS Policies with Rank Partitioning

To keep the discussion simple, we start with the assumption that each memory rank is assigned to a single thread. In our example system with eight threads and eight memory ranks, each thread places all of its data in its single assigned rank. Later, we consider the effects of relaxing this assumption.

Shaping the memory access pattern. We force every thread to have an identical memory access pattern. A thread is forced to issue a memory request at an interval of every Q cycles. If the thread does not have a pending memory request at this time, a dummy request is inserted on behalf of this thread. We effectively rule out row buffer hits, so every memory request takes the same time and has an identical footprint on the memory system. Every memory transaction is broken into the same set of commands: an Activate followed by a Column-Read or a Column-Write. The Column-Reads and Column-Writes are issued with an auto-precharge to reduce command bus utilization. Thus, every thread looks identical. The memory requests from a thread are then inserted into the shared memory system and handled as in any baseline system. Resource collisions reveal nothing about the nature of the co-scheduled threads beyond what is already being advertized: one empty-row access every Q cycles.

Constructing an efficient pipeline. The key here is

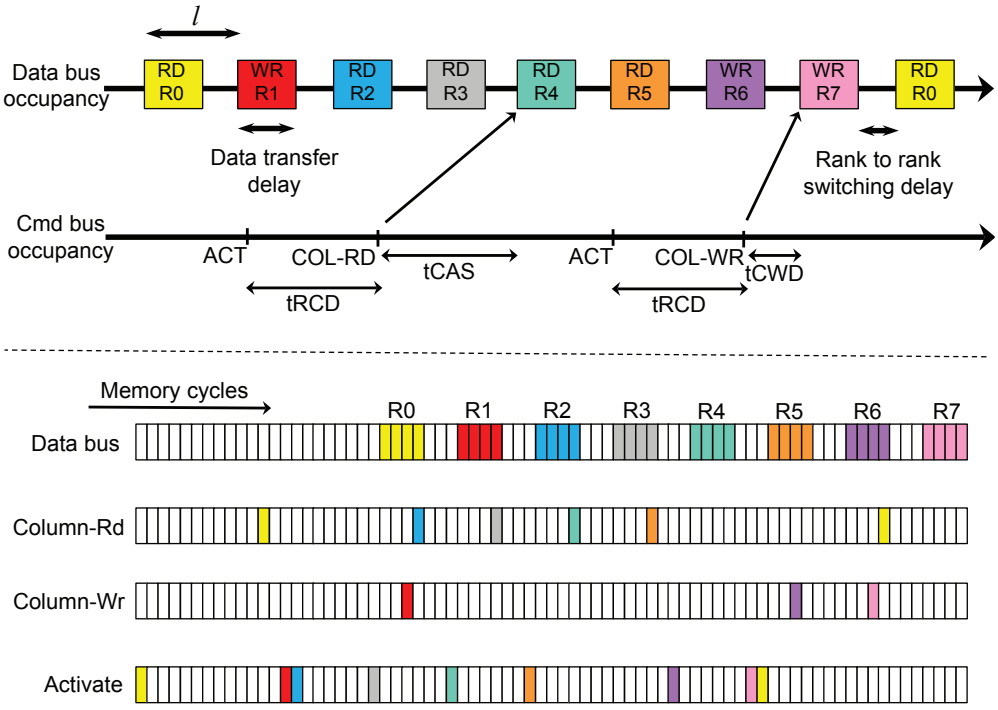

Figure 1: Timing diagrams for eight memory accesses from eight threads to eight different ranks. The top diagram shows the timing parameters involved for each command. The bottom diagram shows the per-cycle command and data bus occupancies in every cycle. Both diagrams use the same color coding scheme for ranks R0-R7. Note that a cycle can only accommodate one of the three commands (Activate, Column-Rd, or Column-Wr). We see that any combination of reads and writes to eight different ranks can be accommodated in 56 memory cycles with this pipeline.

to pick a small enough value for Q that guarantees that the injected memory operation will be serviced before the interval ends. Without this guarantee, the memory controller queue will start to build up and the system will eventually stall. Given the deterministic nature of memory requests from each thread, we can construct a deterministic memory access pipeline that provides high performance and guarantees completion before the end of the interval. This pipeline is shown in Figure 1 for eight threads accessing eight ranks. We will later explain how this pipeline was mathematically determined.

Timing constraints. The first row in the top diagram of Figure 1 shows data bus occupancy as a result of Column-Reads and Column-Writes – in this example, we show six reads and three writes. Each of these operations occupies the data bus for four data bus cycles. The data bus transfers are performed in round-robin fashion from Thread-0 (T0) to Thread-7 (T7), respectively accessing ranks R0 - R7. Because we assume rank partitioning, every consecutive pair of data transfers deals with different ranks. Hence, we insert a delay between consecutive data transfers that is at least as large as the rank-to-rank switching delay ($tRTRS$). The second row in the top diagram (command bus occupancy) shows that each Column-read or Column-write must be preceded by an Activate command by tRCD cycles, which for our simulation parameters is 11 memory cycles. For reads, the Column-Read must happen tCAS cycles (11 cycles) before the transfer on the data bus. For writes, the Column-Write must happen tCWD cycles (5 cycles) before the transfer on the data bus.

Detailed pipeline example. The bottom diagram of Figure 1 shows the details of how the above set of reads and writes move through the memory system. Each rectangle shows resource occupancy in a cycle. The first row shows how each read or write occupies four cycles on the data bus, with three idle cycles between consecutive data transfers. Each data transfer is either preceded by a Column-Rd (11 cycles prior) or a Column-Wr (5 cycles prior). The Column-Rd or Column-Wr are preceded by Activate commands (11 cycles prior). Since the command bus is only asked to carry at most a single command (either Activate, Column-Rd, or Column-Wr) in any cycle, we know that this pipeline will work and we are guaranteed to complete eight memory reads/writes every 56 cycles. The key to achieving this pipeline is the three-cycle gap between consecutive transfers on the data bus. Note that $tRTRS$ (rank-to-rank switching delay, i.e., the minimum gap between data transfers to/from different ranks) is 2 cycles for the DRAM part assumed in our study. But we had to grow this gap to 3 cycles to create a pipeline without any resource conflicts. We next discuss how we mathematically determined this gap.

Equations to encode DRAM timing parameters. We first define l as the uniform gap between successive data transfers (see the very top of Figure 1). We refer to this as *fixed periodic data*. The subsequent equations will solve for l and show that the minimum feasible value of l is 7 cycles. Note that $l \geq tBURST + tRTRS$, i.e., l should be large enough to accommodate the data transfer ($tBURST$) and the rank-to-rank switching de-

lay. For our simulated system, $l \geq 6$. Assume that k and k' refer to the ids of two different data transfers. Since every data transfer is separated by l cycles, the k^{th} data transfer begins in cycle kl.
The preceding Column-Rd (for a read) is in cycle $kl-11$.
The preceding Column-Wr (for a write) is in cycle $kl-5$.
The preceding Activate (for a read) is in cycle $kl-22$.
The preceding Activate (for a write) is in cycle $kl-16$.

A pipeline cannot be constructed if any of these commands (Activate, Column-Rd, Column-Wr) for any two data transfers k and k' happen in the same cycle. In other words, the four possible command bus occupancy times for request k ($kl-11$, $kl-5$, $kl-22$, $kl-16$) should not match the command bus occupancy times for request k' ($k'l-11$, $k'l-5$, $k'l-22$, $k'l-16$). This translates into the following six non-trivial inequalities.

$$\forall k,k' \in \mathbb{Z} \quad kl-22 \neq k'l-16 \Rightarrow \quad (k-k')l \neq 6 \quad (1a)$$
$$\forall k,k' \in \mathbb{Z} \quad kl-22 \neq k'l-11 \Rightarrow \quad (k-k')l \neq 11 \quad (1b)$$
$$\forall k,k' \in \mathbb{Z} \quad kl-22 \neq k'l-5 \Rightarrow \quad (k-k')l \neq 17 \quad (1c)$$
$$\forall k,k' \in \mathbb{Z} \quad kl-16 \neq k'l-11 \Rightarrow \quad (k-k')l \neq 5 \quad (1d)$$
$$\forall k,k' \in \mathbb{Z} \quad kl-16 \neq k'l-5 \Rightarrow \quad (k-k')l \neq 11 \quad (1e)$$
$$\forall k,k' \in \mathbb{Z} \quad kl-11 \neq k'l-5 \Rightarrow \quad (k-k')l \neq 6 \quad (1f)$$

Bottomline. The smallest value of l ($l \geq 6$) that fulfils these equations is 7. Hence, every consecutive pair of 4-cycle data transfers is separated by 3 idle cycles on the data bus. This gives us a minimum value for Q: $7 \times NUMTHREADS$ memory cycles. Thus, in our 8-thread system, a thread can inject a new request into the memory system every 56 memory cycles (224 CPU cycles)[1]. This request is guaranteed to go through the pipeline without any resource conflicts. Not only are we making every thread look alike, we are eliminating the resource conflicts that are the source of information leakage. Since we are eliminating resource conflicts, it is natural to ask if it's necessary to make every thread appear identical. Note that the conflict-free pipeline was only possible because we shaped every thread to perform a single empty-row access to a different rank every Q cycles. Allowing a thread to dynamically inject more/fewer requests into our conflict-free pipeline would reveal if other threads were dynamically using fewer/more than their assigned memory slots. The primary performance penalty of the proposed FS design is the reduction in effective memory bandwidth utilization (57%), with each thread receiving a fair share of this bandwidth. FS also introduces a latency penalty by eliminating row buffer hits and allowing at most one transaction per Q-cycle window.

Fixed periodic commands. We solved our equations assuming *fixed periodic data*. However, we could have also assumed a *fixed periodic RAS* (Activate), or a *fixed periodic CAS* (Column-Rd/Wr). Had we solved the equations with either assumption, we would have arrived at an $l = 12$. Thus, the most efficient pipeline is constructed by assuming *fixed periodic data*. The asymmetry in these equation solutions is because of the different command sequences for reads and writes.

Improving bandwidth. We can also perform a similar analysis by assuming that every thread injects N memory transactions every Q cycles. This may result in a more efficient pipeline because the N transactions from a single thread need not be separated by $tRTRS$. We have to solve a larger set of equations (not shown for space reasons) – our analysis shows that for our chosen parameters, this did not result in a more efficient pipeline.

[1] While a request from a thread is guaranteed to be serviced in every 56-cycle interval in our simulations, we note that because of the asymmetry in Read and Write pipelines, the gap between the handling of consecutive memory operations from a thread is actually 50, 56, or 62 memory cycles.

4. SPATIAL PARTITIONING TRADE-OFFS

4.1 Forms of Spatial Partitioning

Varying thread counts mandate varying policies for security and efficiency. We walk through a few scenarios now, while making the following hardware assumptions: a processor with four channels, each channel has eight ranks, each rank has eight banks. We will assume that all threads are being protected, although, it might also be reasonable to design a few secure memory channels to handle security-critical workloads and a few non-secure high-performance channels for other workloads.

Channel partitioning. If the thread count is four or less (an uncommon case in modern multi-core cloud hardware), it is most efficient to map each thread to one or more channels. Since two threads don't share memory resources in this case, there are no timing channels.

Rank partitioning. If the thread count is greater than four but less than 33, then each thread is assigned to at least one of the 32 available ranks in the system. Any channel that services multiple threads will have to employ the FS memory controller policy to eliminate memory timing channels.

Bank partitioning and no partitioning. If the thread count exceeds 32, then multiple threads will necessarily be mapped to the same rank in our example system. The FS policy as described so far will therefore not be effective in eliminating timing channels. We must therefore design a pipeline that is based on bank partitioning (since there are 512 banks in our example system). We also design a pipeline that is based on no partitioning at all – this would be effective in a system with more than 512 threads, or if the OS/hypervisor complexity of spatial partitioning is not desired. The last point is especially noteworthy. In a real cloud setting with tasks/VMs being constantly spawned or idled, the burden of performing spatial partitioning and frequent page migrations may be too high. Spatial partitioning also limits the granularity at which memory capacity is assigned to threads. It is therefore important to construct an efficient pipeline without assuming any spatial partitioning. Later, we also discuss capacity/bandwidth mismatches that must be considered by the OS/hypervisor when assigning resources to threads.

4.2 Bank Partitioning

Assume a memory system with bank partitioning, i.e., a bank is not shared by multiple threads. Simi-

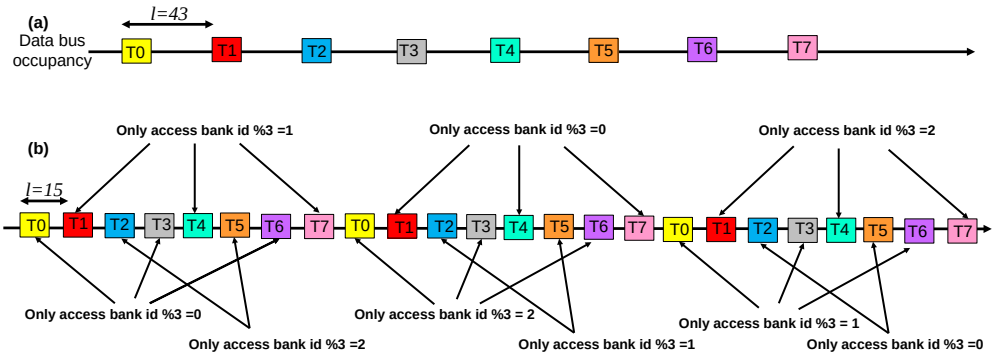

Figure 2: Two pipelines for systems that do not assume any spatial partitioning. The first pipeline is naive and assumes a 43-cycle gap because consecutive requests can go to the same bank. The second pipeline assumes that consecutive requests go to different banks (with triple alternation). This brings the gap between consecutive requests to 15 cycles.

lar to the pipeline in the previous section, all threads in the system inject one read or write into the memory controller in one Q-cycle interval. To keep the discussion consistent, we assume *fixed periodic data*. In the worst case, a number of requests in one interval may be sent to different banks in the same rank. As a result, a few more constraints come into play, such as activation constraints ($tFAW$, $tRRD$), and read-write turnarounds (e.g., $tWTR$). In addition to Equation 1, we have to ensure that the following conditions are met. For our system, $tRRD$ is 5 memory cycles, $tFAW$ is 24 cycles, and $tWTR$ is 6 cycles.

tRRD constraint: There should be at least a 5 cycle gap between two consecutive Activate signals. Equation 2 describes the possible scenarios for the timing for two back-to-back Activates. There are four scenarios depending on whether the two Activates are for reads or writes.

$$if\ k = k' + 1\ then \quad (kl - 22) - (k'l - 16) \geq 5 \Rightarrow \quad l \geq 11 \quad (2a)$$

$$if\ k = k' + 1\ then \quad (kl - 16) - (k'l - 22) \geq 5 \Rightarrow \quad l \geq -1 \quad (2b)$$

$$if\ k = k' + 1\ then \quad (kl - 22) - (k'l - 22) \geq 5 \Rightarrow \quad l \geq 5 \quad (2c)$$

$$if\ k = k' + 1\ then \quad (kl - 16) - (k'l - 16) \geq 5 \Rightarrow \quad l \geq 5 \quad (2d)$$

tFAW constraint: No more than four Activate signals can be sent to a rank in 24 cycles. In other words, the distance between any Activate signal and the fourth Activate after it, should be at least 24 cycles. Equation 3 again represents the four possible scenarios.

$$if\ k = k' + 4\ then \quad (kl - 22) - (k'l - 16) \geq 24 \Rightarrow \quad l \geq 8 \quad (3a)$$

$$if\ k = k' + 4\ then \quad (kl - 16) - (k'l - 22) \geq 24 \Rightarrow \quad l \geq 5 \quad (3b)$$

$$if\ k = k' + 4\ then \quad (kl - 22) - (k'l - 22) \geq 24 \Rightarrow \quad l \geq 6 \quad (3c)$$

$$if\ k = k' + 4\ then \quad (kl - 16) - (k'l - 16) \geq 24 \Rightarrow \quad l \geq 6 \quad (3d)$$

tWTR constraint: If two consecutive Column-Rd/Wr signals to the same rank have the same type, i.e., they are both reads or they are both writes, then the gap between them is $tCCD$. However, if the types are different, then the following two constraints should be considered:

$Rd2Wr\ \ delay = tCAS + tBURST - tCWD = 10$

$Wr2Rd\ \ delay = tCWD + tBURST + tWTR = 15$

Based on these constraints, Equation 4 represents possible scenarios for back-to-back Column-Rd/Wr operations of different types.

$$if\ k = k' + 1\ then\ (kl - 5) - (k'l - 11) \geq 10 \Rightarrow\ l \geq 4 \quad (4a)$$

$$if\ k = k' + 1\ then\ (kl - 11) - (k'l - 5) \geq 15 \Rightarrow\ l \geq 21 \quad (4b)$$

We therefore see that to fulfil these many equations, $l \geq 21$. It turns out that with *fixed periodic RAS*, solving these equations gives an $l \geq 15$ and we arrive at a more efficient pipeline. The length of the best interval is therefore $Q = 15 \times NUMTHREADS$. For our 8-thread case, Q is 120 memory cycles and peak bus utilization is 27%. This basic pipeline is similar to a fine-grained bank-partitioned TP model from prior work [10].

Improving bandwidth. The bandwidth of this model can be improved by allowing every thread to inject N operations in a Q-cycle interval. Multiple requests from a thread can be issued before finally having a 15-cycle gap and switching to the next thread. This is similar to a coarse-grained bank-partitioned TP model and as we show later, turns out to be less effective.

Reordered bank partitioning. We consider an optimization to this design that is based on the observation that the pipeline is primarily limited by the write-to-read delay constraint. We overcome this limitation by re-ordering the reads and writes within a Q-cycle interval. All threads inject their memory transactions at the start of the interval. We first perform all the reads, followed by all the writes. Every back-to-back data transfer is separated by 6 cycles. After the very last write, a 15-cycle gap is introduced before (read) data transfers from the next Q-cycle interval. The value of Q is therefore 63 cycles, and effective bus utilization nearly doubles to 51%. However, such re-ordering may leak some information about the read-write ratios of co-scheduled threads. For example, if a thread runs with write-intensive threads, it'll see faster read latencies than if it ran with read-intensive threads. To prevent this, we must ensure that the results of all read operations in a Q-cycle interval are returned to the processor en masse at the end of the interval.

4.3 No Partitioning

Basic Pipeline. Next, if we attempt to avoid any kind of spatial partitioning (a very desireable design point

from an OS/hypervisor perspective), we see that the length of the interval may be much higher. In the worst case, all threads in a Q-cycle interval may activate different rows in the same bank. The largest gap between two transactions would involve a write followed by a read to different rows in the same bank. With *fixed periodic RAS*, this gives us the best $l = 43$ cycles. For an 8-thread system, this amounts to an interval length of 344 memory cycles and a memory bandwidth utilization of only 9%. This design is similar to the fine-grained no-partitioned TP model. A coarse-grained model is also possible by allowing a thread to issue multiple requests as long as a 43-cycle gap is introduced between requests from different threads.

Triple Alternation Optimization. To address the poor performance with no partitioning, we introduce an optimized pipeline, illustrated in Figure 2 for 8 threads. As seen in this figure, we construct three $Q/3$-cycle intervals. In the first interval, threads 0, 3, and 6 are allowed to access banks that are multiples of three; threads 1, 4, and 7 are allowed to access banks that are multiples of three plus one; threads 2 and 5 are allowed to access banks that are multiples of three plus two. In the next interval, threads 0, 3, 6 are allowed to access banks that are multiples of three plus two, and so on. If we just consider thread 0 and 1 (or any two consecutive threads), we know that they are essentially bank-partitioned, i.e., they never touch the same bank in consecutive accesses. So it is safe to separate their Activations by 15 cycles (see the earlier discussion on bank partitioning). Once this is done, we see that thread 0 and thread 3 are now separated by 45 cycles. Since thread 0 and thread 3 may touch the same bank, their accesses must be separated by at least 43 cycles, a condition guaranteed by the 45-cycle separation. This pipeline makes it safe for groups of threads to access the same bank. Unfortunately, we can't use the read-write re-ordering optimization from the previous sub-section to further reduce the gap between consecutive memory accesses – such re-orderings can upset the triple alternation alignment of memory accesses.

With this optimized pipeline, in 360 memory cycles, every thread is guaranteed service of its next memory request. But in practice, a thread may be able to service three memory requests in that 360 cycle interval. In fact, the transaction scheduler, instead of using FCFS, can look for transactions headed to appropriate banks, depending on the interval. The triple alternation optimization improves effective bandwidth from 9% to 27%.

4.4 Summary

Figure 3 shows a synopsis of the relevant design points discussed in this paper and in prior work. The figure also gives a preview of the performance numbers discussed in Section 7. It highlights the trade-offs involved in spatial partitioning, and the contributions, relative to prior work.

5. DESIGN DETAILS AND OPTIMIZATIONS

5.1 Hardware/Software Changes

Baseline Microarchitecture. In a non-secure baseline, read and write requests emanating from the LLC are placed in transaction queues at the memory controller. The physical organization of the transaction queues is typically designed to match the scheduling algorithm; for example, there may be separate queues for reads and writes. When a transaction queue fills up, back-pressure is applied and processors are stalled. A non-trivial scheduling algorithm, e.g., [27], that considers row buffer hits, thread priorities, read/write priorities, thread memory intensities, etc., is used to pick the next transaction from these queues. The selected transaction (Read X or Write Y) is then broken into its constituent commands (Activate, Column-read/write) and placed in the command queues. The command queues are logically organized as per-bank structures. In every cycle, each bank looks at its oldest command and flags it as ready if all timing constraints are fulfilled. A fair arbiter then selects one of multiple ready commands for issue. When data returns from the memory system, the MSHRs are updated to convey ready data to the processors/LLC. Because the memory controller resources are shared by all threads, resource conflicts manifest as timing channels between threads.

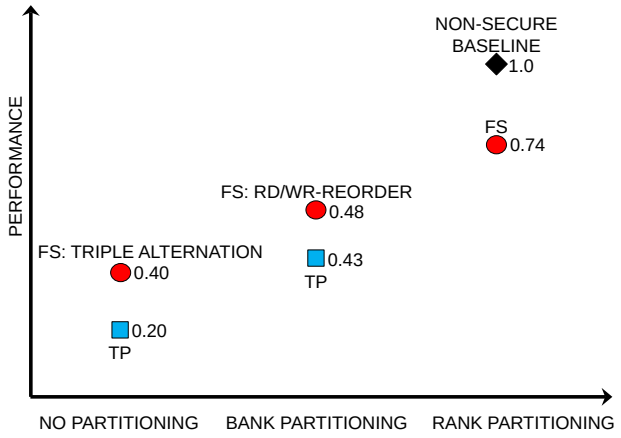

Figure 3: Summary of baseline, prior work (TP), and new FS design points.

Proposed Microarchitecture. To keep the discussion simple, we have used the term "thread" throughout. In reality, we want to isolate "security domains" or VMs. When a cache line request shows up at the memory controller, it carries a tag that indicates its security domain. Such a tag would be required in any comprehensive effort to eliminate timing channels in the cache, NoC, etc. Our proposed design maintains separate transaction queues per security domain; the arriving memory transaction's tag indicates the queue that it should be placed in. The OS prevents users from creating more security domains than the number of transaction queues in the memory controller. Each transaction queue receives a fixed level of service, as determined by the OS and a service-level agreement (SLA). If the queue is full, the corresponding security domain is stalled; this does not create a timing channel because the stall is a result of high activity by that domain, not a result of activ-

ity in other domains. Based on the number of active domains, the service levels for each, and the use of any spatial partitioning, the transaction scheduler computes the values of Q and l. Every l cycles, it picks the head of the appropriate transaction queue (or a dummy operation) and inserts the constituent commands into the command queues. Even in its most complicated form, the transaction scheduler only has to scan the entries in one queue to look for a transaction that meets specific criteria (e.g., dealing with a specific group of banks in triple alternation). Once the right commands are inserted into the command queues at the right time, the rest of the memory controller logic remains unchanged. The commands end up being scheduled in a deterministic order with no resource conflicts.

Software Changes. In a secure cloud setting, the OS/hypervisor assigns resources, including memory capacity and bandwidth, to security domains based on the SLA. The bandwidth influences the number of transactions that a security domain can inject in an interval. The capacity influences the number of ranks or banks that are assigned to that domain under spatial partitioning. For example, under rank partitioning, if a domain receives four ranks and two issue slots, in every Q-cycle interval, the memory controller will select two different transactions for that domain accessing different ranks. If a domain receives two ranks and four issue slots, the accesses in an interval can't all be to different ranks, so the memory controller employs a bank-partitioned schedule – in every Q-cycle interval, four different transactions are selected for that domain accessing different banks. The OS has to communicate the mapping of domains to ranks and banks to the memory controller so that requests can be correctly routed to the appropriate transaction queue.

Security Invariant. The FS protocol enforces non-interference across security domains by a) mapping each transaction queue to a unique security domain, and b) maintaining dedicated logic for each transaction queue. All other optimizations – that provide performance improvements over TP – are due to (offline) constraint solving by the trusted OS-level component to yield a deterministic schedule for issuing memory transactions. Further design points spatially separate the ranks or banks assigned to a security domain.

5.2 Dummy Operations

When it is a thread's turn to issue a memory operation and the thread has no pending operations, the memory controller inserts a dummy read or write on behalf of that thread. A dummy operation can therefore be a read request to a random address within the rank and the returned value is simply discarded. A side-effect of this is that every thread also has a constant memory energy/power requirement. The proposed memory system is therefore also resilient to physical attacks that are based on energy/power measurements.

Performance optimization: Prefetches: One way to use the dummy operation to do useful work is to use that slot to issue a prefetch operation for that thread. We use the sandbox prefetcher [28] to generate a few high-confidence prefetch instructions that are issued when there aren't any pending memory accesses.

If there is no fear of physical attacks based on energy measurements, then the following three energy optimizations are possible.

Energy optimization 1: Suppressed reads/writes: When the command scheduler encounters a dummy operation, it does not actually issue commands to the memory system. It simply updates timing parameters and DRAM state as if the command had issued, i.e., the actual memory read or write is suppressed.

Energy optimization 2: Boosting row buffer hits: Similarly, if the transaction scheduler detects the possibility for a row buffer hit and communicates this to the command scheduler, the latter can avoid the issue of the auto-Precharge and subsequent Activate. Again, DRAM state is updated as if these commands had issued.

Energy optimization 3: Power-down states: Another possibility is to power-down a rank instead of issuing a dummy operation. The deep-sleep states have wake-up times that exceed 56 memory cycles, but some of the lighter power-down modes have transition latencies of 10 memory cycles [29]. Therefore, if there are no pending requests to a rank at the start of an interval, the memory controller can issue a power-down command to that rank. The power-up command is issued 5 cycles before the end of the interval, e.g., the memory access pipeline in Figure 1 shows that the command bus is free to transmit the power-down signal in that cycle.

6. METHODOLOGY

For our simulations, we use Windriver Simics [30] interfaced with the USIMM memory simulator [31]. USIMM is configured to model a DDR3 memory system. Simics and USIMM parameters are summarized in Table 1. Our baseline non-secure scheduler is an optimized form of FR-FCFS [17] that re-orders requests to exploit row buffer hits, closes rows when there aren't any pending operations to that row, and uses high/low water marks to drain writes. This scheduler performs within 5% of the best performing scheduler from the 2012 Memory Scheduling Championship.

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 [33] 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.

For our memory energy analysis, we use the Micron power calculator for a DDR3 4 Gb part [34]. The calculator is fed with memory statistics collected during

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 timing [32] parameters.

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.

7. RESULTS

Information Leakage Analysis

In Figure 4, we show four execution profiles for workload *mcf*. Every point on the X-axis represents 10K instructions and the Y-axis represents the time taken to complete that many instructions. The red curve shows the progress made by *mcf* with a non-secure baseline memory controller when running with 7 synthetic threads that make no memory accesses. The blue curve shows the progress made by *mcf* with a non-secure baseline memory controller when running with 7 synthetic threads that are highly memory-intensive. Given the divergence in the two curves, the attacker (represented by workload *mcf*) can decipher the memory intensity of its co-scheduled threads. The black and green curves represent *mcf* running on the proposed FS scheduler when running with non-memory-intensive and memory-intensive threads respectively. The black and green curves overlap perfectly because FS offers deterministic execution to *mcf* regardless of the nature of the co-scheduled threads. This graph is simply a visual illustration of the performance trade-off and the zero information leakage that was mathematically demonstrated in the previous sections.

Analyzing TP

Our analyses in the previous sections describe a generalized way of constructing efficient memory pipelines in a variety of settings. The models in prior TP work are special cases – they resemble the basic bank-partitioned and no-partitioned pipelines.

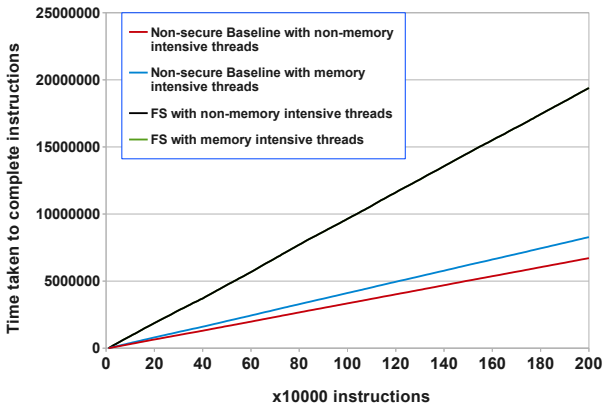

Figure 4: Execution profiles for benchmark *mcf* with and without the FS scheduler.

We first analyze the behavior of TP for 8 threads and 8 ranks as turn length is varied. We assume bank-partitioned and no-partitioned models and show the sum of weighted IPCs in Figure 5. Sum of weighted IPCs for each model is the summation of normalized IPCs for each thread, where the normalization is against the IPC of that thread using a baseline non-secure scheduler. We consider minimum sized turn lengths and a few larger turn lengths. Short turn lengths in TP can reduce wait times, while longer turn lengths improve bandwidth utilization. We see that minimum turn lengths are best in both cases (except for GemsFDTD) because reducing wait times is far more important than improving bandwidth for our workloads on average. The best TP model with bank partitioning has an average memory latency of 683 cycles, theoretical peak bandwidth of 27%, and actual average bandwidth of 17%. Its performance is 57% lower than that of the non-secure baseline.

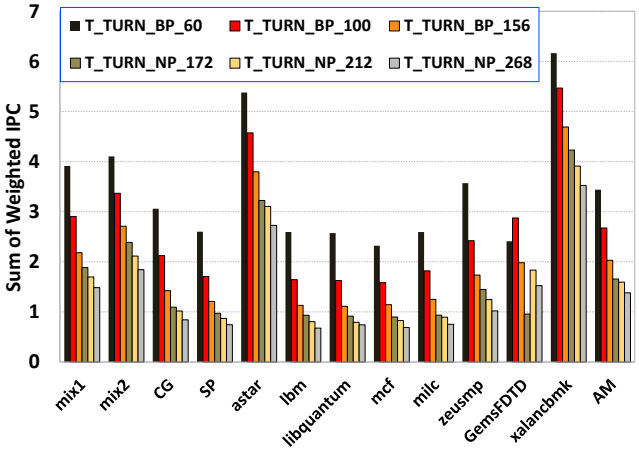

Figure 5: Sum of weighted IPCs for bank-partitioned and no-partitioned TP with varying turn lengths and 8 threads. The non-secure baseline scheduler would have a throughput of 8 with this metric.

FS Performance

Next, we show performance for other optimized FS techniques in Figure 6. The results also show the best TP designs. We see that FS with rank-partitioning yields a 70.1% improvement and FS with re-ordered bank-partitioning yields a 11.3% improvement, relative to the best bank-partitioned TP design. The FS triple alternation method with no partitioning improves performance by 2×, relative to the best TP approach with no partitioning.

The best FS design has an average sum of normalized IPCs that is 26.4% lower than that of the non-secure baseline. This design has an average memory access latency of 288 cycles and a theoretical bandwidth of 57%. 36% of all memory transactions are dummy requests, so the effective bandwidth utilization is 37%.

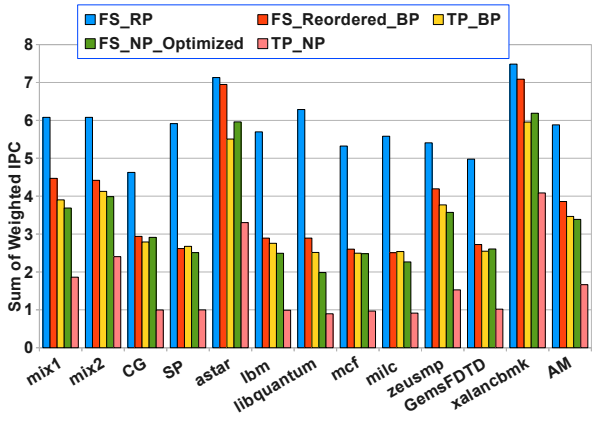

Figure 6: Sum of weighted IPCs for FS and TP with 8 threads.

Prefetch Technique

Figure 7 shows the rank-partitioned FS design with 8 threads, with and without the prefetch optimization. The figure also shows the non-secure baseline with prefetch added. In the baseline, 42.4% of all memory accesses are prefetches and the resulting performance improvement is 7.2%. In FS with the prefetch technique, we see that 13.4% of all memory accesses are prefetch operations and 43.7% of these prefetches prove to be useful. The technique improves the performance of FS by 9.8% on average.

Energy Analysis

Figure 8 shows memory energy for the non-secure baseline, the three FS schemes and the two TP models. The memory energy for the baseline is clearly superior because it has the lowest execution time, fewest memory accesses, and most row buffer hits. FS is able to out-do TP primarily because of its significantly lower execution time even though it issues 36.6% more memory accesses (dummy operations). FS has an energy consumption that is 11.4% lower than that of TP and that is within 17.5% of the non-secure baseline.

Figure 9 shows the energy reductions with the three energy optimizations for rank-partitioned FS. While the first two optimizations are trying to recover some of the losses introduced by FS, the third optimization is a new opportunity created by FS – a deterministic pipeline is amenable to power-down strategies that have zero impact on performance. These optimizations collectively reduce memory energy of FS by 52.5% and this optimized version of FS has a memory energy dissipation that is within 3.4% of the non-secure baseline.

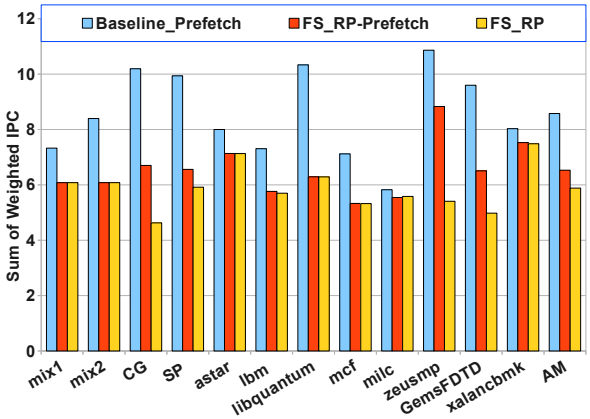

Figure 7: Sum of weighted IPCs for FS with 8 threads and rank-partitioning, with and without the prefetch optimization.

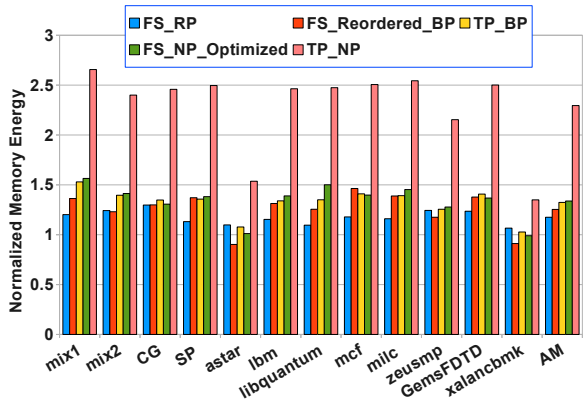

Figure 8: Memory energy for the baseline and various FS and TP schemes.

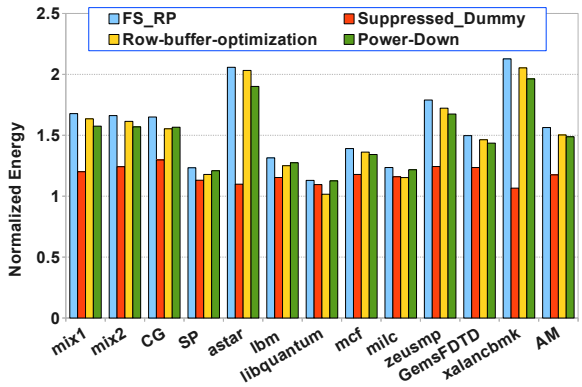

Figure 9: Memory energy for rank-partitioned FS with three energy optimizations.

Sensitivity Analysis

In Figure 10, we show the scalability of bank-partitioned TP and bank/rank-partitioned FS as thread count varies (we assume as many ranks as threads). In TP and

FS, the memory latencies scale roughly linearly with core count. However, at low core counts, the rank-partitioned FS model suffers from the following phenomenon. In the worst case, DRAM timing parameters dictate that two memory transactions to the same rank be separated by 43 cycles (a write followed by a read to a different row in the same bank). With rank partitioning, if the number of threads and ranks in the system is six or lower, a thread may issue two back-to-back memory transactions to the same rank with a gap of 42 cycles or less, thus potentially violating the above constraint. In such cases, the transaction scheduler may have to pick a different transaction from the same thread (to a different bank in the same rank) or insert a dummy operation. Note that this is a case of intra-thread contention, i.e., such non-determinism does not introduce a timing channel between two threads. This phenomenon may limit some of the throughput of the FS pipeline at low thread counts. Even with these effects considered, the FS models out-perform the TP models significantly in the 4-thread (85% improvement) and 2-thread (18% improvement) cases.

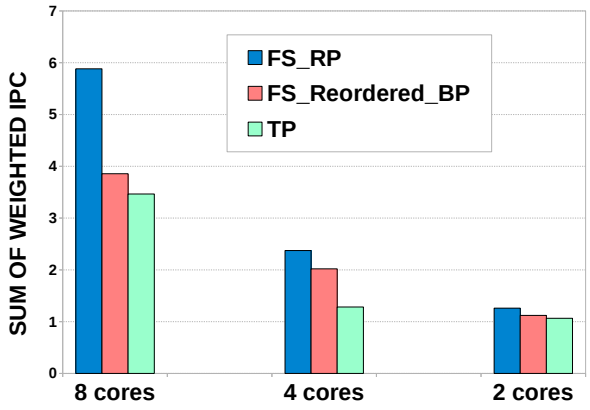

Figure 10: Sum of weighted IPC for rank/bank-partitioned FS and bank-partitioned TP. The threads and ranks are set to 2, 4, and 8.

8. RELATED WORK

In addition to Wang et al. [10], a few other papers have tried to eliminate timing channels in caches and on-chip networks [11, 12, 13, 14, 15, 16]. Note that techniques such as cache partitioning can isolate the cache activity of one thread from another, but it does not eliminate the memory timing channels, i.e., memory latencies of one thread are impacted by the cache miss rate of the co-scheduled thread. Martin et al. [35] thwart timing channel attacks by limiting a user's ability to take fine-grained timing measurements. Saltaformaggio et al. [6] identify potential attacks from atomic instructions that can lock up the memory system; they develop solutions that require hypervisor extensions.

Virtually pipelined network memory [36] (VPNM) is a secure memory system that hides information leaks among threads that contend for memory banks in network processors. VPNM is specifically proposed for router architectures, where attackers can increase packet service times and cause packets to be dropped. VPNM scatters packets randomly over memory banks to avoid prolonged contention for a bank, and to achieve acceptable queuing latency for routers (1000 ns). Our approach, however, is optimized for desktops and servers, with a focus on low-latency memory access and timing channel based attacks. Probabilistic approaches are not as effective to deal with such attacks.

Reineke et al. [37] build a DRAM controller with predictable latencies. Refreshes are broken into smaller RAS-only refresh operations to prevent disruptions from a long refresh operation. The memory controller also uses a close page policy and forcibly goes through bank groups in round-robin fashion. But a co-scheduled attacker thread can estimate the memory intensity of a victim thread because queuing delays at banks will be variable. CCSP arbitration [38] requires assigning priorities and bandwidth requirements, and regulates rate of requests to ensure bounded latency.

Wassel et al. [16] propose SurfNoC, an on-chip network that reduces the latency incurred by temporal partitioning. SurfNoC scheduling follows from dimension-ordered routing. In this approach, a packet does not experience any stall unless it changes dimension. To acheive this, SurfNoC modifies router packet scheduling and relies on static virtual channel partitioning.

A few papers [39, 40, 41, 42] have used memory bandwidth reservation to aid QoS policies in datacenters. QoS policies and timing channel prevention policies differ in two ways: (i) QoS policies allow allocations to change based on need, and (ii) QoS policies allow a thread to steal idle resources from another thread, thus betraying information about co-scheduled threads.

9. CONCLUSIONS

Our paper develops a general framework for constructing deterministic high-throughput memory pipelines that eliminate contention among threads for memory resources. It thus offers zero information leakage and high performance under a variety of scenarios. The best FS model yields a performance degradation of 26.4%, relative to a non-secure baseline. It out-performs the best-known competing approach (TP) by 70%. Even with no OS support, the proposed triple alternation FS approach out-performs TP by 2×. The key to the new design is efficient pipelining of requests from different threads while avoiding problematic DRAM timing constraints.

10. ACKNOWLEDGEMENTS

This work was supported in part by Intel and by NSF grants 1303663, 1423583, and 1314709.

11. REFERENCES

[1] O. Aciiçmez, "Yet Another Microarchitectural Attack: Exploiting I-cache," in *Proceedings of the 2007 ACM workshop on Computer Security Architecture*, pp. 11–18, 2007.

[2] O. Aciiçmez, Ç. K. Koç, and J.-P. Seifert, "On the Power of Simple Branch Prediction Analysis," in *Proceedings of the 2nd ACM symposium on Information, Computer and Communications Security*, pp. 312–320, 2007.

[3] O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert, "Predicting Secret Keys via Branch Prediction," in *Topics in Cryptology–CT-RSA 2007*, pp. 225–242, Springer, 2006.

[4] D. J. Bernstein, "Cache-timing Attacks on AES," 2005.

[5] C. Percival, "Cache Missing for Fun and Profit," 2005.

[6] B. Saltaformaggio, D. Xu, and X. Zhang, "BusMonitor: A Hypervisor-Based Solution for Memory Bus Covert Channels," in *Proceedings of EuroSec*, 2013.

[7] Z. Wang and R. B. Lee, "Covert and Side Channels Due to Processor Architecture," in *Computer Security Applications Conference, 2006 (ACSAC '06)*, pp. 473–482, 2006.

[8] Z. Wu, Z. Xu, and H. Wang, "Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud," in *the 21st USENIX Security Symposium (Security '12)*, 2012.

[9] 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*, pp. 199–212, 2009.

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

[11] J. Kong, O. Aciiçmez, J.-P. Seifert, and H. Zhou, "Hardware-software Integrated Approaches to Defend Against Software Cache-based Side Channel Attacks," in *Proceedings of HPCA*, pp. 393–404, 2009.

[12] D. Page, "Partitioned Cache Architecture as a Side-Channel Defence Mechanism," *IACR Cryptology ePrint Archive*, vol. 2005, p. 280, 2005.

[13] Y. Wang and G. E. Suh, "Efficient Timing Channel Protection for On-chip Networks," in *Proceedings of Networks on Chip (NoCS)*, pp. 142–151, 2012.

[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*, pp. 83–93, 2008.

[16] H. M. Wassel, Y. Gao, J. K. Oberg, T. Huffmire, R. Kastner, F. T. Chong, and T. Sherwood, "SurfNoC: A Low Latency and Provably Non-interfering Approach to Secure Networks-on-chip," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pp. 583–594, 2013.

[17] S. Rixner, W. Dally, U. Kapasi, P. Mattson, and J. Owens, "Memory Access Scheduling," in *Proceedings of ISCA*, 2000.

[18] D. E. Denning, "A Lattice Model of Secure Information Flow," *Commun. ACM*, vol. 19, pp. 236–243, May 1976.

[19] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, "Complete Information Flow Tracking from the Gates Up," in *Proceedings of ASPLOS*, 2009.

[20] J. A. Goguen and J. Meseguer, "Security Policies and Security Models," in *Proceedings of IEEE Symposium on Security and Privacy (Oakland)*, 1982.

[21] M. Tiwari, X. Li, H. M. G. Wassel, F. T. Chong, and T. Sherwood, "Execution leases: A hardware-supported mechanism for enforcing strong non-interference," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, (New York, NY, USA), pp. 493–504, ACM, 2009.

[22] 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.

[23] D. of Defense, *TCSEC: Trusted Computer System Evaluation Criteria. Technical Report 5200.28-STD.* US Department of Defense, 1985.

[24] X. Li, M. Tiwari, J. K. Oberg, V. Kashyap, F. T. Chong, T. Sherwood, and B. Hardekopf, "Caisson: A Hardware Description Language for Secure Information Flow," in *Proceedings of PLDI*, 2011.

[25] X. Li, V. Kashyap, J. K. Oberg, M. Tiwari, V. R. Rajarathinam, R. Kastner, T. Sherwood, B. Hardekopf, and F. T. Chong, "Sapper: A Language for Hardware-Level Security Policy Enforcement," in *Proceedings of ASPLOS*, 2014.

[26] Ed Suh, Yao Wang, Cornell University, *Personal Correspondence*, November 2014.

[27] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," in *Proceedings of MICRO*, 2010.

[28] S. Pugsley, Z. Chishti, C. Wilkerson, T. Chuang, R. Scott, A. Jaleel, S.-L. Lu, K. Chow, and R. Balasubramonian, "Sandbox Prefetching: Safe, Run-Time Evaluation of Aggressive Prefetchers," in *Proceedings of HPCA*, 2014.

[29] Krishna T. Malladi,Ian Shaeffer,Liji Gopalakrishnan,David Lo,Benjamin C. Lee,Mark Horowitz, "Rethinking DRAM Power Modes for Energy Proportionality," 2012.

[30] "Wind River Simics Full System Simulator," 2007. `http://www.windriver.com/products/simics/`.

[31] 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," tech. rep., University of Utah, 2012. UUCS-12-002.

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

[33] 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, pp. 63–73, Fall 1994.

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

[35] R. Martin, J. Demme, and S. Sethumadhavan, "TimeWarp: Rethinking Timekeeping and Performance Monitoring Mechanisms to Mitigate Side-Channel Attacks," in *Proceedings of ISCA*, 2012.

[36] B. Agrawal and T. Sherwood, "High-bandwidth Network Memory System Through Virtual Pipelines," *IEEE/ACM Trans. Netw.*, 2009.

[37] Reineke, Jan and Liu, Isaac and Patel, Hiren D. and Kim, Sungjun and Lee, Edward A., "PRET DRAM Controller: Bank Privatization for Predictability and Temporal Isolation," in *Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '11, 2011.

[38] B. Akesson, L. Steffens, E. Strooisma, and K. Goossens, "Real-time scheduling using credit-controlled static-priority arbitration," in *Proceedings of the 2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '08, 2008.

[39] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith, "Fair Queuing Memory Systems," in *Proceedings of MICRO*, 2006.

[40] N. Rafique, W. Lim, and M. Thottethodi, "Effective Management of DRAM Bandwidth in Multicore Processors," in *Proceedings of PACT*, 2007.

[41] R. Iyer, L. Zhao, F. Guo, R. Illikkal, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt, "QoS Policies and Architecture for Cache/Memory in CMP Platforms," in *Proceedings of SIGMETRICS*, 2007.

[42] K. Sudan, S. Srinivasan, R. Balasubramonian, and R. Iyer, "Optimizing Datacenter Power with Memory System Levers for Guaranteed Quality-of-Service," in *Proceedings of PACT*, 2012.