Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems

Thomas Moscibroda Onur Mutlu
Microsoft Research
{moscitho,onur}@microsoft.com

Abstract

We are entering the multi-core era in computer science. All major high-performance processor manufacturers have integrated at least two cores (processors) on the same chip — and it is predicted that chips with many more cores will become widespread in the near future. As cores on the same chip share the DRAM memory system, multiple programs executing on different cores can interfere with each others' memory access requests, thereby adversely affecting one another's performance.

In this paper, we demonstrate that current multi-core processors are vulnerable to a new class of Denial of Service (DoS) attacks because the memory system is "unfairly" shared among multiple cores. An application can maliciously destroy the memory-related performance of another application running on the same chip. We call such an application a *memory performance hog (MPH)*. With the widespread deployment of multi-core systems in commodity desktop and laptop computers, we expect MPHs to become a prevalent security issue that could affect almost all computer users.

We show that an MPH can reduce the performance of another application by 2.9 times in an existing dual-core system, without being significantly slowed down itself; and this problem will become more severe as more cores are integrated on the same chip. Our analysis identifies the root causes of unfairness in the design of the memory system that make multi-core processors vulnerable to MPHs. As a solution to mitigate the performance impact of MPHs, we propose a new memory system architecture that provides fairness to different applications running on the same chip. Our evaluations show that this memory system architecture is able to effectively contain the negative performance impact of MPHs in not only dual-core but also 4-core and 8-core systems.

1 Introduction

For many decades, the performance of processors has increased by hardware enhancements (increases in clock frequency and smarter structures) that improved single-thread (sequential) performance. In recent years, however, the immense complexity of processors as well as limits on power-consumption has made it increasingly difficult to further enhance single-thread performance [17]. For this reason, there has been a paradigm shift away from implementing such additional enhancements. Instead, processor manufacturers have moved on to integrating multiple processors on the same chip in a tiled fashion to increase system performance power-efficiently. In a *multi-core chip*, different applications can be executed on different processing cores concurrently, thereby improving overall system throughput (with the hope that the execution of an application on one core does not interfere with an application on another core). Current high-performance general-purpose computers have at least two processors on the same chip (e.g. Intel Pentium D and Core Duo (2 processors), Intel Core-2 Quad (4), Intel Montecito (2), AMD Opteron (2), Sun Niagara (8), IBM Power 4/5 (2)). And, the industry trend is toward integrating many more cores on the same chip. In fact, Intel has announced experimental designs with up to 80 cores on chip [15].

The arrival of multi-core architectures creates significant challenges in the fields of computer architecture, software engineering for parallelizing applications, and operating systems. In this paper, we show that there are important challenges beyond these areas. In particular, we expose a new security problem that arises due to the design of multi-core architectures – a Denial-of-Service (DoS) attack that was not possible in a traditional single-threaded processor.[1] We identify the "security holes" in the hardware design of multi-core systems that make such attacks possible and propose a solution that mitigates the problem.

[1]While this problem could also exist in SMP (symmetric shared-memory multiprocessor) and SMT (simultaneous multithreading) systems, it will become much more prevalent in multi-core architectures which will be widespreadly deployed in commodity desktop and laptop computers.

In a multi-core chip, the DRAM memory system is shared among the threads concurrently executing on different processing cores. The way current DRAM memory systems work, it is possible that a thread with a particular memory access pattern can occupy shared resources in the memory system, preventing other threads from using those resources efficiently. In effect, the memory requests of some threads can be denied service by the memory system for long periods of time. Thus, an aggressive memory-intensive application can severely degrade the performance of other threads with which it is co-scheduled (often without even being significantly slowed down itself). We call such an aggressive application a *Memory Performance Hog (MPH)*. For example, we found that on an existing dual-core Intel Pentium D system one aggressive application can slow down another co-scheduled application by 2.9X while it suffers a slowdown of only 18% itself. In a simulated 16-core system, the effect is significantly worse: the same application can slow down other co-scheduled applications by 14.6X while it slows down by only 4.4X. This shows that, although already severe today, the problem caused by MPHs will become much more severe as processor manufacturers integrate more cores on the same chip in the future.

There are three discomforting aspects of this novel security threat:

- First, an MPH can maliciously destroy the memory-related performance of other programs that run on different processors on the same chip. Such Denial of Service in a multi-core memory system can ultimately cause significant discomfort and productivity loss to the end user, and it can have unforeseen consequences. For instance, an MPH (perhaps written by a competitor organization) could be used to fool computer users into believing that some other applications are inherently slow, even without causing easily observable effects on system performance measures such as CPU usage. With the widespread deployment of multi-core systems in commodity desktop and laptop computers, we expect MPHs to become a much more prevalent security issue that could affect almost all computer users.
- Second, the problem of memory performance attacks is radically different from other, known attacks on shared resources in systems, because it cannot be prevented in software. The operating system or the compiler (or any other application) has no direct control over the way memory requests are scheduled in the DRAM memory system. For this reason, even carefully designed and otherwise highly secured systems are vulnerable to memory performance attacks, unless a solution is implemented in *memory system hardware* itself. For example, numerous sophisticated software-based solutions are known to prevent DoS and other attacks involving mobile or untrusted code (e.g. [9, 24, 26, 5, 7]), but these are unsuited to prevent our memory performance attacks.
- Third, while an MPH can be designed intentionally, a regular application can unintentionally behave like an MPH and damage the memory-related performance of co-scheduled applications, too. This is discomforting because an existing application that runs without significantly affecting the performance of other applications in a single-threaded system may deny memory system service to co-scheduled applications in a multi-core system. Consequently, critical applications can experience severe performance degradations if they are co-scheduled with a non-critical but memory-intensive application.

The fundamental reason why an MPH can deny memory system service to other applications lies in the "unfairness" in the design of the multi-core memory system. State-of-the-art DRAM memory systems service memory requests on a First-Ready First-Come-First-Serve (FR-FCFS) basis to maximize memory bandwidth [29, 28, 22]. This scheduling approach is suitable when a single thread is accessing the memory system because it maximizes the utilization of memory bandwidth and is therefore likely to ensure fast progress in the single-threaded processing core. However, when multiple threads are accessing the memory system, servicing the requests in an order that ignores which thread generated the request can unfairly delay some thread's memory requests while giving unfair preference to others. As a consequence, the progress of an application running on one core can be significantly hindered by an application executed on another.

In this paper, we identify the causes of unfairness in the DRAM memory system that can result in DoS attacks by MPHs. We show how MPHs can be implemented and quantify the performance loss of applications due to unfairness in the memory system. Finally, we propose a new memory system design that is based on a novel definition of *DRAM fairness*. This design provides memory access fairness across different threads in multi-core systems and thereby mitigates the impact caused by a memory performance hog.

The major contributions we make in this paper are:

- We expose a new Denial-of-Service attack that can significantly degrade application performance on multi-core systems and we introduce the concept of Memory Performance Hogs (MPHs). An MPH is an application that can destroy the memory-related performance of another application running on a different processing core on the same chip.
- We demonstrate that MPHs are a real problem by evaluating the performance impact of DoS attacks on both real and simulated multi-core systems.
- We identify the major causes in the design of the DRAM memory system that result in DoS attacks: hardware algorithms that are unfair across different threads accessing the memory system.
- We describe and evaluate a new memory system design that provides fairness across different threads and mitigates the large negative performance impact of MPHs.

2 Background

We begin by providing a brief background on multi-core architectures and modern DRAM memory systems. Throughout the section, we abstract away many details in order to give just enough information necessary to understand how the design of existing memory systems could lend itself to denial-of-service attacks by explicitly-malicious programs or real applications. Interested readers can find more details in [29, 8, 37].

2.1 Multi-Core Architectures

Figure 1 shows the high-level architecture of a processing system with one core (single-core), two cores (dual-core) and N cores (N-core). In our terminology, a "core" includes the instruction processing pipelines (integer and floating-point), instruction execution units, and the L1 instruction and data caches. Many general-purpose computers manufactured today look like the dual-core system in that they have two separate but identical cores. In some systems (AMD Athlon/Turion/Opteron, Intel Pentium-D), each core has its own private L2 cache, while in others (Intel Core Duo, IBM Power 4/5) the L2 cache is shared between different cores. The choice of a shared vs. non-shared L2 cache affects the performance of the system [13] and a shared cache can be a possible source of vulnerability to DoS attacks. However, this is not the focus of our paper because DoS attacks at the L2 cache level can be easily prevented by providing a private L2 cache to each core (as already employed by some current systems) or by providing "quotas" for each core in a shared L2 cache [27].

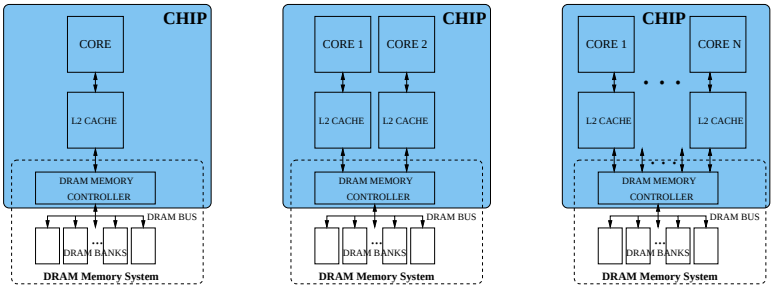

Figure 1: High-level architecture of an example single-core system (left), a dual-core system (middle), and an N-core system (right). The chip is shaded. The DRAM memory system, part of which is usually off chip, is encircled.

Regardless of whether or not the L2 cache is shared, the DRAM Memory System of current multi-core systems is shared among all cores. In contrast to the L2 cache, assigning a private DRAM memory system to each core would significantly change the programming model of shared-memory multiprocessing, which is commonly used in commercial applications. Furthermore, in a multi-core system, partitioning the DRAM memory system across cores (while still maintaining a shared-memory programming model) is also undesirable because:

1. DRAM memory is still a very expensive resource in modern systems. Partitioning it requires more DRAM chips along with a separate memory controller for each core, which significantly increases the cost of a commodity general-purpose system, especially in future systems that will incorporate tens of cores on chip.

2. In a partitioned DRAM system, a processor accessing a memory location needs to issue a request to the DRAM partition that contains the data for that location. This incurs additional latency and a communication network to access another processor's DRAM if the accessed address happens to reside in that partition.

For these reasons, we assume in this paper that each core has a private L2 cache but all cores share the DRAM memory system. We now describe the design of the DRAM memory system in current multi-core processors.

2.2 DRAM Memory Systems

A DRAM memory system consists of three major components: (1) the DRAM banks that store the actual data, (2) the DRAM controller (scheduler) that schedules commands to read/write data from/to the DRAM banks, and (3) DRAM address/data/command buses that connect the DRAM banks and the DRAM controller.

2.2.1 DRAM Banks

A DRAM memory system is organized into multiple banks such that memory requests to different banks can be serviced in parallel. As shown in Figure 2, each DRAM bank has a two-dimensional structure, consisting of multiple rows and columns. Consecutive addresses in memory are located in consecutive columns in the same row.[2] The size of a row varies, but it is usually between 1-32Kbytes in commodity DRAMs. In other words, in a system with 32-byte L2 cache blocks, a row contains 32-1024 L2 cache blocks.

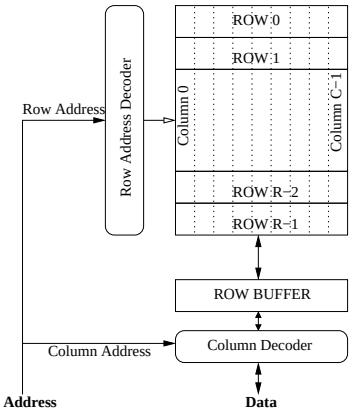

Figure 2: Organization of a DRAM bank

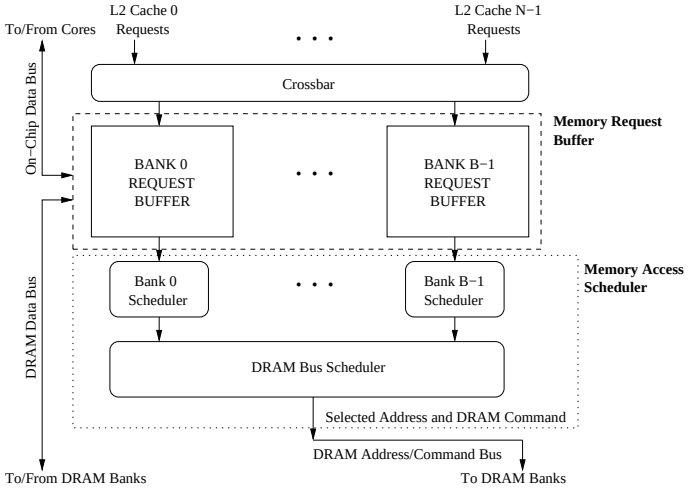

Figure 3: Organization of the DRAM controller

Each bank has one *row-buffer* and data can only be read from this buffer. The row-buffer contains at most a single row at any given time. Due to the existence of the row-buffer, modern DRAMs are not truly random access (equal access time to all locations in the memory array). Instead, depending on the access pattern to a bank, a DRAM access can fall into one of the three following categories:

1. **Row hit**: The access is to the row that is already in the row-buffer. The requested column can simply be read from or written into the row-buffer (called a *column access*). This case results in the lowest latency (typically 40-50ns in commodity DRAM, including data transfer time, which translates into 120-150 processor cycles for a core running at 3GHz clock frequency). Note that sequential/streaming memory access patterns (e.g. accesses to cache blocks A, A+1, A+2, ...) result in row hits since the accessed cache blocks are in consecutive columns in a row. Such requests can therefore be handled relatively quickly.
2. **Row conflict:** The access is to a row different from the one that is currently in the row-buffer. In this case, the row in the row-buffer first needs to be written back into the memory array (called a *row-close*) because the row access had destroyed the row's data in the memory array. Then, a *row access* is performed to load the requested row into the row-buffer. Finally, a column access is performed. Note that this case has much higher latency than a row hit (typically 80-100ns or 240-300 processor cycles at 3GHz).

[2]Note that consecutive memory rows are located in different banks.

3. **Row closed:** There is no row in the row-buffer. Due to various reasons (e.g. to save energy), DRAM memory controllers sometimes close an open row in the row-buffer, leaving the row-buffer empty. In this case, the required row needs to be first loaded into the row-buffer (called a *row access*). Then, a column access is performed. We mention this third case for the sake of completeness because in the paper, we focus primarily on row hits and row conflicts, which have the largest impact on our results.

Due to the nature of DRAM bank organization, sequential accesses to the same row in the bank have low latency and can be serviced at a faster rate. However, sequential accesses to different rows in the same bank result in high latency. Therefore, to maximize bandwidth, current DRAM controllers schedule accesses to the same row in a bank before scheduling the accesses to a different row even if those were generated earlier in time. We will later show how this policy causes unfairness in the DRAM system and makes the system vulnerable to DoS attacks.

2.2.2 DRAM Controller

The DRAM controller is the mediator between the on-chip caches and the off-chip DRAM memory. It receives read/write requests from L2 caches. The addresses of these requests are at the granularity of the L2 cache block. Figure 3 shows the architecture of the DRAM controller. The main components of the controller are the *memory request buffer* and the *memory access scheduler*.

The memory request buffer buffers the requests received for each bank. It consists of separate *bank request buffers*. Each entry in a bank request buffer contains the address (row and column), the type (read or write), the timestamp, and the state of the request along with storage for the data associated with the request.

The memory access scheduler is the brain of the memory controller. Its main function is to select a memory request from the memory request buffer to be sent to DRAM memory. It has a two-level hierarchical organization as shown in Figure 3. The first level consists of separate per-bank schedulers. Each bank scheduler keeps track of the state of the bank and selects the highest-priority request from its bank request buffer. The second level consists of an across-bank scheduler that selects the highest-priority request among all the requests selected by the bank schedulers. When a request is scheduled by the memory access scheduler, its state is updated in the bank request buffer, and it is removed from the buffer when the request is served by the bank (For simplicity, these control paths are not shown in Figure 3).

2.2.3 Memory Access Scheduling Algorithm

Current memory access schedulers are designed to maximize the bandwidth obtained from the DRAM memory. As shown in [29], a simple request scheduling algorithm that serves requests based on a first-come-first-serve policy is prohibitive, because it incurs a large number of bank conflicts. Instead, current memory access schedulers usually employ what is called a First-Ready First-Come-First-Serve (FR-FCFS) algorithm to select which request should be scheduled next [29, 22]. This algorithm prioritizes requests in the following order in a bank:

1. **Row-hit-first**: A bank scheduler gives higher priority to the requests that would be serviced faster. In other words, a request that would result in a *row hit* is prioritized over one that would cause a *row conflict*.
2. **Oldest-within-bank-first**: A bank scheduler gives higher priority to the request that arrived earliest.

Selection from the requests chosen by the bank schedulers is done as follows:

Oldest-across-banks-first: The across-bank DRAM bus scheduler selects the request with the earliest arrival time among all the requests selected by individual bank schedulers.

In summary, this algorithm strives to maximize DRAM bandwidth by scheduling accesses that cause row hits first (regardless of when these requests have arrived) within a bank. Hence, streaming memory access patterns are prioritized within the memory system. The oldest row-hit request has the highest priority in the memory access scheduler. In contrast, the youngest row-conflict request has the lowest priority.

2.3 Vulnerability of the Multi-Core DRAM Memory System to DoS Attacks

As described above, current DRAM memory systems do not distinguish between the requests of different threads (i.e. cores)[3]. Therefore, multi-core systems are vulnerable to DoS attacks that exploit unfairness in the memory system. Requests from a thread with a particular access pattern can get prioritized by the memory access scheduler over requests from other threads, thereby causing the other threads to experience very long delays. We find that there are two major reasons why one thread can deny service to another in current DRAM memory systems:

1. **Unfairness of row-hit-first scheduling**: A thread whose accesses result in row hits gets higher priority compared to a thread whose accesses result in row conflicts. We call an access pattern that mainly results in row hits as a pattern with *high row-buffer locality*. Thus, an application that has a high row-buffer locality (e.g. one that is streaming through memory) can significantly delay another application with low row-buffer locality if they happen to be accessing the same DRAM banks.
2. **Unfairness of oldest-first scheduling**: Oldest-first scheduling implicitly gives higher priority to those threads that can generate memory requests at a faster rate than others. Such aggressive threads can flood the memory system with requests at a faster rate than the memory system can service. As such, aggressive threads can fill the memory system's buffers with their requests, while less memory-intensive threads are blocked from the memory system until all the earlier-arriving requests from the aggressive threads are serviced.

Based on this understanding, it is possible to develop a memory performance hog that effectively denies service to other threads. In the next section, we describe such an example MPH and show its impact on another application.

3 Motivation: Examples of Denial of Memory Service in Existing Multi-Cores

In this section, we present measurements from real systems to demonstrate that Denial of Memory Service attacks are possible in existing multi-core systems.

3.1 Applications

We consider two applications to motivate the problem. One is a modified version of the popular *stream* benchmark [19], an application that streams through memory and performs operations on two one-dimensional arrays. The arrays in *stream* are sized such that they are much larger than the L2 cache on a core. Each array consists of 2.5M 128-byte elements.[4] *Stream* (Figure 4(a)) has very high row-buffer locality since consecutive cache misses almost always access the same row (limited only by the size of the row-buffer). Even though we cannot directly measure the row-buffer hit rate in our real experimental system (because hardware does not directly provide this information), our simulations show that 96% of all memory requests in *stream* result in row-hits.

```
// initialize arrays a, b
for (j=0; j<N; j++)
   index[j] = j;       // streaming index
•••
for (j=0; j<N; j++)
   a[index[j]] = b[index[j]];
for (j=0; j<N; j++)
   b[index[j]] = scalar * a[index[j]];
•••
```

(a) STREAM

```
// initialize arrays a, b
for (j=0; j<N; j++)
   index[j] = rand();  // random # in [0,N]
•••
for (j=0; j<N; j++)
   a[index[j]] = b[index[j]];
for (j=0; j<N; j++)
   b[index[j]] = scalar * a[index[j]];
•••
```

(b) RDARRAY

Figure 4: Major loops of the *stream* (a) and *rdarray* (b) programs

The other application, called *rdarray*, is almost the exact opposite of *stream* in terms of its row-buffer locality. Its pseudo-code is shown in Figure 4(b). Although it performs the same operations on two very large arrays (each consisting of 2.5M 128-byte elements), *rdarray* accesses the arrays in a pseudo-random fashion. The array indices accessed in each iteration of the benchmark's main loop are determined using a pseudo-random number

[3] We assume, without loss of generality, one core can execute one thread.

[4] Even though the elements are 128-byte, each iteration of the main loop operates on only one 4-byte integer in the 128-byte element. We use 128-byte elements to ensure that consecutive accesses miss in the cache and exercise the DRAM memory system.

generator. Consequently, this benchmark has very low row-buffer locality; the likelihood that any two outstanding L2 cache misses in the memory request buffer are to the same row in a bank is low due to the pseudo-random generation of array indices. Our simulations show that 97% of all requests in *rdarray* result in row-conflicts.

3.2 Measurements

We ran the two applications alone and together on two existing multi-core systems and one simulated future multi-core system.

3.2.1 A Dual-core System

The first system we examine is an Intel Pentium D 930 [16] based dual-core system with 2GB SDRAM. In this system each core has an L2 cache size of 2MB. Only the DRAM memory system is shared between the two cores. The operating system is Windows XP Professional.[5] All the experiments were performed when the systems were unloaded as much as possible. To account for possible variability due to system state, each run was repeated 10 times and the execution time results were averaged (error bars show the variance across the repeated runs). Each application's main loops consist of $N = 2.5 \cdot 10^6$ iterations and were repeated 1000 times in the measurements.

Figure 5(a) shows the normalized execution time of *stream* when run (1) alone, (2) concurrently with another copy of *stream*, and (3) concurrently with *rdarray*. Figure 5(b) shows the normalized execution time of *rdarray* when run (1) alone, (2) concurrently with another copy of *rdarray*, and (3) concurrently with *stream*.

When *stream* and *rdarray* execute concurrently on the two different cores, *stream* is slowed down by only 18%. In contrast, *rdarray* experiences a dramatic slowdown: its execution time increases by up to 190% (it takes 2.9X longer to complete compared to when it is run alone). Hence, *stream* effectively denies memory service to *rdarray* without being significantly slowed down itself.

We hypothesize that this behavior is due to the row-hit-first scheduling policy in the DRAM memory controller. As most of *stream*'s memory requests hit in the row-buffer, they are prioritized over *rdarray*'s requests, most of which result in row conflicts. Consequently, *rdarray* is denied access to the DRAM banks that are being accessed by *stream* until the *stream* program's access pattern moves on to another bank. With a row size of 8KB and a cache line size of 64B, 128 (=8KB/64B) of *stream*'s memory requests can be serviced by a DRAM bank before *rdarray* is allowed to access that bank![6] Thus, due to the thread-unfair implementation of the DRAM memory system, *stream* can act as an MPH against *rdarray*.

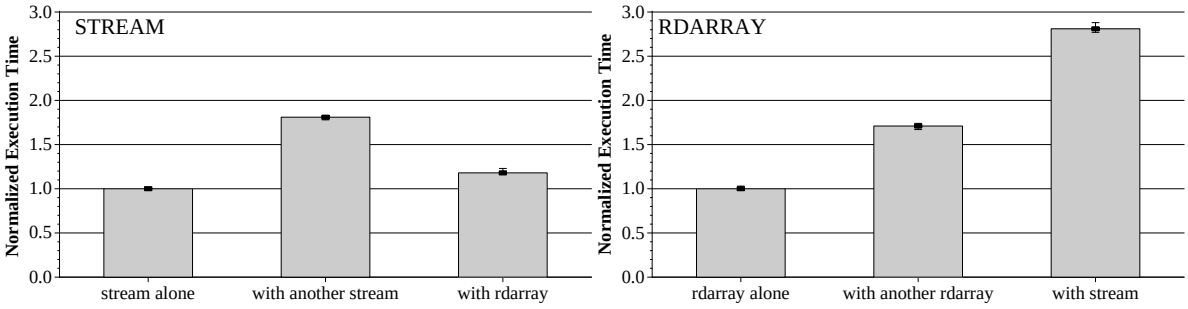

Figure 5: Normalized execution time of (a) *stream* and (b) *rdarray* when run alone/together on a dual-core system

Note that the slowdown *rdarray* experiences when run with *stream* (2.90X) is much greater than the slowdown it experiences when run with another copy of *rdarray* (1.71X). Because neither copy of *rdarray* has good row-

[5] We also repeated the same experiments in (1) the same system with the RedHat Fedora Core 6 operating system and (2) an Intel Core Duo based dual-core system running RedHat Fedora Core 6. We found the results to be almost exactly the same as those reported.

[6] Note that we do not know the exact details of the DRAM memory controller and scheduling algorithm that is implemented in the existing systems. These details are not made public in either Intel's or AMD's documentation. Therefore, we hypothesize about the causes of the behavior based on public information available on DRAM memory systems - and later support our hypotheses with our simulation infrastructure (see Section 6). It could be possible that existing systems have a threshold up to which younger requests can be ordered over older requests as described in a patent [20], but even so our experiments suggest that memory performance attacks are still possible in existing multi-core systems.

buffer locality, another copy of *rdarray* cannot deny service to *rdarray* by holding on to a row-buffer for a long time. In this case, the performance loss comes from increased bank conflicts and contention in the DRAM bus.

On the other hand, the slowdown *stream* experiences when run with *rdarray* is significantly smaller than the slowdown it experiences when run with another copy of *stream*. When two copies of *stream* run together they are both able to deny access to each other because they both have very high row-buffer locality. Because the rates at which both *stream*s generate memory requests are the same, the slowdown is not as high as *rdarray*'s slowdown with *stream*: copies of *stream* take turns in denying access to each other (in different DRAM banks) whereas *stream always* denies access to *rdarray* (in all DRAM banks).

3.2.2 A Dual Dual-core System

The second system we examine is a dual dual-core AMD Opteron 275 [1] system with 4GB SDRAM. In this system, only the DRAM memory system is shared between a total of four cores. Each core has an L2 cache size of 1 MB. The operating system used was RedHat Fedora Core 5. Figure 6(a) shows the normalized execution time of *stream* when run (1) alone, (2) with one copy of *rdarray*, (3) with 2 copies of *rdarray*, (4) with 3 copies of *rdarray*, and (5) with 3 other copies of *stream*. Figure 6(b) shows the normalized execution time of *rdarray* in similar but "dual" setups.

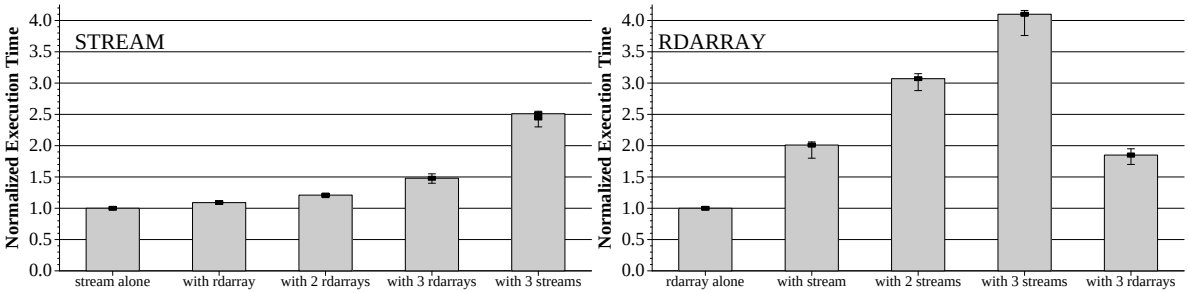

Figure 6: Slowdown of (a) *stream* and (b) *rdarray* when run alone/together on a dual dual-core system

Similar to the results shown for the dual-core Intel system, the performance of *rdarray* degrades much more significantly than the performance of *stream* when the two applications are executed together on the 4-core AMD system. In fact, *stream* slows down by only 48% when it is executed concurrently with 3 copies of *rdarray*. In contrast, *rdarray* slows down by 408% when running concurrently with 3 copies of *stream*. Again, we hypothesize that this difference in slowdowns is due to the row-hit-first policy employed in the DRAM controller.

3.2.3 A Simulated 16-core System

While the problem of MPHs is severe even in current dual- or dual-dual-core systems, it will be significantly aggravated in future multi-core systems consisting of many more cores. To demonstrate the severity of the problem, Figure 7 shows the normalized execution time of *stream* and *rdarray* when run concurrently with 15 copies of *stream* or 15 copies of *rdarray*, along with their normalized execution times when 8 copies of each application are run together. Note that our simulation methodology and simulator parameters are described in Section 6.1. In a 16-core system, our memory performance hog, *stream*, slows down *rdarray* by 14.6X while *rdarray* slows down *stream* by only 4.4X. Hence, *stream* is an even more effective performance hog in a 16-core system, indicating that the problem of "memory performance attacks" will become more severe in the future if the memory system is not adjusted to prevent them.

4 Towards a Solution: Fairness in DRAM Memory Systems

The fundamental unifying cause of the attacks demonstrated in the previous section is *unfairness* in the shared DRAM memory system. The problem is that the memory system cannot distinguish whether a harmful memory access pattern issued by a thread is due to a malicious attack, due to erroneous programming, or simply a necessary memory behavior of a specific application. Therefore, the best the DRAM memory scheduler can do is to *contain and limit* memory attacks by providing fairness among different threads.

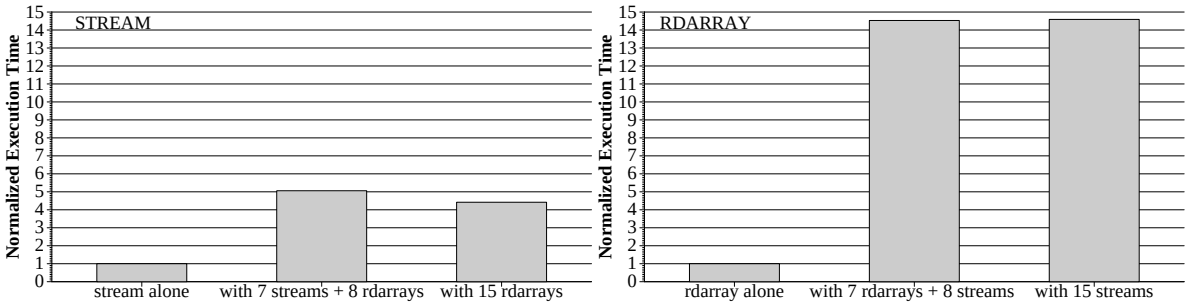

Figure 7: Slowdown of (a) *stream* and (b) *rdarray* when run alone and together on a simulated 16-core system

Difficulty of Defining DRAM Fairness: But what exactly constitutes fairness in DRAM memory systems? As it turns out, answering this question is non-trivial and coming up with a reasonable definition is somewhat problematic. For instance, simple algorithms that schedule requests in such a way that memory latencies are equally distributed among different threads disregard the fact that different threads have different row-buffer latencies. As a consequence, such *equal-latency scheduling algorithms* will unduly slow down threads that have high row-buffer locality and prioritize threads that have poor row-buffer locality. Whereas the standard FR-FCFS scheduling algorithm can starve threads with bad row-buffer locality (Section 2.3), any algorithm seeking egalitarian memory fairness would unfairly punish "well-behaving" threads with good row-buffer locality. Neither of the two options therefore rules out unfairness and the possibility of memory attacks.

Another challenge is that DRAM memory systems have a notion of *state* (consisting of the currently buffered rows in each bank). For this reason, well-studied notions of fairness that deal with stateless systems cannot be applied in our setting. In *network fair queuing* [23, 36, 3], for example, the idea is that if N processes share a common channel with bandwidth B, every process should achieve exactly the same performance as if it had a single channel of bandwidth B/N. When mapping the same notion of fairness onto a DRAM memory system (as done in [22]), however, the memory scheduler would need to schedule requests in such a way as to guarantee the following: *In a multi-core system with N threads, no thread should run slower than the same thread on a single-core system with a DRAM memory system that runs at $1/N$th of the speed.* Unfortunately, because memory banks have state and row conflicts incur a higher latency than row hit accesses, this notion of fairness is ill-defined. Consider for instance two threads in a dual-core system that constantly access the same bank but different rows. While each of these threads by itself has perfect row-buffer locality, running them together will inevitably result in row-buffer conflicts. Hence, it is impossible to schedule these threads in such a way that each thread runs at the same speed as if it ran by itself on a system at half the speed. On the other hand, requests from two threads that consistently access different banks could (almost) entirely be scheduled in parallel and there is no reason why the memory scheduler should be allowed to slow these threads down by a factor of 2.

In summary, in the context of memory systems, notions of fairness–such as network fair queuing–that attempt to equalize the latencies experienced by different threads are unsuitable. In a DRAM memory system, it is neither possible to achieve such a fairness nor would achieving it significantly reduce the risk of memory performance attacks. In Section 4.1, we will present a novel definition of DRAM fairness that takes into account the inherent row-buffer locality of threads and attempts to balance the "relative slowdowns".

The Idleness Problem: In addition to the above observations, it is important to observe that any scheme that tries to balance latencies between threads runs into the risk of what we call the *idleness problem*. Threads that are temporarily idle (not issuing many memory requests, for instance due to an I/O operation) will be slowed down when returning to a more memory intensive access pattern. On the other hand, in certain solutions based on network fair queuing [22], a memory hog could intentionally issue no or few memory requests for a period of time. During that time, other threads could "move ahead" at a proportionally lower latency, such that, when the malicious thread returns to an intensive access pattern, it is temporarily prioritized and normal threads are blocked. The idleness problem therefore poses a severe security risk: By exploiting it, an attacking memory hog could temporarily slow down or even block time-critical applications with high performance stability requirements from memory.

4.1 Fair Memory Scheduling: A Model

As discussed, standard notions of fairness fail in providing fair execution and hence, security, when mapping them onto shared memory systems. The crucial insight that leads to a better notion of fairness is that we need to *dissect* the memory latency experienced by a thread into two parts: First, the latency that is inherent to the thread itself (depending on its row-buffer locality) and second, the latency that is caused by contention with other threads in the shared DRAM memory system. A fair memory system should—unlike the approaches so far—schedule requests in such a way that the *second* latency component is fairly distributed, while the first component remains untouched. With this, it is clear why our novel notion of *DRAM shared memory fairness* is based on the following intuition: *In a multi-core system with N threads, no thread should suffer more relative performance slowdown—compared to the performance it gets if it used the same memory system by itself—than any other thread.* Because each thread's slowdown is thus measured against its own baseline performance (single execution on the same system), this notion of fairness successfully dissects the two components of latency and takes into account the inherent characteristics of each thread.

In more technical terms, we consider a measure χ_i for each currently executed thread i.[7] This measure captures the price (in terms of relative additional latency) a thread i pays because the shared memory system is used by multiple threads in parallel in a multi-core architecture. In order to provide fairness and contain the risk of denial of memory service attacks, the memory controller should schedule outstanding requests in the buffer in such a way that the χ_i values are as balanced as possible. Such a scheduling will ensure that each thread only suffers a fair amount of additional latency that is caused by the parallel usage of the shared memory system.

Formal Definition: Our definition of the measure χ_i is based on the notion of *cumulated bank-latency* $L_{i,b}$ that we define as follows.

Definition 4.1. *For each thread i and bank b, the* cumulated bank-latency $L_{i,b}$ *is the number of memory cycles during which there exists an outstanding memory request by thread i for bank b in the memory request buffer. The cumulated latency of a thread $L_i = \sum_b L_{i,b}$ is the sum of all cumulated bank-latencies of thread i.*

The motivation for this formulation of $L_{i,b}$ is best seen when considering latencies on the level of individual memory requests. Consider a thread i and let $R^k_{i,b}$ denote the kth memory request of thread i that accesses bank b. Each such request $R^k_{i,b}$ is associated with three specific times: Its arrival time $a^k_{i,b}$ when it is entered into the request buffer; its finish time $f^k_{i,b}$, when it is completely serviced by the bank and sent to processor i's cache; and finally, the request's *activation time*

$$s^k_{i,b} := \max\{f^{k-1}_{i,b}, a^k_{i,b}\}.$$

This is the earliest time when request $R^k_{i,b}$ could be scheduled by the bank scheduler. It is the larger of its arrival time and the finish time of the previous request $R^{k-1}_{i,b}$ that was issued by the same thread to the same bank. A request's activation time marks the point in time from which on $R^k_{i,b}$ is responsible for the ensuing latency of thread i; before $s^k_{i,b}$, the request was either not sent to the memory system or an earlier request to the same bank by the same thread was generating the latency. With these definitions, the *amortized latency* $\ell^k_{i,b}$ of request $R^k_{i,b}$ is the difference between its finish time and its activation time, i.e., $\ell^k_{i,b} = f^k_{i,b} - s^k_{i,b}$. By the definition of the activation time $s^k_{i,b}$, it is clear that at any point in time, the amortized latency of exactly one outstanding request is increasing (if there is at least one in the request buffer). Hence, when describing time in terms of executed memory cycles, our definition of cumulated bank-latency $L_{i,b}$ corresponds exactly to the sum over all amortized latencies to this bank, i.e., $L_{i,b} = \sum_k \ell^k_{i,b}$.

In order to compute the experienced slowdown of each thread, we compare the actual experienced cumulated latency L_i of each thread i to an imaginary, *ideal single-core cumulated latency* $\widetilde{L}_i$ that serves as a baseline. This latency $\widetilde{L}_i$ is the minimal cumulated latency that thread i would have accrued if it had run as the only thread in the system using the same DRAM memory; it captures the latency component of L_i that is inherent to the thread itself and not caused by contention with other threads. Hence, threads with good and bad row-buffer locality

[7] The DRAM memory system only keeps track of threads that are currently issuing requests.

have small and large $\widetilde{L}_i$, respectively. The measure χ_i that captures the relative slowdown of thread i caused by multi-core parallelism can now be defined as follows.

Definition 4.2. *For a thread i, the* DRAM memory slowdown index χ_i *is the ratio between its cumulated latency L_i and its ideal single-core cumulated latency $\widetilde{L}_i$:*[8]

$$\chi_i := L_i / \widetilde{L}_i.$$

Finally, we define the **DRAM unfairness** Ψ of a DRAM memory system as the ratio between the maximum and minimum slowdown index over all currently executed threads in the system:

$$\Psi := \frac{\max_i \chi_i}{\min_j \chi_j}$$

The "ideal" DRAM unfairness index $\Psi = 1$ is achieved if all threads experience exactly the same slowdown; the higher Ψ, the more unbalanced is the experienced slowdown of different threads. The goal of a fair memory access scheduling algorithm is therefore to achieve a Ψ that is as close to 1 as possible. This ensures that no thread is over-proportionally slowed down due to the shared nature of DRAM memory in multi-core systems.

Notice that by taking into account the different row-buffer localities of different threads, our definition of DRAM unfairness prevents punishing threads for having either good or bad memory access behavior. Hence, a scheduling algorithm that achieves low DRAM unfairness mitigates the risk that any thread in the system, regardless of its bank and row access pattern, is unduly bogged down by other threads. Notice further that DRAM unfairness is virtually unaffected by the idleness problem, because both cumulated latencies L_i and ideal single-core cumulated latencies $\widetilde{L}_i$ are only accrued when there are requests in the memory request buffer.

Short-Term vs. Long-Term Fairness: So far, the aspect of time-scale has remained unspecified in our definition of DRAM-unfairness. Both L_i and $\widetilde{L}_i$ continue to increase throughout the lifetime of a thread. Consequently, a short-term unfair treatment of a thread would have increasingly little impact on its slowdown index χ_i. While still providing long-term fairness, threads that have been running for a long time could become vulnerable to short-term DoS attacks even if the scheduling algorithm enforced an upper bound on DRAM unfairness Ψ. In this way, delay-sensitive applications could be blocked from DRAM memory for limited periods of time.

We therefore generalize all our definitions to include an additional parameter T that denotes the time-scale for which the definitions apply. In particular, $L_i(T)$ and $\widetilde{L}_i(T)$ are the maximum (ideal single-core) cumulated latencies over all time-intervals of duration T during which thread i is active. Similarly, $\chi_i(T)$ and $\Psi(T)$ are defined as the maximum values over all time-intervals of length T. The parameter T in these definitions determines how short- or long-term the considered fairness is. In particular, a memory scheduling algorithm with good long term fairness will have small $\Psi(T)$ for large T, but possibly large $\Psi(T')$ for smaller T'. In view of the security issues raised in this paper, it is clear that a memory scheduling algorithm should aim at achieving small $\Psi(T)$ for both small and large T.

5 Our Solution

In this section, we propose FairMem, a new fair memory scheduling algorithm that achieves good fairness according to the definition in Section 4 and hence, reduces the risk of memory-related DoS attacks.

5.1 Basic Idea

The reason why MPHs can exist in multi-core systems is the unfairness in current memory access schedulers. Therefore, the idea of our new scheduling algorithm is to enforce fairness by balancing the relative memory-related slowdowns experienced by different threads. The algorithm schedules requests in such a way that each thread experiences a similar degree of memory-related slowdown relative to its performance when run alone.

[8] Notice that our definitions do not take into account the service and waiting times of the shared DRAM bus and across-bank scheduling. Both our definition of fairness as well as our algorithm presented in Section 5 can be extended to take into account these and other more subtle hardware issues. As the main goal of this paper point out and investigate potential security risks caused by DRAM unfairness, our model abstracts away numerous aspects of secondary importance because our definition provides a good approximation.

In order to achieve this goal, the algorithm maintains a value (χ_i in our model of Section 4.1) that characterizes the relative slowdown of each thread. As long as all threads have roughly the same slowdown, the algorithm schedules requests using the regular FR-FCFS mechanism. When the slowdowns of different threads start diverging and the difference exceeds a certain threshold (i.e., when Ψ becomes too large), however, the algorithm switches to an alternative scheduling mechanism and starts prioritizing requests issued by threads experiencing large slowdowns.

5.2 Fair Memory Scheduling Algorithm (FairMem)

The memory scheduling algorithm we propose for use in DRAM controllers for multi-core systems is defined by means of two input parameters, α and β. These parameters can be used to fine-tune the involved trade-offs between fairness and throughput on the one hand (α) and short-term versus long-term fairness on the other (β). More concretely, α is a parameter that expresses to what extent the scheduler is allowed to optimize for DRAM throughput at the cost of fairness, i.e., how much DRAM unfairness is tolerable. The parameter β corresponds to the time-interval T that denotes the time-scale of the above fairness condition. In particular, the memory controller divides time into windows of duration β and, for each thread maintains an accurate account of its accumulated latencies $L_i(\beta)$ and $\widetilde{L}_i(\beta)$ in the current time window.[9]

Instead of using the (FR-FCFS) algorithm described in Section 2.2.3, our algorithm first determines two *candidate requests* from each each bank b, one according to each of the following rules:

- **Highest FR-FCFS priority**: Let $R_{\text{FR-FCFS}}$ be the request to bank b that has the highest priority according to the FR-FCFS scheduling policy of Section 2.2.3. That is, row hits have higher priority than row conflicts, and—given this partial ordering—the oldest request is served first.
- **Highest fairness-index**: Let i' be the thread with highest current DRAM memory slowdown index $\chi_{i'}(\beta)$ that has at least one outstanding request in the memory request buffer to bank b. Among all requests to b issued by i', let R_{Fair} be the one with highest FR-FCFS priority.

Between these two candidates, the algorithm chooses the request to be scheduled based on the following rule:

- **Fairness-oriented Selection**: Let $\chi_\ell(\beta)$ and $\chi_s(\beta)$ denote largest and smallest DRAM memory slowdown index of any request in the memory request buffer for a current time window of duration β. If it holds that

$$\frac{\chi_\ell(\beta)}{\chi_s(\beta)} \;\geq\; \alpha$$

  then R_{Fair} is selected by bank b's scheduler and $R_{\text{FR-FCFS}}$ otherwise.

Instead of using the oldest-across-banks-first strategy as used in current DRAM memory schedulers, selection from requests chosen by the bank schedulers is handled as follows:

Highest-DRAM-fairness-index-first across banks: The request with highest slowdown index $\chi_i(\beta)$ among all selected bank-requests is sent on the shared DRAM bus.

In principle, the algorithm is built to ensure that at no time DRAM unfairness $\Psi(\beta)$ exceeds the parameter α. Whenever there is the risk of exceeding this threshold, the memory controller will switch to a mode in which it starts prioritizing threads with higher χ_i values, which decreases χ_i. It also increases the χ_j values of threads that have had little slowdown so far. Consequently, this strategy balances large and small slowdowns, which decreases DRAM unfairness and—as shown in Section 6—keeps potential memory-related DoS attacks in check.

Notice that this algorithm does not–in fact, cannot–guarantee that the DRAM unfairness Ψ does stay below the predetermined threshold α at all times. The impossibility of this can be seen when considering the corner-case $\alpha = 1$. In this case, a violation occurs after the first request regardless of which request is scheduled by the algorithm. On the other hand, the algorithm always attempts to keep the necessary violations as small as possible.

[9]Notice that in principle, there are various possibilities of interpreting the term "current time window". The simplest way is to completely reset $L_i(\beta)$ and $\widetilde{L}_i(\beta)$ after each completion of a window. More sophisticated techniques could include maintaining multiple, say k, such windows of size β in parallel, each shifted in time by β/k memory cycles. In this case, all windows are constantly updated, but only the oldest is used for the purpose of decision-making. This could help in reducing volatility.

Another advantage of our scheme is that an approximate version of it lends itself to efficient implementation in hardware. Finally, notice that our algorithm is robust with regard to the *idleness problem* mentioned in Section 4. In particular, neither L_i nor $\widetilde{L}_i$ is increased or decreased if a thread has no outstanding memory requests in the request buffer. Hence, not issuing any requests for some period of time (either intentionally or unintentionally due to I/O, for instance) does not affect this or any other thread's priority in the buffer.

5.3 Hardware Implementations

The algorithm as described so far is abstract in the sense that it assumes a memory controller that always has full knowledge of every active (currently-executed) thread's L_i and $\widetilde{L}_i$. In this section, we show how this exact scheme could be implemented, and we also briefly discuss a more efficient practical hardware implementation.

Exact Implementation: Theoretically, it is possible to ensure that the memory controller always keeps accurate information of $L_i(\beta)$ and $\widetilde{L}_i(\beta)$. Keeping track of $L_i(\beta)$ for each thread is simple. For each active thread, a counter maintains the number of memory cycles during which at least one request of this thread is buffered for each bank. After completion of the window β (or when a new thread is scheduled on a core), counters are reset. The more difficult part of maintaining an accurate account of $\widetilde{L}_i(\beta)$ can be done as follows: At all times, maintain for each active thread i and for each bank the row that would currently be in the row-buffer if i had been the only thread using the DRAM memory system. This can be done by simulating an FR-FCFS priority scheme for each thread and bank that ignores all requests issued by threads other than i. The $\widetilde{\ell}^k_{i,b}$ latency of each request $R^k_{i,b}$ then corresponds to the latency this request would have caused if DRAM memory was not shared. Whenever a request is served, the memory controller can add this "ideal latency" to the corresponding $\widetilde{L}_{i,b}(\beta)$ of that thread and–if necessary–update the simulated state of the row-buffer accordingly. For instance, assume that a request $R^k_{i,b}$ is served, but results in a row conflict. Assume further that the same request would have been a row hit, if thread i had run by itself, i.e., $R^{k-1}_{i,b}$ accesses the same row as $R^k_{i,b}$. In this case, $\widetilde{L}_{i,b}(\beta)$ is increased by row-hit latency T_{hit}, whereas $L_{i,b}(\beta)$ is increased by the bank-conflict latency T_{conf}. By thus "simulating" its own execution for each thread, the memory controller obtains accurate information for all $\widetilde{L}_{i,b}(\beta)$.

The obvious problem with the above implementation is that it is expensive in terms of hardware overhead and cost. It requires maintaining at least one counter for each core×bank pair. Similarly severe, it requires one divider per core in order to compute the value $\chi_i(\beta) = L_i(\beta)/\widetilde{L}_i(\beta)$ for the thread that is currently running on that core in every memory cycle. Fortunately, much less expensive hardware implementations are possible because the memory controller does not need to know the exact values of $L_{i,b}$ and $\widetilde{L}_{i,b}$ at any given moment. Instead, using reasonably accurate approximate values suffices to maintain an excellent level of fairness and security.

Reduce counters by sampling: Using sampling techniques, the number of counters that need to be maintained can be reduced from $O(\#\mathrm{Banks} \times \#\mathrm{Cores})$ to $O(\#\mathrm{Cores})$ with only little loss in accuracy. Briefly, the idea is the following. For each core and its active thread, we keep two counters S_i and H_i denoting the number of samples and sampled hits, respectively. Instead of keeping track of the exact row that would be open in the row-buffer if a thread i was running alone, we randomly sample a subset of requests $R^k_{i,b}$ issued by thread i and check whether the next request by i to the same bank, $R^{k+1}_{i,b}$, is for the same row. If so, the memory controller increases both S_i and H_i, otherwise, only S_i is increased. Requests $R^q_{i,b'}$ to different banks $b' \neq b$ served between $R^k_{i,b}$ and $R^{k+1}_{i,b}$ are ignored. Finally, if none of the Q requests of thread i following $R^k_{i,b}$ go to bank b, the sample is discarded, neither S_i nor H_i is increased, and a new sample request is taken. With this technique, the probability H_i/S_i that a request results in a row hit gives the memory controller a reasonably accurate picture of each thread's row-buffer locality. An approximation of $\widetilde{L}_i$ can thus be maintained by adding the expected amortized latency to it whenever a request is served, i.e.,

$$\widetilde{L}_i^{new} := \widetilde{L}_i^{old} + (H_i/S_i \cdot T_{hit} + (1 - H_i/S_i) \cdot T_{conf}) \,.$$

Reuse dividers: The ideal scheme employs $O(\#\mathrm{Cores})$ hardware dividers, which significantly increases the memory controller's energy consumption. Instead, a single divider can be used for all cores by assigning individual threads to it in a round robin fashion. That is, while the slowdowns $L_i(\beta)$ and $\widetilde{L}_i(\beta)$ can be updated in every memory cycle, their quotient $\chi_i(\beta)$ is recomputed in intervals.

6 Evaluation

6.1 Experimental Methodology

We evaluate our solution using a detailed processor and memory system simulator based on the Pin dynamic binary instrumentation tool [18]. Our in-house instruction-level performance simulator can simulate applications compiled for the x86 instruction set architecture. We simulate the memory system in detail using a model loosely based on DRAMsim [33], which is commonly used for DRAM performance evaluations. Both our processor model and the memory model mimmick the design of a modern high-performance dual-core processor based on Intel Core Duo [10]. The size/bandwidth/latency/capacity of different processor structures along with the number of cores and other structures are parameters to the simulator. The simulator faithfully models the bandwidth, latency, and capacity of each buffer, bus, and structure in the memory subsystem (including the caches, memory controller, DRAM buses, and DRAM banks). The relevant parameters of the modeled baseline processor are shown in Table 1. Unless otherwise stated, all evaluations in this section are performed on a simulated dual-core system using these parameters. For our measurements with the FairMem system presented in Section 5, the parameters are set to $\alpha = 1.025$ and $\beta = 10^5$.

Processor pipeline	4 GHz processor, 128-entry instruction window, 12-stage pipeline
Fetch/Execute width per core	3 instructions can be fetched/executed every cycle; only 1 can be a memory operation
L1 Caches	32 K-byte per-core, 4-way set associative, 32-byte block size, 2-cycle latency
L2 Caches	512 K-byte per core, 8-way set associative, 32-byte block size, 12-cycle latency
Memory controller	128 request buffer entries, FR-FCFS baseline scheduling policy, runs at 2 GHz
DRAM parameters	8 banks, 2K-byte row-buffer
DRAM latency (round-trip L2 miss latency)	row-buffer hit: 50ns (200 cycles), closed: 75ns (300 cycles), conflict: 100ns (400 cycles)

Table 1: Baseline processor configuration

We simulate each application for 100 million x86 instructions. The portions of applications that are simulated are determined using the SimPoint tool [31], which selects simulation points in the application that are representative of the application's behavior as a whole. Our applications include *stream* and *rdarray* (described in Section 3), several large benchmarks from the SPEC CPU2000 benchmark suite [32], and one memory-intensive application from the Olden benchmark suite [30]. These applications are described in Table 2.

Benchmark	Suite	Brief description	Base performance	L2-misses per 1K inst.	row-buffer hit rate
stream	Microbenchmark	Streaming on 32-byte-element arrays	46.30 cycles/inst.	629.65	96%
rdarray	Microbenchmark	Random access on arrays	56.29 cycles/inst.	629.18	3%
small-stream	Microbenchmark	Streaming on 4-byte-element arrays	13.86 cycles/inst.	71.43	97%
art	SPEC 2000 FP	Object recognition in thermal image	7.85 cycles/inst.	70.82	88%
crafty	SPEC 2000 INT	Chess game	0.64 cycles/inst.	0.35	15%
health	Olden	Columbian health care system simulator	7.24 cycles/inst.	83.45	27%
mcf	SPEC 2000 INT	Single-depot vehicle scheduling	4.73 cycles/inst.	45.95	51%
vpr	SPEC 2000 INT	FPGA circuit placement and routing	1.71 cycles/inst.	5.08	14%

Table 2: Evaluated applications and their performance characteristics on the baseline processor

6.2 Evaluation Results

6.2.1 Dual-core Systems

Two microbenchmark applications - stream and rdarray: Figure 8 shows the normalized execution time of *stream* and *rdarray* applications when run alone or together using either the baseline FR-FCFS or our FairMem memory scheduling algorithms. Execution time of each application is normalized to the execution time they experience when they are run alone using the FR-FCFS scheduling algorithm (This is true for all normalized results in this paper). When *stream* and *rdarray* are run together on the baseline system, *stream*—which acts as an MPH—experiences a slowdown of only 1.22X whereas *rdarray* slows down by 2.45X. In contrast, a memory controller that uses our FairMem algorithm prevents *stream* from behaving like an MPH against *rdarray* – both applications experience a similar slowdown when run together. FairMem does not significantly affect performance when the applications are run alone or when run with identical copies of themselves (i.e. when

the memory performance is not unfairly impacted). These experiments show that our simulated system closely matches the behavior we observe in an existing dual-core system (Figure 5), and that FairMem successfully provides fairness among threads. Next, we show that with real applications, the effect of an MPH can be drastic.

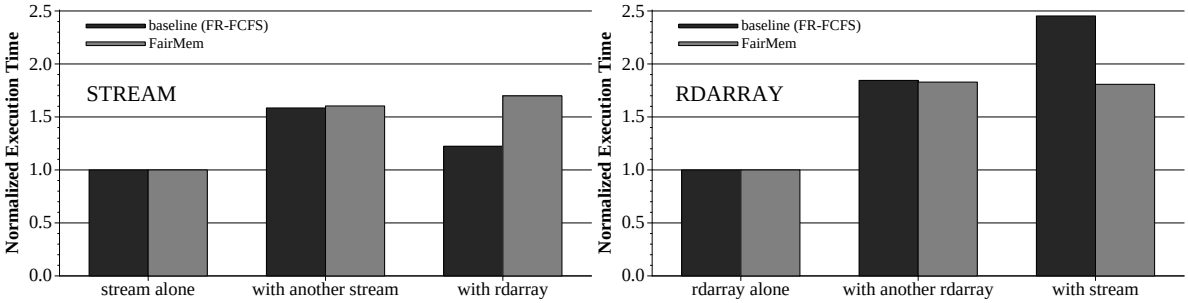

Figure 8: Slowdown of (a) *stream* and (b) *rdarray* benchmarks using FR-FCFS and our FairMem algorithm

Effect on real applications: Figure 9 shows the normalized execution time of 8 different pairs of applications when run alone or together using either the baseline FR-FCFS or FairMem. The results show that 1) an MPH can severely damage the performance of another application, and 2) our FairMem algorithm is effective at preventing it. For example, when *stream* and *health* are run together in the baseline system, *stream* acts as an MPH slowing down *health* by more than 8.5X while itself being slowed down by only 1.05X. This is because it has 7 times higher L2 miss rate and much higher row-buffer locality (96% vs. 25%) — therefore, it exploits unfairness in both row-buffer-hit first and oldest-first scheduling policies by flooding the memory system with its requests. When the two applications are run on our FairMem system, *health*'s slowdown is reduced from 8.63X to 2.28X. The figure also shows that even regular applications with high row-buffer locality can act as MPHs. For instance when *art* and *vpr* are run together in the baseline system, *art* acts as an MPH slowing down *vpr* by 2.35X while itself being slowed down by only 1.05X. When the two are run on our FairMem system, each slows down by only 1.35X; thus, *art* is no longer a performance hog.

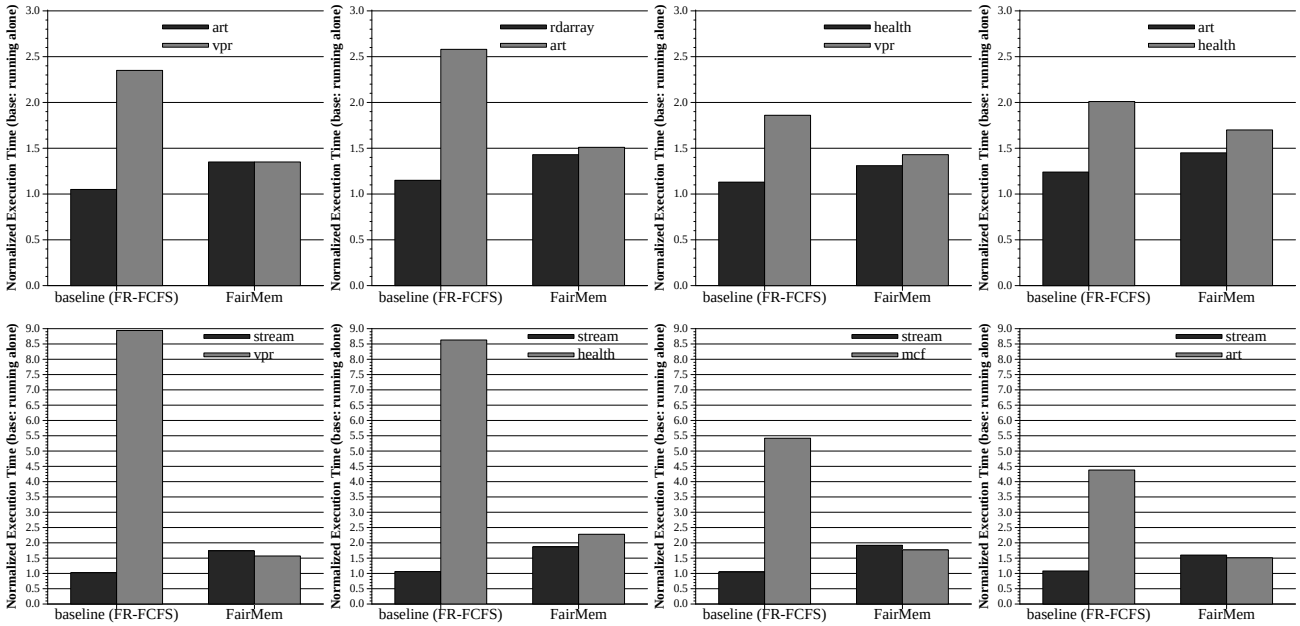

Figure 9: Slowdown of different application combinations using FR-FCFS and our FairMem algorithm

Effect on Throughput and Unfairness: Table 3 shows the overall throughput (in terms of executed instructions per 1000 cycles) and DRAM unfairness (relative difference between the maximum and minimum memory-related slowdowns, defined as Ψ in Section 4) when different application combinations are executed together. In all cases, FairMem reduces the unfairness to below 1.20 (Remember that 1.00 is the best possible Ψ value). Interestingly,

in most cases, FairMem also improves overall throughput significantly. This is especially true when a very memory-intensive application (e.g. *stream*) is run with a much less memory-intensive application (e.g. *vpr*).

Providing fairness leads to higher overall system throughput because it enables better utilization of the cores (i.e. better utilization of the multi-core system). The baseline FR-FCFS algorithm significantly hinders the progress of a less memory-intensive application, whereas FairMem allows this application to stall less due to the memory system, thereby enabling it to make fast progress through its instruction stream. Hence, rather than wasting execution cycles due to unfairly-induced memory stalls, some cores are better utilized with FairMem.[10] On the other hand, FairMem reduces the overall throughput by 9% when two extremely memory-intensive applications, *stream* and *rdarray*, are run concurrently. In this case, enforcing fairness reduces *stream*'s data throughput without significantly increasing the throughput of *rdarray* because *rdarray* encounters L2 cache misses as frequently as *stream* (as shown in Table 2).

Combination	**Baseline (FR-FCFS)**		**FairMem**		**Throughput improvement**	**Fairness improvement**
	Throughput	Unfairness	Throughput	Unfairness		
stream-rdarray	24.8	2.00	22.5	1.06	0.91X	1.89X
art-vpr	401.4	2.23	513.0	1.00	1.28X	2.23X
health-vpr	463.8	1.56	508.4	1.09	1.10X	1.43X
art-health	179.3	1.62	178.5	1.15	0.99X	1.41X
rdarray-art	65.9	2.24	97.1	1.06	1.47X	2.11X
stream-health	38.0	8.14	72.5	1.18	1.91X	6.90X
stream-vpr	87.2	8.73	390.6	1.11	4.48X	7.86X
stream-mcf	63.1	5.17	117.1	1.08	1.86X	4.79X
stream-art	51.2	4.06	98.6	1.06	1.93X	3.83X

Table 3: Effect of FairMem on overall throughput (in terms of instructions per 1000 cycles) and unfairness

6.2.2 Effect of Row-buffer Size

From the above discussions, it is clear that the exploitation of row-buffer locality by the DRAM memory controller makes the multi-core memory system vulnerable to DoS attacks. The extent to which this vulnerability can be exploited is determined by the size of the row-buffer. In this section, we examine the impact of row-buffer size on the effectiveness of our algorithm. For these sensitivity experiments we use two real applications, *art* and *vpr*, where *art* behaves as an MPH against *vpr*.

Figure 10 shows the mutual impact of *art* and *vpr* on machines with different row-buffer sizes. Additional statistics are presented in Table 4. As row-buffer size increases, the extent to which *art* becomes a memory performance hog for *vpr* increases when FR-FCFS scheduling algorithm is used. In a system with very small, 512-byte row-buffers, *vpr* experiences a slowdown of 1.65X (versus *art*'s 1.05X). In a system with very large, 64 KB row-buffers, *vpr* experiences a slowdown of 5.50X (versus *art*'s 1.03X). Because *art* has very high row-buffer locality, a large buffer size allows its accesses to occupy a bank much longer than a small buffer size does. Hence, *art*'s ability to deny bank service to *vpr* increases with row-buffer size. FairMem effectively contains this denial of service and results in similar slowdowns for both *art* and *vpr* (1.32X to 1.41X). It is commonly assumed that row-buffer sizes will increase in the future to allow more throughput for streaming applications [37]. As our results show, this implies that memory-related DoS attacks will become a larger problem and algorithms to prevent them will become more important.[11]

	512 B	1 KB	2 KB	4 KB	8 KB	16 KB	32 KB	64 KB
art's row-buffer hit rate	56%	67%	87%	91%	92%	93%	95%	98%
vpr's row-buffer hit rate	13%	15%	17%	19%	23%	28%	38%	41%
FairMem throughput improvement	1.08X	1.16X	1.28X	1.44X	1.62X	1.88X	2.23X	2.64X
FairMem fairness improvement	1.55X	1.75X	2.23X	2.42X	2.62X	3.14X	3.88X	5.13X

Table 4: Statistics for *art* and *vpr* with different row-buffer sizes

[10] Note that the data throughput obtained from the DRAM itself may be, and usually is reduced using FairMem. However, overall throughput in terms of instructions executed per cycle usually increases.

[11] Note that reducing the row-buffer size may at first seem like one way of reducing the impact of memory-related DoS attacks. However, this solution is not desirable because reducing the row-buffer size significantly reduces the memory bandwidth (hence performance) for applications with good row-buffer locality even when they are running alone or when they are not interfering with other applications.

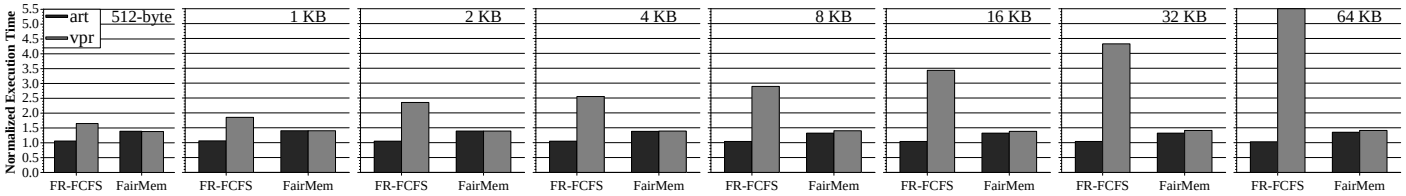

Figure 10: Normalized execution time of *art* and *vpr* when run together on processors with different row-buffer sizes. Execution time is independently normalized to each machine with different row-buffer size.

6.2.3 Effect of Number of Banks

The number of DRAM banks is another important parameter that affects how much two threads can interfere with each others' memory accesses. Figure 11 shows the impact of *art* and *vpr* on each other on machines with different number of DRAM banks. As the number of banks increases, the available parallelism in the memory system increases, and thus *art* becomes less of a performance hog; its memory requests conflict less with *vpr*'s requests. Regardless of the number of banks, our mechanism significantly mitigates the performance impact of *art* on *vpr* while at the same time improving overall throughput as shown in Table 5. Current DRAMs usually employ 4-16 banks because a larger number of banks increases the cost of the DRAM system. In a system with 4 banks, *art* slows down *vpr* by 2.64X (while itself being slowed down by only 1.10X). FairMem is able to reduce *vpr*'s slowdown to only 1.62X and improve overall throughput by 32%. In fact, Table 5 shows that FairMem achieves the same throughput on only 4 banks as the baseline scheduling algorithm on 8 banks.

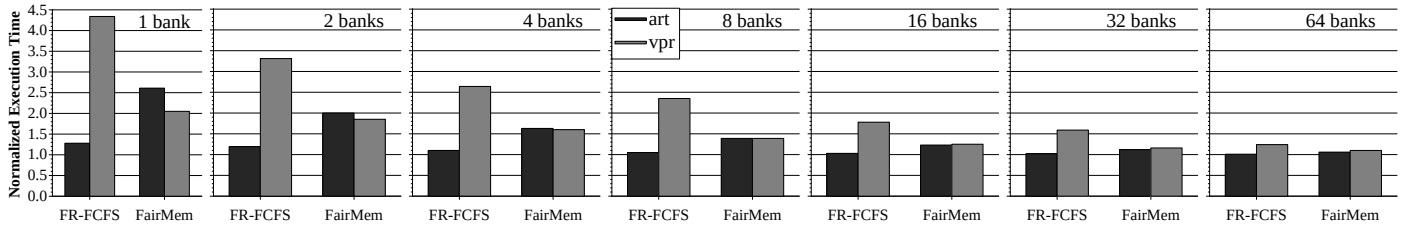

Figure 11: Slowdown of *art* and *vpr* when run together on processors with various number of DRAM banks. Execution time is independently normalized to each machine with different number of banks.

	1 bank	2 banks	4 banks	8 banks	16 banks	32 banks	64 banks
art-vpr base throughput (IPTC)	122	210	304	401	507	617	707
art-vpr FairMem throughput (IPTC)	190	287	402	513	606	690	751
FairMem throughput improvement	1.56X	1.37X	1.32X	1.28X	1.20X	1.12X	1.06X
FairMem fairness improvement	2.67X	2.57X	2.35X	2.23X	1.70X	1.50X	1.18X

Table 5: Statistics for *art-vpr* with different number of DRAM banks (IPTC: Instructions/1000-cycles)

6.2.4 Effect of Memory Latency

Clearly, memory latency also has an impact on the vulnerability in the DRAM system. Figure 12 shows how different DRAM latencies influence the mutual performance impact of *art* and *vpr*. We vary the round-trip latency of a request that hits in the row-buffer from 50 to 1000 processor clock cycles, and scale closed/conflict latencies proportionally. As memory latency increases, the impact of *art* on *vpr* also increases. *Vpr*'s slowdown is 1.89X with a 50-cycle latency versus 2.57X with a 1000-cycle latency. Again, FairMem reduces *art*'s impact on *vpr* for all examined memory latencies while also improving overall system throughput. As main DRAM latencies are expected to increase in modern processors (in terms of processor clock cycles) [35], scheduling algorithms that mitigate the impact of MPHs will become more important and effective in the future.

6.2.5 Effect of Number of Cores

Finally, this section analyzes FairMem within the context of 4-core and 8-core systems. Our results show that FairMem effectively mitigates the impact of MPHs while improving overall system throughput in both 4-core and 8-core systems running different application mixes with varying memory-intensiveness.

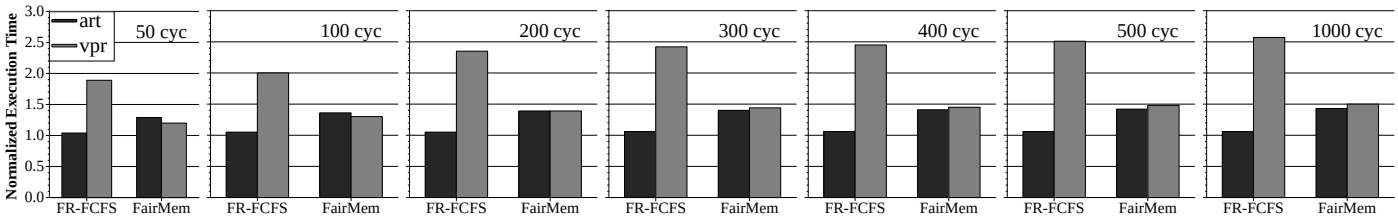

Figure 12: Slowdown of *art* and *vpr* when run together on processors with different DRAM access latencies. Execution time is independently normalized to each machine with different number of banks. Row-buffer hit latency is denoted.

	50 cycles	100 cycles	200 cycles	300 cycles	400 cycles	500 cycles	1000 cycles
art-vpr base throughput (IPTC)	1229	728	401	278	212	172	88
art-vpr FairMem throughput (IPTC)	1459	905	513	359	276	224	114
FairMem throughput improvement	1.19X	1.24X	1.28X	1.29X	1.30X	1.30X	1.30X
FairMem fairness improvement	1.69X	1.82X	2.23X	2.21X	2.25X	2.23X	2.22X

Table 6: Statistics for *art-vpr* with different DRAM latencies (IPTC: Instructions/1000-cycles)

Figure 13 shows the effect of FairMem on three different application mixes run on a 4-core system. In all three mixes *stream* and *small-stream* act as severe MPHs when run on the baseline FR-FCFS system, slowing down other applications by up to 10.4X (and at least 3.5X) while themselves being slowed down by no more than 1.10X. FairMem reduces the maximum slowdown caused by these two hogs to at most 2.98X while also improving the overall throughput of the system (shown in Table 7).

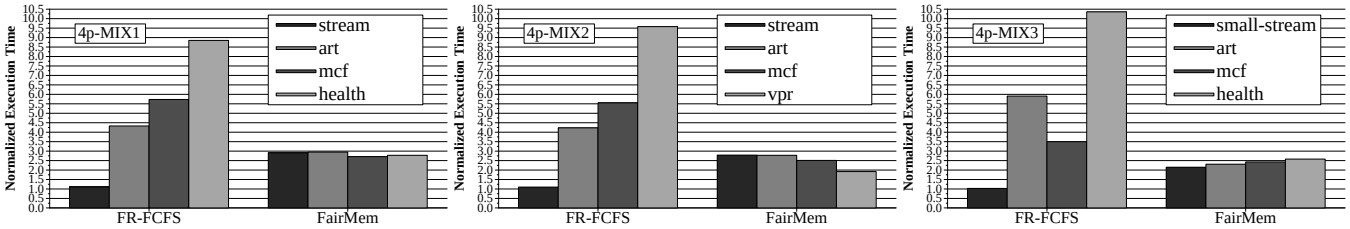

Figure 13: Effect of FR-FCFS and FairMem scheduling on different application mixes in a 4-core system

Figure 14 shows the effect of FairMem on three different application mixes run on an 8-core system. Again, in the baseline system, *stream* and *small-stream* act as MPHs, sometimes degrading the performance of another application by as much as 17.6X. FairMem effectively contains the negative performance impact caused by the MPHs for all three application mixes. Furthermore, it is important to observe that FairMem is also effective at isolating non-memory-intensive applications (such as *crafty* in MIX2 and MIX3) from the performance degradation caused by the MPHs. Even though crafty rarely generates a memory request (0.35 times per 1000 instructions), it is slowed down by 7.85X by the baseline system when run within MIX2! With FairMem *crafty*'s rare memory requests are not unfairly delayed due to a memory performance hog — and its slowdown is reduced to only 2.28X. The same effect is also observed for *crafty* in MIX3. We conclude that FairMem provides fairness in the memory system, which improves the performance of both memory-intensive and non-memory-intensive applications that are unfairly delayed by an MPH.[12]

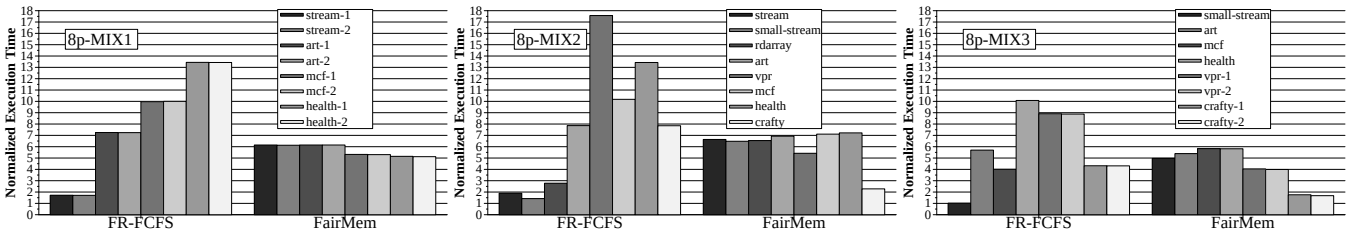

Figure 14: Effect of FR-FCFS and FairMem scheduling on different application mixes in an 8-core system

[12]Notice that 8p-MIX2 and 8p-MIX3 are much less memory intensive than 8p-MIX1. Due to this, their baseline overall throughput is significantly higher than 8p-MIX1 as shown in Table 7.

	4p-MIX1	4p-MIX2	4p-MIX3	8p-MIX1	8p-MIX2	8p-MIX3
base throughput (IPTC)	107	156	163	131	625	1793
FairMem throughput (IPTC)	179	338	234	189	1233	2809
base unfairness (Ψ)	8.05	8.71	10.98	7.89	13.56	10.11
FairMem unfairness (Ψ)	1.09	1.32	1.21	1.18	1.34	1.32
FairMem throughput improvement	1.67X	2.17X	1.44X	1.44X	1.97X	1.57X
FairMem fairness improvement	7.39X	6.60X	9.07X	6.69X	10.11X	7.66X

Table 7: Throughput and fairness statistics for 4-core and 8-core systems

7 Related Work

The possibility of exploiting vulnerabilities in the *software system* to deny memory allocation to other applications has been considered in a number of works. For example, [34] describes an attack in which one process continuously allocates virtual memory and causes other processes on the same machine to run out of memory space because swap space on disk is exhausted. The "memory performance attack" we present in this paper is conceptually very different from such "memory allocation attacks" because (1) it exploits vulnerabilities in the *hardware system*, (2) it is not amenable to software solutions — the hardware algorithms must be modified to mitigate the impact of attacks, and (3) it can be caused even unintentionally by well-written, non-malicious but memory-intensive applications.

There are only few research papers that consider *hardware* security issues in computer architecture. The one closest to our paper is a work by Grunwald and Ghiasi [11] who investigate the possibility of microarchitectural denial of service attacks. In particular, they show that SMT (simultaneous multithreading) processors exhibit a number of vulnerabilities that could be exploited by malicious threads. More specifically, they study a number of DoS attacks that affect caching behavior, including one that uses self-modifying code to cause the trace cache to be flushed. The authors then propose counter-measures that ensure fair pipeline utilization. The work of Hasan et al. [12] studies the possibility of so-called *heat stroke* attacks that repeatedly access a shared resource to create a hot spot at the resource, thus slowing down the SMT pipeline. The authors propose a solution that selectively slows down malicious threads. These two papers present involved ways of "hacking" existing systems using sophisticated techniques such as self-modifying code or identifying on-chip hardware resources that can heat up. In contrast, our paper describes a more prevalent problem: a trivial type of attack that could be easily developed by anyone who writes a program. In fact, even existing simple applications may behave like memory performance hogs and future multi-core systems are bound to become even more vulnerable to MPHs. In addition, neither of the above works consider vulnerabilities in shared DRAM memory in multi-core architectures.

The FR-FCFS scheduling algorithm implemented in many current single-core and multi-core systems was studied in [29, 28, 14, 22], and its best implementation—the one we presented in Section 2—is due to Rixner et al [29]. This algorithm was initially developed for single-thread general purpose applications and shows good throughput performance in such scenarios. As shown in [22], however, this algorithm can have negative effects on fairness in chip-multiprocessor systems. The performance impact of different memory scheduling techniques in SMT processors and multiprocessors has been considered in [38, 21].

Fairness issues in managing access to shared resources have been studied in a variety of contexts. *Network fair queuing* has been studied in order to offer guaranteed service to simultaneous flows over a shared network link, e.g., [23, 36, 3], and techniques from network fair queuing have since been applied in numerous fields, e.g., CPU scheduling [6]. The best currently known algorithm for network fair scheduling that also effectively solves the idleness problem was proposed in [2]. In [22], Nesbit et al. propose a fair memory scheduler that uses the definition of fairness in network queuing and is based on techniques from [3, 36]. As we pointed out in Section 4, directly mapping the definitions and techniques from network fair queuing to DRAM memory scheduling is problematic. Also, the scheduling algorithm in [22] can significantly suffer from the idleness problem. Fairness in *disk scheduling* has been studied in [4, 25]. The techniques used to achieve fairness in disk access are highly influenced by the physical association of data on the disk (cylinders, tracks, sectors...) and can therefore not directly be applied in DRAM scheduling.

8 Conclusion

The advent of multi-core architectures has spurred a lot of excitement in recent years. It is widely regarded as the most promising direction towards increasing computer performance in the current era of power-consumption-limited processor design. In this paper, we show that this development—besides posing numerous challenges in fields like computer architecture, software engineering, or operating systems—bears important security risks.

In particular, we have shown that due to unfairness in the memory system of multi-core architectures, some applications can act as *memory performance hogs* and destroy the memory-related performance of other applications that run on different processors in the chip; without even being significantly slowed down themselves. In order to contain the potential of such attacks, we have proposed a memory request scheduling algorithm whose design is based on our novel definition of DRAM fairness. As the number of processors integrated on a single chip increases, and as multi-chip architectures become ubiquitous, the danger of memory performance hogs is bound to aggravate in the future and more sophisticated solutions may be required. We hope that this paper helps in raising awareness of the security issues involved in the rapid shift towards ever-larger multi-core architectures.

References

[1] Advanced Micro Devices. AMD Opteron. `http://www.amd.com/us-en/Processors/ProductInformation/`.
[2] J. H. Anderson, A. Block, and A. Srinivasan. Quick-release fair scheduling. In *IEEE Real-Time Systems Symposium (RTSS)*, 2003.
[3] J. C. Bennett and H. Zhang. Hierarchical packet fair queueing algorithms. In *In Proceedings of SIGCOMM'96*, 1996.
[4] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz. Disk scheduling with quality of service guarantees. In *Proceedings of IEEE Conference on Multimedia Computing and Systems*, 1999.
[5] A. Chander, J. C. Mitchell, and I. Shin. Mobile code security by Java bytecode instrumentation. In *DARPA Information Survivability Conference & Exposition (DISCEX II)*, 2001.
[6] A. Chandra, M. Adler, P. Goyal, and P. Shenoy. Surplus fair scheduling: A proportional-share CPU scheduling algorithm for symmetric multiprocessors. In *OSDI-4*, 2000.
[7] R. S. Cox, J. G. Hansen, S. D. Gribble, and H. M. Levy. A safety-oriented platform for web applications. In *IEEE Symposium on Security and Privacy*, 2006.
[8] V. Cuppu, B. Jacob, B. Davis, and T. Mudge. A performance comparison of contemporary DRAM architectures. In *ISCA-26*, 1999.
[9] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003.
[10] S. Gochman, R. Ronen, I. Anati, A. Berkovits, T. Kurts, A. Naveh, A. Saeed, Z. Sperber, and R. C. Valentine. The Intel Pentium M processor: Microarchitecture and performance. *Intel Technology Journal*, 7(2), May 2003.
[11] D. Grunwald and S. Ghiasi. Microarchitectural denial of service: Insuring microarchitectural fairness. In *MICRO-35*, 2002.
[12] J. Hasan et al. Heat stroke: power-density-based denial of service in SMT. In *HPCA-11*, 2005.
[13] L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni. Communist, utilitarian, and capitalist cache policies on CMPs: Caches as a shared resource. In *PACT-15*, 2006.
[14] I. Hur and C. Lin. Adaptive history-based memory schedulers. In *MICRO-37*, 2004.
[15] Intel Corporation. Intel Develops Tera-Scale Research Chips. `http://www.intel.com/pressroom/archive/releases/20060926corp_b.htm`.
[16] Intel Corporation. Pentium D. `http://www.intel.com/products/processor_number/chart/pentium_d.htm`.
[17] Intel Corporation. Terascale computing. `http://www.intel.com/research/platform/terascale/index.htm`.
[18] C. K. Luk et al. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.
[19] J. D. McCalpin. STREAM: Sustainable memory bandwidth in high performance computers. `http://www.cs.virginia.edu/stream/`.
[20] O. Mutlu and E. Sprangle. Method and apparatus to control memory accesses. U.S. Patent Number 6,799,257, 2004.
[21] C. Natarajan, B. Christenson, and F. Briggs. A study of performance impact of memory controller features in multi-processor server environment. In *Workshop on Memory Performance Issues (WMPI'04)*, 2004.
[22] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair queuing memory systems. In *MICRO-39*, 2006.
[23] A. K. Parekh. *A Generalized Processor Sharing Approach to Flow Control in Integrated Service Networks*. PhD thesis, MIT, 1992.
[24] D. Peterson, M. Bishop, and R. Pandey. A flexible containment mechanism for executing untrusted code. In *Proceedings of the 11th USENIX Security Symposium*, 2002.
[25] T. Pradhan and J. Haritsa. Efficient fair disk schedulers. In *3rd Conference on Advanced Computing (ADCOMP)*, 1995.
[26] V. Prevelakis and D. Spinellis. Sandboxing applications. In *USENIX 2001 Technical Conference: FreeNIX Track*, 2001.
[27] N. Rafique et al. Architectural support for operating system-driven CMP cache management. In *PACT-15*, 2006.
[28] S. Rixner. Memory controller optimizations for web servers. In *MICRO-37*, pages 355–366, 2004.
[29] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *ISCA-27*, 2000.
[30] A. Rogers, M. C. Carlisle, J. Reppy, and L. Hendren. Supporting dynamic data structures on distributed memory machines. *ACM Transactions on Programming Languages and Systems*, 17(2):233–263, Mar. 1995.
[31] T. Sherwood et al. Automatically characterizing large scale program behavior. In *ASPLOS-X*, 2002.
[32] Standard Performance Evaluation Corporation. *SPEC CPU2000 V1.3*. `http://www.spec.org/cpu2000/`.
[33] D. Wang et al. DRAMsim: A memory system simulator. *Computer Architecture News*, 33(4):100–107, 2005.
[34] Y.-M. Wang, Y. Huang, K.-P. Vo, P.-Y. Chung, and C. Kintala. Checkpointing and its applications. In *FTCS-25*, 1995.
[35] W. Wulf and S. McKee. Hitting the memory wall: Implications of the obvious. *ACM Computer Architecture News*, 23(1), 1995.
[36] H. Zhang. Service disciplines for guaranteed performance service in packet-switching networks. In *Proceedings of the IEEE*, 1995.
[37] Z. Zhang, Z. Zhu, and X. Zhang. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *MICRO-33*, 2000.
[38] Z. Zhu and Z. Zhang. A performance comparison of DRAM memory system optimizations for SMT processors. In *HPCA-11*, 2005.