Lattice Priority Scheduling: Low-Overhead Timing-Channel Protection for a Shared Memory Controller

Andrew Ferraiuolo[1], Yao Wang[1], Danfeng Zhang[2], Andrew C. Myers[1], G. Edward Suh[1]

[1] Cornell University
Ithaca, NY 14850, USA
af433@cornell.edu, yw438@cornell.edu,
andru@cs.cornell.edu, suh@csl.cornell.edu

[2] Penn State University
University Park, PA 16802
zhang@cse.psu.edu

ABSTRACT

Computer hardware is increasingly shared by distrusting parties in platforms such as commercial clouds and web servers. Though hardware sharing is critical for performance and efficiency, this sharing creates timing-channel vulnerabilities in hardware components such as memory controllers and shared memory. Past work on timing-channel protection for memory controllers assumes all parties are mutually distrusting and require timing-channel protection. This assumption limits the capability of the memory controller to allocate resources effectively, and causes severe performance penalties. Further, the assumption that all entities are mutually distrusting is often a poor fit for the security needs of real systems. Often, some entities do not require timing-channel protection or trust others with information. We propose lattice priority scheduling (LPS), a secure memory scheduling algorithm that improves performance by more precisely meeting the target system's security requirements, expressed as a lattice policy. We evaluate LPS in a simulated 8-core microprocessor. Compared to prior solutions [34], lattice priority scheduling improves system throughput by over 30% on average and by up to 84% for some workloads.

1. INTRODUCTION

Hardware has become increasingly shared among distrusting parties. Cloud services like Amazon EC2 lease virtual machines which run on the same hardware. A great deal of trust is placed on the assumption that these VMs are completely isolated and secrets cannot be leaked between them. Providing such isolation has been a major focus of many research efforts [43, 30, 17, 9, 39, 6, 19]. Unfortunately, hardware timing channels subvert the isolation provided by virtual machines, virtual memory, and access controls.

Timing channels are not merely a speculative research problem — timing channels in the memory hierarchy have been exploited in production EC2 servers [27]. Hunger et al. [15] find that a covert timing channel caused by accesses to main memory has a capacity of 500Kbps, which is greater than the capacity of timing channels caused by caches and branch predictors. Therefore, timing channels caused by memory accesses are among the most attractive microarchitectural timing-channel vulnerabilities for attackers to exploit.

Existing approaches to remove timing channels in memory controllers are costly. Temporal Partitioning (TP) [34] addresses timing channels in shared memory controllers. It supports a security model which assumes all entities are mutually distrusting, and closes all timing channels among these entities. By enforcing this model in a system with 8 cores, TP increases memory latency by 5.39x compared to that of the baseline on average, and reduces system throughput by up to 80%. Fixed Service (FS) [29] memory scheduling improves upon TP with several optimizations. However, it still assumes a security model in which entities are mutually distrusting, and can have significant overheads.

Our insight is that for many systems, complete isolation is not only unnecessary, but also hurts performance. For example, consider a cloud computing service which leases out a mix of low-assurance and high-assurance VMs, and charges a higher cost for high-assurance VMs. Although preventing information leakage from a high-assurance VM is necessary, there is no reason to prevent leakage from a low-assurance VM. TP enforces security conservatively by preventing all leakage. However, in practice this conservative model limits performance.

We propose a new memory scheduling algorithm called *lattice priority scheduling* (LPS), which significantly reduces the overhead of timing-channel protection for main memory by precisely enforcing the security needs of the target system. This new memory controller enforces a security model in which some information flows between processes are permitted. Applied to the previously mentioned cloud computing scenario, this algorithm would prevent leakage from the high-assurance VMs but not from the low-assurance VMs. In doing so, it can find a more efficient schedule, thereby improving performance while meeting the security needs of the system just as well as prior, more restrictive approaches.

Lattice priority scheduling enforces policies described in the lattice model of security [11]. In the lattice model, rather than treating all entities as mutually distrusting, protection can be applied in just one direction between some entities. That is, timing channels can be prevented from entity A to entity B, but not from B to A. The lattice model has been widely adapted in programming language [5] and software system security [39] domains since it is highly expressive and can capture the security needs of many systems.

LPS enforces policies in the lattice model precisely, permitting scheduling decisions to be made based on run-time program behavior. For example, if timing information can flow from B to A, LPS can consider the dynamic resource utilization of B when scheduling requests from A. In past

978-1-4673-9211-2/16/$31.00 ©2016 IEEE

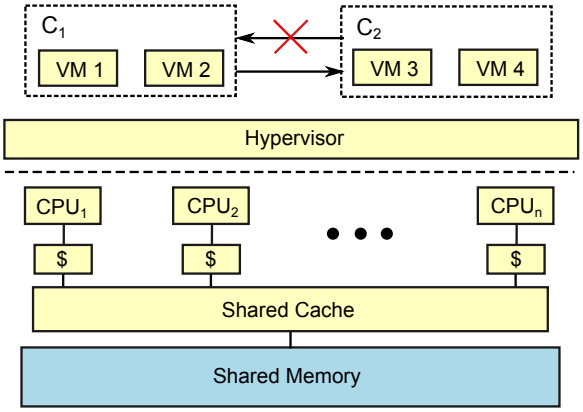


Figure 1: System model

approaches which enforce total isolation, such dynamic information cannot be used, because doing so would leak information from B to A.

By responding to run-time program behavior, LPS improves performance compared to prior approaches significantly. Simulations of an 8-core out-of-order processor show that lattice priority scheduling increases system throughput by up to 84% for some workloads and by 30% on average, compared to TP.

This work is the first to describe an algorithm for allocating a shared, microarchitectural resource among entities while providing timing-channel protection under a security policy expressed in the lattice model. It does so precisely, leveraging permissible flows to improve the efficiency of allocation decisions. The lattice policy is highly expressive, and the proposed algorithms support the full generality of the lattice model.

Section 2 presents the target system and threat model, and discusses timing channels in main memory. Section 3 presents lattice priority scheduling. Section 4 provides background on the lattice model of security. Section 5 generalizes lattice priority scheduling to support arbitrary lattice model policies. Section 6 discusses how LPS is implemented in hardware. Section 7 evaluates LPS. Related work is discussed in Section 8, and finally, the paper concludes with Section 9.

2. MAIN-MEMORY TIMING CHANNELS

2.1 System Model

This work considers a multi-core processor with two or more cores connected to a shared memory as shown in Figure 1. The cores may also share caches, on-chip networks, and other hardware components. The architecture allows processes to be grouped into security classes according to their security needs. In Figure 1, the class C_1 does not require timing protection from C_2. However, C_2 distrusts C_1, so timing channels that leak information from C_2 to C_1 must be prevented. As discussed in Section 4, lattice priority scheduling supports a wide range of policies describing trust relationships of this form.

This work assumes the target system has a conventional

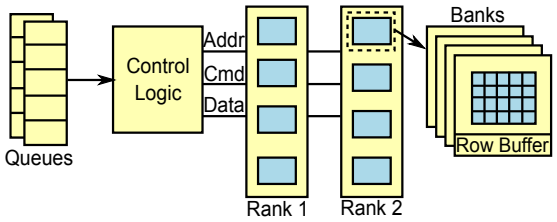


Figure 2: A Conventional DRAM Channel

DRAM and memory controller architecture. A modern processor has multiple memory channels, the structure of which is shown in Figure 2. Each channel is managed by a separate memory controller. Each memory controller has a set of queues of pending memory transactions (read or write requests) and control logic which governs the use of the address, command, and data buses. A DRAM channel is divided into several sets of chips, called ranks, that work in unison to handle each memory transaction. Ranks are further divided into banks. Each bank has an array of DRAM cells which are broken into rows and columns, and each bank has a row buffer that stores the most recently used row. Banks and ranks both improve the parallelism of main memory.

2.2 Threat Model

Lattice priority scheduling removes all timing channels that are introduced when a group of coresident processes share main memory. In particular, this work addresses timing *side channels*, as well as timing *covert channels*. In a side-channel attack, a victim unintentionally leaks a secret to the attacker through timing. For example, Wang et al. [34] present a side-channel attack in which the number of "1s" in a private RSA key is leaked through shared memory traffic. Timing channels enable covert-channel attacks in which one attacker intentionally communicates a secret to another attacker through event timing to bypass a communication restriction. Wang et al. [34] also present a covert-channel attack where two attackers share a memory. One attacker sends a message by modulating its memory demand. The other attacker issues a large number of memory requests (which interfere with the first attacker's requests), and then measures its memory throughput to receive the message.

LPS addresses a threat model which includes attackers that can run arbitrary programs on the target system and can measure the timing of their own events (e.g., program execution time). The scheduling algorithm is assumed to be public and known to the attacker. Attackers can leverage this information to improve their ability to correlate scheduling decisions with secrets. The threat model includes sophisticated attackers capable of filtering out noise and performing statistical analysis when carrying out both covert and side-channel attacks.

It is assumed that there is adequate protection for explicit communication (such as virtual memory and access controls). The attackers lack physical access to the target system, and therefore cannot execute physical side-channel attacks, such as those which exploit power side channels.

2.3 Timing-Channel Attacks in Memory

Conventional memory controllers have timing-channel vulnerabilities due to 1) queue interference, 2) row buffer state, and 3) contention for DRAM resources [34]. In conventional memory controllers, transactions from distrusting processes are placed in a shared queue, where they can interfere, causing measurable delays. Memory banks store the most recently used row in a row buffer for faster access. An attacker can learn that a particular row was used recently if it gets a row buffer hit. Finally, DRAM devices have a number of resources (such as ranks, banks, and the address, command, and data buses) which can service a finite number of simultaneous requests. Contention for these resources also causes timing channels.

2.4 Temporal Partitioning

Temporal Partitioning (TP) [34] addresses timing channels in main memory. Fixed Service (FS) [29] improves upon TP, but uses the same high-level approach. This approach prevents all timing leakage between *security classes*, which are groups of processes or virtual machines. Memory transactions are tagged to indicate the security class that owns them. Queue interference is removed by providing separate queues for each security class or statically partitioning a shared queue. The row-buffer timing channel is addressed by using a closed page policy, which precharges the row buffer after each read or write command to clear the buffer. The secure memory controller presented in this paper also uses duplicated/partitioned queues and a closed page policy to address these problems.

TP addresses the contention timing channel through time-division multiplexing. Memory transactions are issued on a fixed, static schedule. Each security class is given a *turn*, which is a time slot in the static schedule during which it is permitted to issue requests, as illustrated in Figure 3. Each of the three security classes is allocated a turn in the static schedule. The duration of a turn can be configured to improve performance. The security class currently scheduled with a turn is said to be *active*. If the active security class has no useful work, the turn is wasted. No other security class can use the turn since this would indicate the memory usage of the originally scheduled security class.

Memory transactions require a variable number of cycles to complete. Since the presence of an in-flight transaction from one class could influence the timing of transactions owned by another, any in-flight transactions must be drained before the turn for the next class starts. This is done using a period at the end of the turn called *dead time*, which is long enough to drain the worst-case memory transaction. During dead time, the turn owner can no longer issue transactions. The turn length must be at least as long as dead time. Dead time is indicated in Figure 3 by a dark gray segment at the end of each turn.

2.4.1 Performance of Temporal Partitioning

Unfortunately, Temporal Partitioning is costly. With 8 cores, TP increases the memory latency by 5.39x compared to the baseline on average, and reduces system throughput (STP) by up to 80%. Additionally, we simulated a 4-core system with timing-channel protection mechanisms applied

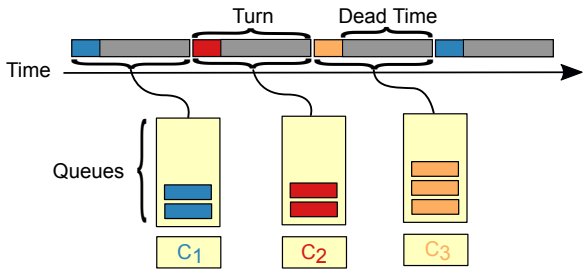


Figure 3: A temporal partitioning schedule with three security classes.

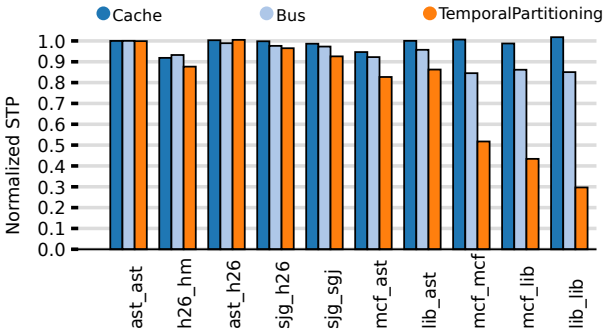


Figure 4: System throughput with off-core protection mechanisms normalized to throughput of insecure baseline.

to all major off-core resources including the shared cache, the interconnects, and the memory controller. The shared caches are statically partitioned, the interconnects are time-multiplexed with a fixed, round-robin schedule, and TP is used to protect the memory controller. Figure 4 shows the system throughput with each of these protection mechanisms enabled individually. The throughput is normalized to the system throughput of the same system with all protection mechanisms disabled. Temporal Partitioning is the greatest source of performance overhead and accounts for over 81% of the total overhead of the system. This implies that memory controller protection is the performance bottleneck for systems that remove timing channels through shared off-core hardware resources.

In TP, time multiplexing is the primary source of performance overhead. A transaction can only be issued during the turn of its security class, but not during dead time. Transactions that arrive outside this period must wait in the queue. Therefore, TP increases *queueing delay*: the part of memory latency transaction waits in the queue. The other component of memory latency is *in-flight time*, which is the time from when the transaction issues from the queue until it completes. TP increases queueing delay in several ways.

Dead time reduces the total usable bandwidth compared to a conventional memory controller, increasing queueing delay. With the DRAM timing parameters used in this paper, each turn must include 43 cycles of dead time. If the minimum turn length is used, transactions are issued at a rate of at most one per 44 memory cycles. Though the turn length can be increased to improve the bandwidth, the bandwidth-latency tradeoff favors shorter turns. Increasing the turn length increases the latency imposed on transactions issued

by a security class other than the active one. Prior studies [34, 29] confirm that shorter turn lengths achieve the best performance.

In addition to reducing the total usable bandwidth, scheduling constraints restrict TP from efficiently allocating memory bandwidth among security classes. If the active security class has no pending transactions, no other security class can be scheduled in its place as this would leak the demand of the originally scheduled class. Instead, no transactions are issued and memory bandwidth is wasted. Generally, this means that under the security model of TP, the scheduler cannot respond to the dynamic resource needs of each security class leading to performance overhead. We call this problem *demand imprecision*.

Even if bandwidth is allocated to security classes proportionally to the memory demand from each security class, static scheduling still imposes restrictions which lead to delays. If a transaction from one security class is enqueued when a different security class is active, it is delayed until its security class becomes active.

3. LATTICE PRIORITY SCHEDULING

This section proposes a secure scheduling algorithm, called *lattice priority scheduling* (LPS), that enables a timing-safe memory controller to precisely meet the security requirements of the system, thereby improving performance. LPS improves performance by enforcing security policies which include uni-directional protection – in other words, policies which allow information to flow in just one direction between security classes. LPS leverages uni-directional protection to address demand imprecision, remove delays due to static scheduling, and reduce dead time. For simplicity, lattice priority scheduling is first introduced for a system with two security classes, L and H. Information is permitted to flow from L to H, but not from H to L. Section 5 generalizes LPS to support arbitrary lattice policies.

Lattice priority scheduling improves upon TP in two ways. First, it schedules security classes dynamically. *Dynamic scheduling* allows LPS to respond to the run-time resource demands of applications, and removes delays caused by static scheduling. Second, it uses *dead time elision*, which improves the total amount of usable memory bandwidth by reducing the dead time.

To illustrate how lattice priority scheduling addresses demand imprecision, Section 3.1 proposes a simpler scheduling algorithm based on the concept of *dynamic bandwidth allocation*. Then, Section 3.2 extends this idea with a fully dynamic schedule, further improving performance.

3.1 Dynamic Bandwidth Allocation

Lattice priority scheduling uses dynamic bandwidth allocation to alleviate demand imprecision. To illustrate dynamic bandwidth allocation, we introduce a scheduling algorithm called dynamic bandwidth scheduling (DBS). DBS begins with a static schedule as in TP. Then, at the start of L's turn, DBS checks if L has pending transactions. If it does not, the turn is given to H.

Figure 5 shows an example run of DBS. Initially, the TDM schedule is $H, L, \cdots$ The contents of the queues at the start of L's turn at time t_L are shown. Since L has no transactions

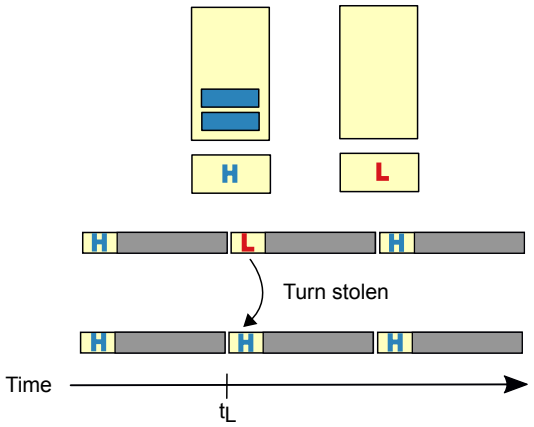


Figure 5: Dynamic bandwidth allocation example.

Algorithm 1 Priority Scheduling

```
1: procedure SELECTTURNOWNER
2:     turn_owner ← ⊥
3:     while ( turn_owner≠ ⊤ and (
           QueueEmpty(turn_owner) or
           not HasBandwidth(turn_owner) ) ) do
4:         turn_owner←AscendFrom(turn_owner)
5:     end while
6:     ConsumeBandwidth(turn_owner)
7:     return turn_owner
8: end procedure
```

in its queue at t_L, its turn is reallocated to H which does have pending requests.

The decision to reallocate the turn from L cannot depend on H. If neither have queued transactions at the start of L's turn, L's turn is still given up so that information about H is not leaked to L. The decision to reallocate the turn is final. If L's turn is given up, but it enqueues a transaction later in that turn, L cannot reclaim its turn. If the scheduler allowed turns to be reclaimed, it would leak whether or not the security class which received the bandwidth (H) actually issued a transaction.

Thus, DBS adds a small overhead not present in TP. If neither L nor H have pending transactions at the start of L's turn, it will be given to H. Then, if L enqueues a transaction later in the turn, it cannot be issued. In this case H does not make use of the turn. In TP, L would have kept its turn. However, experiments confirm that this case is exceptional and the overhead is small.

3.2 Dynamic Scheduling

The performance of DBS can be improved by scheduling security classes dynamically. The lattice priority scheduling algorithm applies the concept dynamic bandwidth allocation to a fully dynamic schedule, removing delays caused by static scheduling. It is strictly better than DBS. At a high level, lattice priority scheduling prioritizes L over H, and H is scheduled when L has no pending transactions or when L reaches a bandwidth limit.

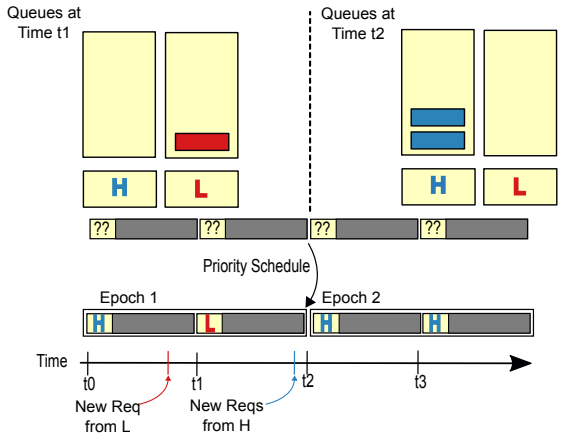


Figure 6: Lattice priority example.

Since memory transactions take multiple cycles, memory resources must still be granted at the granularity of a turn (including dead time), so that in-flight transactions from different classes do not interfere. Instead of using a static schedule, the arbiter selects which security class is active each turn using Algorithm 1.

The algorithm searches for the lowest class (`turn_owner`) with pending transactions by checking if the queue of each class is empty. It stops when it finds a security class with pending transactions. It would be insecure for the arbiter to check H and then only schedule L if H did not have pending requests. In this case, L could observe whether or not H had pending transactions. Instead, $L(\bot)$ is the first candidate turn owner. Then, if the candidate turn owner has an empty queue (`QueueEmpty(turn_owner)`), the algorithm calls `AscendFrom(turn_owner)` to select the next security class which can see information from `turn_owner`. In this simple case with only two security classes, `AscendFrom()` always returns H. Section 5 describes how `AscendFrom()` works in general. The algorithm stops if there are no other security classes permitted to see information from the candidate. This is true when the candidate is $H(\top)$.

If both L and H are memory-intensive, simply prioritizing L over H would be unfair, and would lead to performance loss for H. Further, L could execute a denial-of-service attack causing H to starve. Instead, priority scheduling places an upper bound on the number of turns that L can be granted within an *epoch*, which is a static, fixed-length interval expressed as a number of turns. As Algorithm 1 searches for a turn owner, it calls `HasBandwidth()` which returns true when the argument class has turns left in the epoch. When the turn owner is selected, it calls `ConsumeBandwidth()` to indicate that it has used up a turn. Counters which track the remaining bandwidth for each security class are reset at the start of each epoch. The epoch length and the maximum number of turns granted to each security class can be adjusted depending on static characterizations of the programs, as long as static characterization does not violate security.

Dynamic scheduling with an epoch of 2 turns is similar to turn stealing with the static schedule $L,H,\cdots$, but dynamic scheduling is strictly better. In both cases, H is granted at least one out of every two turns. When L has few requests, H is granted more turns with either approach. However, priority scheduling is more flexible. Figure 6 demonstrates the additional flexibility of lattice priority scheduling with an example. Assume the epoch length is 2 and L is granted at most 1 turn per epoch. The queues for H and L at time t_0 are empty. Since L has no pending requests, H is allocated the turn even though H has no pending requests. Otherwise, L could learn that H did not have a request.

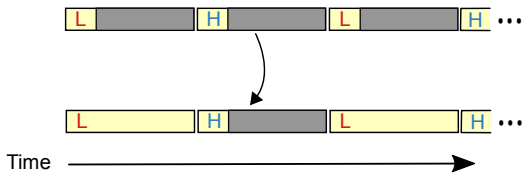


Figure 7: Dead-Time Elision.

Some time between t_0 and t_1, a new request is enqueued by L. At the start of time t_1, both H and L have pending requests. Since L has higher priority than H, and L has not consumed its maximum of one turn during the first epoch, lattice priority scheduling schedules L. If instead, DBS was used with the static schedule $L,H,\cdots$, the first turn would have been reallocated to H since L had no useful work. However, since the second turn is statically scheduled to H, the turn is not given to L even though H has no pending requests. This is wasteful since L does have pending requests and could make use of the turn.

Continuing with the example run with dynamic scheduling, at time t_2 the second epoch begins and there are two new requests enqueued by H, but none enqueued by L. Both turns are given to H since L has no pending requests. Since this decision depends only on the fact that L has no requests, L learns nothing about whether or not H had any requests. LPS responds to dynamic program behavior just as well as DBS, but can also remove additional delays.

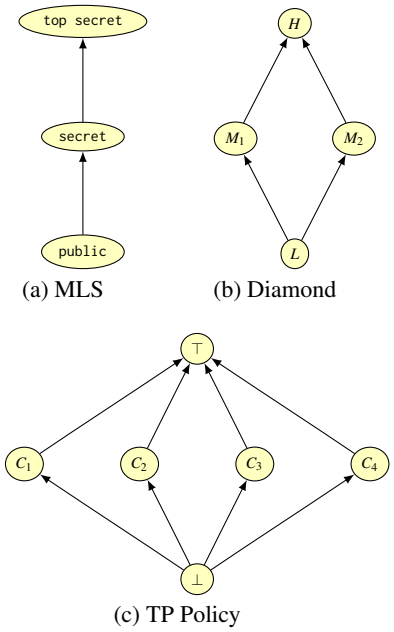
3.3 Dead Time Elision

Since H can learn about L, it is permissible for transactions from L to interfere with H. Transactions from L can remain in-flight at the start of H's turn, and the dead time between them can be elided (skipped) as shown in Figure 7. By eliding the dead time (shown in gray) during L's turn, L can continue issuing transactions until the turn ends. However, the dead time is still needed at the end of H's turn to prevent transactions from H from interfering with L.

For the system with classes L and H, dead times can be elided whenever L is currently scheduled. The owner of the next turn could be either L or H, but information is permitted to flow from L to either L or H. Section 5.2 generalizes dead-time elision to support arbitrary security policies.

4. LATTICE SECURITY MODEL

This work leverages the widely accepted lattice model [11] of security to precisely capture the security needs of a wide range of systems. Under the lattice model, entities in a system are assigned a security class. A set of security classes SC, together with an ordering relation $\sqsubseteq$, form a lattice $\langle SC, \sqsubseteq \rangle$. Information is allowed to flow from class



(a) MLS (b) Diamond

(c) TP Policy

Figure 8: Example lattice policies

$A \in SC$ to $B \in SC$ if and only if $A \sqsubseteq B$ holds. The relation, $\sqsubseteq$, must be reflexive, transitive, and antisymmetric.

Since lattices are a type of *partial* order, not all security classes are necessarily ordered — they may be incomparable. Security classes A and B are *incomparable* if neither $A \sqsubseteq B$ nor $B \sqsubseteq A$ hold. The *meet* of A and B, written $A \sqcap B$, is the greatest class less than both A and B. Similarly, the *join* of A and B, written $A \sqcup B$, is the least class greater than both A and B. For a partial order to be a lattice, taking the meet or join of any two classes in the lattice must result in a class which is also in the lattice. The classes $\top$ and $\bot$ denote the greatest and least of all classes in the sense that for all classes A, $A \sqsubseteq \top$ and $\bot \sqsubseteq A$ hold. The terms "A is lower than B" and "A is higher than B" are occasionally used as abbreviations for $A \sqsubseteq B$ and $B \sqsubseteq A$ throughout this paper.

The lattice model is highly expressive, and it can be used to describe the needs of many practical systems. For example, it can be used to describe the Bell-Lapadula multi-level security (MLS) model [21] in which information can flow from `public` to `secret` and from `secret` to `top secret`. Figure 8a shows the MLS model in a common pictorial representation of a lattice policy. The arrows show the direction in which information is allowed to flow. For example, an arrow points from `public` to `secret` since `public` $\sqsubseteq$ `secret`. Since the lattice is transitive, it is implied that information can flow from `public` to `top secret` as well. This policy is totally ordered since it contains no incomparable elements.

Incomparable security classes are useful for describing mutual distrust. For example, with the "diamond" lattice shown in Figure 8b, M_1 and M_2 are incomparable and information cannot flow in either direction between them. However, information from L can flow to M_1 or M_2 and information from both M_1 and M_2 can flow to H. These security classes might be used to describe a cloud system with several low-security VMs (L), two high-security VMs that require timing-channel protection from all other VMs in the system (M_1 and M_2), and a cloud owner class (H) that includes the hypervisor and scheduling/analysis programs that compute using input data from all the clients.

In this paper, we assume that a security class is assigned to each process (e.g., by the OS or hypervisor). Memory requests are queued and scheduled according to the security class of the process that issued them. The policy determines what scheduling restrictions are needed for protection. The security policy supported by TP can also be described in the lattice model as shown in Figure 8c for a system with 4 security classes. Each of the four security classes are incomparable, so the scheduling decisions made by the memory controller must be heavily restricted. The security classes $\top$ and $\bot$ are not actually used, but they are needed formally to represent the join and meet of the other classes. Given this lattice, the approaches described in this paper will behave equivalently to TP.

5. MEMORY PROTECTION UNDER THE LATTICE MODEL

Section 3 presents lattice priority scheduling for a system with two security classes, $L \sqsubseteq H$. Practical systems may contain any number of entities, some pairs of which may be mutually distrusting. This section generalizes LPS to support a wider range of systems by leveraging the lattice model

5.1 Generalized Dynamic Scheduling

When scheduling security classes, it is preferable to schedule classes that actually have pending transactions. In a memory controller that prevents all timing channels, the time that one class is scheduled cannot depend on the contents of another class's queue. Under a policy in the lattice model, it is permissible for the timing of a security class to depend on the demand from any lower class.

To make the best use of memory bandwidth, LPS searches through the security classes until it finds one that has a pending transaction. A sequence of classes, $C_1, C_2, \cdots, C_i$ that are searched for pending transactions is defined as a *lattice traversal*. An attacker at class C_i that gets scheduled can observe (through timing) that classes $C_1, \cdots, C_{i-1}$ must not have had transactions. Therefore, C_i must be higher than all those checked before it for this to be secure. That is, the sequence in which security classes are searched must form an ascending chain.

As an example, consider three classes, $L \sqsubseteq M \sqsubseteq H$. If the scheduler checks L and finds that it has no transactions, it would be safe to schedule either M or H. If the scheduler then checks M and finds it also has no transactions, it can safely check H. However, if H is checked first, it would be insecure to check M, since this creates a timing dependence of M on H. Therefore, the only secure traversal that checks all classes is L, M, H.

Since lattices are partial orders, there may be incomparable classes. That is, classes $A, B \in SC$ where neither $A \sqsubseteq B$ nor $B \sqsubseteq A$ is true. Whenever there are incomparable classes, there are multiple ascending chains, and there is no ascending chain which includes all security classes.

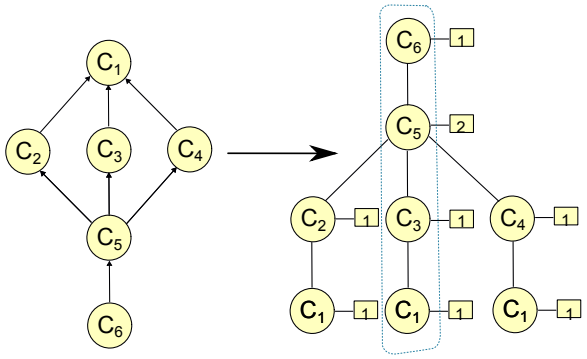


Figure 9: Tree structure for selecting traversals in the lattice priority scheduling algorithm.

For example, in Figure 8b, both L, M_1, H and L, M_2, H are ascending chains. For fairness it is necessary to ensure that both M_1 and M_2 are scheduled, so both traversals must be used. Though care must be taken — information from M_1 and M_2 cannot be used to decide which ascending chain to use (e.g., it is insecure to simply pick the one with pending transactions). However, both M_1 and M_2 can see information from L since they are both above L. More generally, a class may be chosen from among incomparable classes $C_1, C_2, \cdots C_n$ using information from classes at or below $C_1 \sqcap C_2 \sqcap \cdots \sqcap C_n$.

The lattice priority scheduler simply alternates between incomparable classes in a round-robin fashion. Consider the lattice in Figure 8b with two incomparable classes M_1 and M_2 above class L. Assume all classes have turns remaining in this epoch. The first time L has no pending requests, M_1 is granted the turn regardless of whether or not M_1 or M_2 actually have requests. The next time, M_2 is granted the turn, and the time after that M_1 is scheduled, and so on. The decision of which incomparable class to check depends only on how often L has an empty queue, which is acceptable for both M_1 and M_2 to learn.

Using the above intuition, lattice priority scheduling is generalized to support any lattice. `AscendFrom` in Algorithm 1 uses a tree structure as shown in Figure 9 to select an ascending chain. Each node of the tree is a security class. Each parent is directly less than its children (i.e. it is covered by its children). Each node has a counter that increments up to the number of children it has (i.e., the number of classes that it is directly less than in the lattice). Whenever a node is reached during a traversal, its counter is incremented and then used as an index to select from among its children.

In the example shown in Figure 9, class C_5 is less than three incomparable classes C_2, C_3, and C_4, so it has a counter that increments from 1 to 3. Assume all classes have bandwidth remaining (turns left) during this epoch. First, C_6 is checked. Since it has no requests, it is not scheduled and C_5 is checked. Since C_5 has no requests either, it is also not scheduled. In this iteration, the counter is 2 so the next class to be offered the turn is C_5's second child, C_3.

5.1.1 *Fairness for General Policies*

To prevent starvation and improve fairness, security classes are guaranteed a minimum number of turns within each epoch.

Algorithm 2 Start-of-Turn Turn Allocation

```
1:  procedure AllocateTurn
2:      if IsTurnStart() then
3:          active_class←SelectTurnOwner()
4:      end if
5:  end procedure
6:  procedure ElideDeadTime
7:      lower_bound ← ⊤
8:      for C ∈ {C′|C′ ⊑ active_class} do
9:          if HasBandwidth(C) then
10:             lower_bound ← lower_bound ⊓ C
11:         end if
12:     end for
13:     return active_class ⊑ lower_bound
14: end procedure
```

To enforce this minimum, each class C is given an additional *bandwidth counter* representing the number of turns C can become active this epoch. The counter decreases whenever C uses a turn that it is offered. It is initially the maximum number of turns C can be active in an epoch, which is

$$T_{epoch} - \sum_{C_i \in C+} T_{min}(C_i)$$

where T_{epoch} is the number of turns in an epoch, $C+$ is the set of classes that C is less than, or $\{C_i | C \sqsubset C_i\}$, and $T_{min}(C_i)$ is the minimum number of turns guaranteed to C_i in an epoch.

When selecting the next active class, Algorithm 1 calls `HasBandwidth(turn_owner)` to check if the candidate class, `turn_owner`, has used its maximum number of turns during this epoch. If the bandwidth counter of `turn_owner` is zero, `HasBandwidth(turn_owner)` returns false, preventing `turn_owner` from being scheduled. When `turn_owner` is scheduled, `ConsumeBandwidth(turn_owner)` decrements the bandwidth counter of `turn_owner`.

5.2 Generalized Dead Time Elision

In general, dead time can be elided whenever the scheduler is *certain* that the currently scheduled security class will be less than or equal to the next security class to become active (even though the next class may not have been decided yet). The time that the next active security class is decided affects whether or not dead time can be elided. There are two suitable choices: 1) at the start of the turn being allocated and 2) at the start of when the dead time would begin if it is not elided.

Algorithm 2 shows the first approach in which the next active security class is decided at the start of the turn. In Algorithm 2, `AllocateTurn` decides which security class is active at the start of each turn by calling `SelectTurnOwner` which is defined in Algorithm 1. Then, `ElideDeadTime` determines if dead time can be elided at the end of this turn. If all the security classes below the currently active one have already consumed all of their bandwidth for this epoch, none of them can become active next turn. In this case, the current active class will always be less than or equal to the next one, so dead time can be skipped. `ElideDeadTime` checks for this condition.

Algorithm 3 Dead-Time Turn Allocation

```
1: procedure AllocateTurn
2:     if IsTurnStart() then
3:         active_class←next_active
4:     end if
5: end procedure
6: procedure AllocateNext
7:     if IsStartOfDeadTime() then
8:         next_active← SelectTurnOwner()
9:     end if
10: end procedure
11: procedure ElideDeadTime
12:     return active_class ⊑ next_active
13: end procedure
```

Algorithm 3 shows the second approach, in which the next active class is decided just before dead time will begin if it is needed. It uses `AllocateNext` to pick the next active class just before the start of dead time. Then it uses `ElideDeadTime` to decide if dead time can be skipped. Now the class which is scheduled next is known at the time when `ElideDeadTime` is called, so it can simply compare the active class to the next one. At the start of the turn it makes the previously decided next active class (`next_active`) the new active class (`active_class`).

Dead time may be dropped more often when turns are allocated at the start of dead time. However, there is a tradeoff. When the turn is allocated sooner, the turn is "locked in" earlier. If in a system with classes $L \sqsubseteq H$, L had no requests at the start of dead time, H would be scheduled next regardless of whether or not H had requests. However, if requests from L arrive later, L cannot reclaim the turn.

6. HARDWARE IMPLEMENTATION

This section describes how the lattice priority scheduling algorithm is implemented in hardware. As with TP, each physical thread (i.e. thread in SMT) has a register located in the core which stores a security ID representing the security class of the software running in that thread. The memory controller has a number of pending transaction queues equal to the number of threads. Each queue has a register which stores a security ID indicating its owner. The security ID registers in the cores and in the memory controller are managed by the trusted hypervisor or OS. Access controls should prevent modifications by untrusted software.

Unlike TP, lattice priority scheduling supports an arbitrary policy specified in the lattice model. The lattice policy is stored in a dedicated table in the memory controller. The table stores a 1-bit entry for each pair of security classes (A, B) indicating whether or not $A \sqsubseteq B$ is true. The table is initialized and managed by trusted software, and access controls should prevent modifications by untrusted software.

LPS requires registers to configure the number of turns in the epoch and a counter to track the current turn in the epoch. Registers set the maximum bandwidth per epoch for each security class. A set of counters track the bandwidth consumed in the current epoch by each security class.

Unlike TP, which decides the active security class statically, LPS decides the active class for each turn dynamically. Doing so requires checking the queues and bandwidth counters for each security class. This information can be checked for each security class in parallel. A priority encoder is used to select the next turn owner based on the security policy.

While in general a system might have many security classes, the hardware data structures just described need only support as many security classes as there are physical threads. The security classes can be virtualized to support arbitrarily many classes. The maximum number of simultaneously running security classes is the number of physical threads. Since the number physical threads is small, the area overhead of lattice priority scheduling is small as well. For example, to support a 64 thread system, the policy table would require 512B of storage.

7. EVALUATION

7.1 Methodology

The performance of lattice priority scheduling is evaluated in a multicore out-of-order processor using a simulator based on Gem5 [8] integrated with DRAMSim2 [28]. Simulation parameters are given in Table 1. The experiments simulate 4 cores each with private 32KB L1I/D caches and a private 256kB L2. In our experiments, each core runs at most one security class concurrently. In practice, there may be as many simultaneously executing security classes as there are physical threads. Each core has a private 1MB last-level cache (LLC). Contemporary server processors have a shared LLC with 1MB per thread [16]. These experiments use private caches so that only the direct performance improvement of the memory controller is measured, and changes in cache interference patterns are not measured.

Our experiments use multiprogram workloads, and we describe our methodology precisely enough that it can be repeated [18]. The simulations are fast-forwarded until each benchmark has executed at least 1 billion instructions. Benchmarks may reach this threshold at different times, meaning the benchmarks which run faster will be fast-forwarded for more instructions. However, detailed simulations begin from the same point for each workload and for all system configurations. After fast-forwarding, results are collected with a detailed simulation until each core has executed for at least 100M instructions. Statistics are collected for each benchmark at the 100M instruction mark, but all benchmarks continue to run until the simulation ends, so that there is interference for the entire simulation of the insecure baseline.

The performance evaluation metric is system throughput (STP) which is the aggregate normalized IPC of programs in the multiprogram workload. It is computed as

$$\sum_{i=1}^{n} \frac{IPC_{MP,i}}{IPC_{SP,i}}, \tag{1}$$

where $IPC_{MP,i}$ is the IPC of the i^{th} program in the workload when run in parallel with the others, and $IPC_{SP,i}$ is the IPC for the same program when run alone in the same system.

The experiments use multiprogram workloads comprising SPEC benchmarks. Workloads are selected to capture different mixes of memory intensity. Some workloads use the

Processor			
Cores and Frequency			4/8cores, 2GHz
Gem5 core model			"O3"
ISA			ARMv8-A
Cache Hierarchy			
L1d / L1i	32kB	2-way	2 cycles
L2 private	256kB	8-way	7 cycles
L3 private	1MB	16-way	10 cycles
Network Clock			1GHz
Memory			
Size and Frequency		8GB	667MHz
Channels, ranks, and banks			1, 8, 8

Table 1: Simulator configuration parameters.

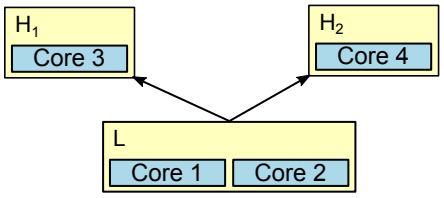


Figure 10: Security policy for performance evaluation.

Workload name	Benchmarks
mix_1	astar x2, libquantum x2
mix_2	astar x3, libquantum
mix_3	h264ref, hmmer, sjeng, libquantum
mix_4	astar x2, mcf x2
mix_5	mcf x2, libquantum x2
mix_6	libquantum x2, hmmer, gobmk
mix_7	libquantum x1, astar x3
mix_8	libquantum x2, mcf x2

Table 2: Multiprogram workloads.

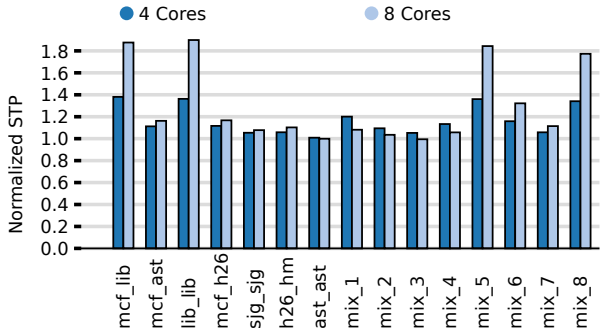


Figure 11: Normalized STP as core count increases.

naming convention bench1_bench2 to indicate that bench1 is executed on odd-numbered cores and bench2 is executed on even-numbered cores. Table 2 summarizes the remaining workloads by listing benchmarks in order of core number. Note that order matters since LPS schedules security classes differently, and cores may be in different security classes.

The security policy affects the performance of lattice priority scheduling. Unless otherwise stated, these experiments use the policy shown in Figure 10, which captures the security requirements of a cloud computing environment with both high and low-confidentiality VMs. Standard VMs run in the security class L, and VMs which require timing-channel protection run in security classes H_1 and H_2. The classes H_1 and H_2 are incomparable and above L. This policy guarantees that information cannot leak out of H_1 or H_2 to any other VM, but relaxes protection for L. Cores 1 and 2 run applications with security class L. Cores 3 and 4 run in security classes H_1 and H_2 respectively. The top security class above H_1 and H_2 (not shown) is not used, and is not allocated any bandwidth.

The turn lengths are chosen based on prior findings that turn lengths that are close to the minimum (the dead time) achieve better performance [34, 29]. The dead time is 43 memory cycles. For lattice priority scheduling, the turns are 44 cycles. In all experiments, TP uses three incomparable security classes where Core 1 and 2 share a class. For TP, the same turn length (44) is used for security classes containing one program. Since the first security class runs two programs, the turn length is doubled to accommodate the bandwidth demands of both programs. The lattice priority scheduler uses an epoch of 4 turns with a minimum of 1 turn reserved for each of H_1 and H_2 per epoch. This closely follows the configuration used for TP; assuming L has high memory demand it will consume 2 turns per epoch, and if these turns are adjacent, the dead time between them is elided, mirroring the behavior of a turn that is twice the length of the minimum.

7.2 Performance and Scalability

Figure 11 studies the performance improvement of lattice priority scheduling compared to temporal partitioning as the core count increases from 4 to 8 cores. The performance metric is system throughput normalized to temporal partitioning. The 4 core machine uses the cloud computing policy described earlier, and the 8 core machine uses its natural extension. In the policy for 8 cores, Cores 1–4 share the lowest security class and Cores 5–8 are all higher than cores 1–4 and incomparable with each other. The priority scheduler uses dead time elision and decides the active class at the start of the turn. Experiments show that selecting the next active class at the start of the turn achieves better performance than selecting the next active class at the start of the dead time. In the best case (lib_lib), priority scheduling improves STP by 89% and 38% for 8 and 4 cores respectively. The greatest performance improvement is observed for this workload because libquantum is very memory intensive. On average, priority scheduling improves the STP compared to TP by 30% and 17% for the 8 and 4 core systems respectively.

7.3 Per-Core Performance

Figure 12 shows the speedup of each core individually. Each bar represents the IPC of an individual core when using LPS, normalized to the IPC of that same core when using TP. Notably, the lattice-aware memory controller improves performance for cores 1 and 2 which run low-confidentiality applications as well as cores 2 and 3, which run higher-confidentiality applications. Dead time elision allows lower-confidentiality applications to continue issuing memory requests after the dead time. Priority scheduling allows the higher-confidentiality applications to use more bandwidth when the lower-confidentiality applications have few requests.

Lattice priority scheduling provides large performance improvements for some workloads in the system (77% for core 4 in mix_4), but only infrequently worsens the performance

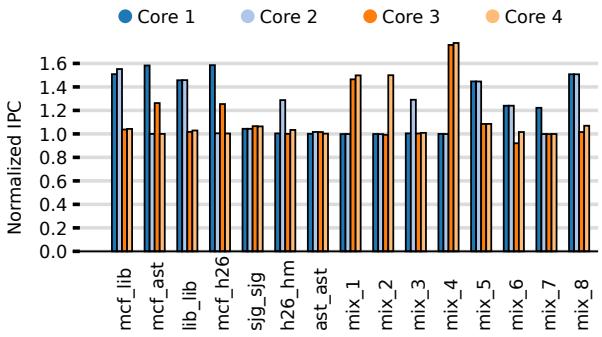


Figure 12: Individual core speedup.

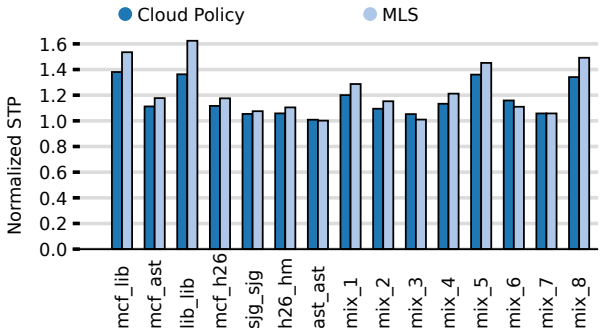


Figure 13: STP with two different policies normalized to the STP of the insecure baseline.

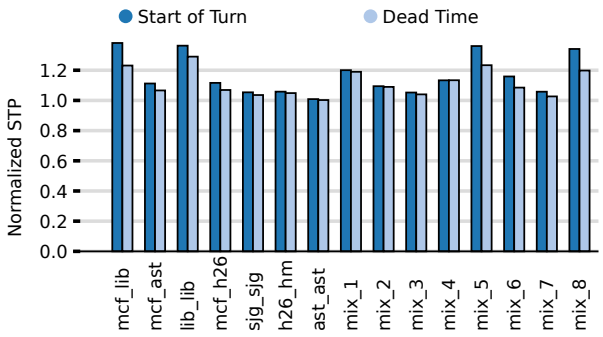


Figure 14: Performance impact of elision and turn allocation time.

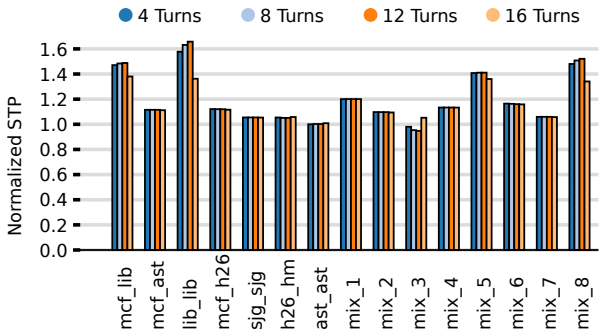


Figure 15: STP normalized to TP as epoch length changes.

of other workloads. The only workload that non-negligibly reduces the IPC of one core compared to TP is `mix_6` which reduces the IPC of core 3 by 8%. However, priority scheduling still improves STP for this workload since the IPC of cores 1 and 2, are increased by over 24% each.

7.4 Lattice Policies and Performance

LPS provides more flexibility than strict static scheduling by preventing only timing channels that are specified in the policy. Therefore, performance depends on the security policy. The performance was evaluated for a 4-core system with strict TP and lattice priority scheduling using two different policies. The first policy is the cloud policy used in all other experiments. The second policy is MLS as shown in Figure 8a where cores 1 and 2 share the public security class. As before, TP is configured so that cores 1 and 2 share a security class.

Figure 13 shows the STP of lattice-aware scheduling normalized to TP. The cloud policy is more restrictive than the MLS policy since the MLS policy allows information to leak from core 3 to core 4. Therefore, with the MLS policy the improvement is higher. The MLS policy has an average improvement of 23% compared to TP and the cloud policy has an average improvement of 17%.

7.5 Scheduling Decision Time

Figure 14 shows how the time when the scheduling decision is made (either at the start of the turn or at the start of the dead time) affects performance. Allocating at the dead time allows dead times to be dropped more often, but increases the chance that a lower security class will give up its turn preemptively, and have a request later in the turn get delayed. The bars represent the STP of priority scheduling when turns are allocated at the start of the turn and at the start of the dead time, each normalized to TP. For all evaluated applications, deciding which security class is scheduled next at the start of the turn is better than deciding at the start of the dead time before that turn. This is because often, the overhead of locking-in the turn allocation decision earlier is greater than the improvement gained by eliding turns more often.

7.6 Epoch Length

Figure 15 shows the STP of LPS as the epoch length is changed. The STP is normalized to that of the insecure baseline. The epoch length is increased from 4 to 16. Cores 3 and 4 are each given a minimum of 1 turn per epoch in all cases. This experiment shows the tradeoff between providing more fairness and providing more flexibility. For some workloads, longer epochs, and therefore more flexibility, leads to better performance. Lattice priority scheduling achieves the best average STP across all workloads with an epoch length of 12. With this epoch length, the average system throughput increases by 20% and by up to 63% for `lib_lib` compared to TP. However, this increase in throughput comes at the expense of fairness. The IPC of cores 3 and 4 for `lib_lib` are reduced by 20% compared to TP. With an epoch of length 4, the scheduler is more fair, as can be seen in Figure 12. However, the average STP improvement is slightly lower (17%).

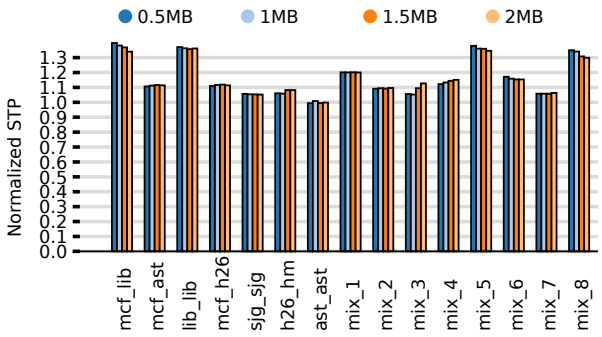


Figure 16: STP of lattice scheduling normalized to TP as cache per thread changes.

7.7 Impact of Last-Level Cache Size

The performance of both TP and lattice priority scheduling depends on the size of the LLC, since fewer cache misses means that the memory latency is incurred less often. In all experiments, private LLCs are used so that only the direct impact of our improvements to the memory controller are measured, and performance changes caused by differences in cache interference patterns are removed. Figure 16 shows the STP of lattice priority scheduling normalized to TP as the size of the last level cache allocated to each thread changes. Both TP and lattice priority scheduling have the same cache size. For workloads which are not very cache-sensitive, such as `lib_lib` or `ast_ast`, the size of the cache has almost no impact. In other cases, since there are fewer misses for both systems, and since the miss penalty is higher for TP, the improvement from increasing the cache size is greater for TP than for priority scheduling. The improvement of priority scheduling compared to TP is high even for larger caches.

8. RELATED WORK

Many microarchitectural timing-channel attacks have been demonstrated in: caches [23, 26, 7, 4, 27, 20, 1, 44, 38], branch predictors [2, 3], on-chip networks [35], processor pipelines [36], memory buses [13, 37], and main memory [34, 12]. This paper addresses microarchitectural timing channels in main memory.

Temporal partitioning [34] (TP) has been proposed to address timing channels in main memory. The memory controller discussed in this paper builds on TP. Shaifee et al. [29] propose Fixed Service policies (FS) for memory controller protection. FS is equivalent to TP configured with the minimum turn length and with the addition of performance optimizations. Like TP, FS assumes a mutually distrusting security model, and cannot precisely enforce lattice model policies. The optimizations proposed by Shaifee et al. [29] can be applied to LPS.

Fletcher et al. [12] address a timing channel in a memory controller where the attacker can directly measure the frequency of memory requests (e.g., by replacing the DRAM with a malicious device). These attacks are present without shared hardware, and the solution by Fletcher et al. does not address timing channels due to shared hardware.

Several techniques have been proposed for verifying that timing protection is enforced. GLIFT [33, 24, 32, 25, 31] addresses hardware timing channels at the gate level. Hardware description language tools with information flow type systems [22, 42] address timing channels, since they are indistinguishable from other types of information flow at the register transfer level.

Programming-language techniques can also prevent timing-channel attacks. Zhang et al. [40, 41] address timing channels, including some hardware timing channels, with an information flow control type system. However, this approach cannot address the timing channels through shared memory that are prevented by LPS.

The lattice model proposed by Denning [11] is widely used in information flow control type systems which are surveyed in [5]. Since both LPS and information flow control type systems both enforce policies expressed in the lattice model, LPS could be provided with information from such a type system at run-time to precisely enforce memory timing-channel protection.

Efforts have been made to detect covert timing channels. Hunger et al. [15] propose a formal model of timing channels and show that whenever an attacker reads a covert channel, it causes interference that can be used for detection. Notably, they also find that a covert channel caused by memory accesses has a higher channel capacity than covert channels caused by caches. Chen et al. [10] propose CC-Hunter, which uses hardware support to detect timing channels. The techniques proposed in this paper could be combined with a detector by enabling the lattice memory scheduler only after an attack has been detected.

Hu [14] proposed lattice scheduling, which uses the lattice model to efficiently address timing channels in process schedulers. Lattice scheduling schedules processes so they increase monotonically in security class order to avoid flushing caches when context switching from one process to another with a higher class. Lattice scheduling was an inspiration for dead time elision. Wang et al. [35] schedules on-chip network transactions based on a total order to provide network timing-channel protection. Lattice priority scheduling similarly schedules memory transactions using a partially-ordered security policy. However, this work is the first to enforce a lattice policy in a memory controller. Memory transactions have variable-cycle transactions that add complexity not present in on-chip networks. The approach presented in this work are also supports lattice policies, which are more general than total orders.

9. CONCLUSION

This work presents a way to improve the performance of a timing-channel protected memory controller. We leverage the lattice model to precisely capture the security needs of the target system and enforce protection only as needed among security classes. A key aspect is the ability of these approaches to remove timing channels in only one direction between a pair of security classes. By enforcing information flow in just one direction, the memory controller gains the ability to respond to run-time program behavior, significantly increasing performance.

10. REFERENCES

[1] O. Aciiçmez, B. B. Brumley, and P. Grabher. New Results on Instruction Cache Attacks. In *Proceedings of the International Conference on Cryptographic Hardware and Embedded Systems*, 2010.

[2] O. Aciiçmez, c. K. Koç, and J.-P. Seifert. On the Power of Simple Branch Prediction Analysis. In *Proceedings of the Symposium on Information, computer and Communications Security*, 2007.

[3] O. Aciiçmez, c. K. Koç, and J.-P. Seifert. Predicting Secret Keys via Branch Prediction. In *Proceedings of the Cryptographers' track at the RSA conference on Topics in Cryptology*, 2007.

[4] O. Aciiçmez, W. Schindler, and c. K. Koç. Cache Based Remote Timing Attack on the AES. . In *Proceedings of the Cryptographers' track at the RSA conference on Topics in Cryptology*, 2007.

[5] A. M. Andrei Sablefield. Language-Based Information-Flow Security . *Journal on Selected Areas in Communications*, 2003.

[6] A. Baumann, M. Peinado, and G. Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *Proceedings of the Conference on Operating Systems Design and Implementation*, 2014.

[7] D. J. Bernstein. Cache-Timing Attacks on AES. Tech. Report, 2005.

[8] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The Gem5 Simulator. *SIGARCH Comput. Archit. News*, 2011.

[9] R. Boivie. SecureBlue++: CPU Support for Secure Execution. Technical report, 2012.

[10] J. Chen and G. Venkataramani. CC-Hunter: Uncovering Covert Timing Channels on Shared Processor Hardware. In *Proceedings of the International Symposium on Microarchitecture*, 2014.

[11] D. E. Denning. A Lattice Model of Secure Information Flow. *Commun. ACM*, 1976.

[12] C. W. Fletcher, L. Ren, X. Yu, M. van Dijk, O. Khan, and S. Devadas. Suppressing the Oblivious RAM timing channel while making information leakage and program efficiency trade-offs. In *International Symposium on High Performance Computer Architecture*, 2014.

[13] R. E. Fryer. The Memory Bus Monitor: A New Device for Developing Real-time Systems. In *Proceedings of the National Computer Conference and Exposition*, 1973.

[14] W.-M. Hu. Lattice Scheduling and Covert Channels. In *Proceedings of the Symposium on Security and Privacy*, 1992.

[15] C. Hunger, M. Kazdagli, A. S. Rawat, A. G. Dimakis, S. Vishwanath, and M. Tiwari. Understanding Contention-Based Channels and Using Them for Defense. In *Proceedings of the International Symposium on High Performance Computer Architecture*, 2015.

[16] Intel Corporation. http://ark.intel.com/compare/84679,84678,84677,84676.

[17] Intel Corporation. Intel Software Guard Extensions Programming Reference, 2014.

[18] A. N. Jacobvitz, A. D. Hilton, and D. J. Sorin. Multi-Program Benchmark definition. In *International Symposium on Performance Analysis of Systems and Software*, 2015.

[19] S. Jin, J. Ahn, S. Cha, and J. Huh. Architectural Support for Secure Virtualization Under a Vulnerable Hypervisor. In *Proceedings of the International Symposium on Microarchitecture*, 2011.

[20] J. Kong, O. Aciicmez, J.-P. Seifert, and H. Zhou. Deconstructing New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *Proceedings of the Workshop on Computer Security Architectures*, 2008.

[21] L. J. LaPadula and D. E. Bell. Secure Computer Systems: A Mathematical Model. 1996.

[22] 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 the Conference on Programming Language Design and Implementation*, 2011.

[23] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *Proceedings of the Symposium on Security and Privacy*, 2015.

[24] J. Oberg, W. Hu, A. Irturk, M. Tiwari, T. Sherwood, and R. Kastner. Theoretical Analysis of Gate Level Information Flow Tracking. In *Proceedings of the Design Automation Conference*, 2010.

[25] J. Oberg, W. Hu, A. Irturk, M. Tiwari, T. Sherwood, and R. Kastner. Information Flow Isolation in I2C and USB. In *Proceedings of the Design Automation Conference*, 2011.

[26] C. Percival. Cache Missing for Fun and Profit. In *BSDCan*, 2005.

[27] 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 Conference on Computer and Communications Security*, 2009.

[28] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE Comput. Archit. Lett.*, 2011.

[29] 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 the International Symposium on Microarchitecture*, 2015.

[30] J. Szefer and R. B. Lee. Architectural Support for Hypervisor-secure Virtualization. 2012.

[31] 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*, 2009.

[32] M. Tiwari, J. K. Oberg, X. Li, J. Valamehr, T. Levin, B. Hardekopf, R. Kastner, F. T. Chong, and T. Sherwood. Crafting a Usable Microkernel, Processor, and I/O System with Strict and Provable Information Flow Security. In *Proceedings of the International Symposium on Computer Architecture*, 2011.

[33] 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 the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.

[34] Y. Wang, A. Ferraiuolo, and E. Suh. Timing Channel Protection for a Shared Memory Controller. In *Proceedings of the 20th International Symposium on High Performance Computer Architecture*, 2014.

[35] Y. Wang and E. Suh. Efficient Timing Channel Protection for On-Chip Networks. In *Proceedings of the International Symposium on Networks-on-Chip.*, 2012.

[36] Z. Wang and R. B. Lee. Covert and Side Channels Due to Processor Architecture. In *Proceedings of the Annual Computer Security Applications Conference*, 2006.

[37] Z. Wu, Z. Xu, and H. Wang. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud. Proceedings of the USENIX Security Symposium, 2012.

[38] Y. Yarom and K. Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack. In *Proceedings of the USENIX Security Symposium*, 2014.

[39] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making Information Flow Explicit in HiStar. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, 2006.

[40] D. Zhang, A. Askarov, and A. C. Myers. Predictive Mitigation of Timing Channels in Interactive Systems. In *Proceedings of the Conference on Computer and Communications Security*, 2011.

[41] D. Zhang, A. Askarov, and A. C. Myers. Language-based Control and Mitigation of Timing Channels. In *Proceedings of the Conference on Computer and Communications Security*, 2012.

[42] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers. "A Hardware Design Language for Timing-Sensitive Information-Flow Security". In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[43] T. Zhang and R. B. Lee. CloudMonatt: An Architecture for Security Health Monitoring and Attestation of Virtual Machines in Cloud Computing. In *Proceedings of the International Symposium on Computer Architecture*, 2015.

[44] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In *Proceedings of the Conference on Computer and Communications Security*, 2012.