Analyzing Performance Vulnerability due to Resource Denial-of-Service Attack on Chip Multiprocessors

Dong Hyuk Woo Hsien-Hsin S. Lee

School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332
{dhwoo, leehs}@ece.gatech.edu

ABSTRACT

Due to the ever-increasing design complexity and physical constraint in frequency scaling, chip multiprocessors are considered the de facto architecture baseline for future processor generation. Through resource sharing, applications running on a CMP can achieve better resource utilization and faster inter-core communication, leading to a higher overall throughput for the entire system. From a different perspective, however, such architectures are also more susceptible to Denial-of-Service (DoS) attacks on these shared resources, increasing the vulnerability in performance. Furthermore, as the number of cores increases, attacks similar to Distributed Denial-of-Service (DDoS) attacks on the Internet can be employed to throttle these on-chip resources with the presence of multiple malicious applications. In this paper, we design several types of Denial-of-Service attacks and analyze their impact to the performance of CMPs. We also suggest a few potential countermeasure techniques to address these vulnerability for legitimate applications.

1. INTRODUCTION

Continuously increasing design complexity and verification costs are driving the microprocessor industry to abandon the conventional frequency design strategy and make a paradigm shift toward the design of chip multiprocessors (CMP) by placing multiple cores on a processor die. In the meantime, the design of each processor core for these systems is also becoming simpler due to many facts such as limited instruction-level parallelism in applications, elongated on-die wire communication, and the growing concern of power consumption and its ensuing thermal effect. While such design trend greatly alleviates the design constraints and continues to expand Moore's law, they are, however, less likely to achieve desirable performance as a result of simplified hardware. Thus, to realize the full performance potential provided by these systems, one must exploit the parallelism at the software level as much as possible with or without architectural support.

The concept of parallel processing is not new, however. Industries have been developing parallel systems from customized massively parallel machines for special purpose computing to solve specific domain problems to smaller, more affordable systems based on symmetric multiprocessors (SMP) for general-purpose applications. The programming models of these systems are often based on MIMD-style. One major difference between the recent MIMD CMP design and a traditional MIMD SMP system lies in the degree of resource sharing among cores or processors. For example, processors in a shared-bus SMP system are generally designed to share the frontside bus (FSB), whereas cores in a CMP, for they are more tightly coupled, can be designed to share the backside bus (BSB), and their corresponding lower-level memory structures.[1] Sharing resources such as the L2 cache provides the advantage of reduced inter-core communication overhead. On the other hand, compared to private L2 cache design, which splits each resource evenly for each core, sharing resources provides more flexibility to satisfy dynamic workload behavior, thus increasing the overall utilization efficiency [17].

Most of the existing or proposed CMP designs share the last level caches, e.g. the L2 caches. By sharing instructions and data in the L2 cache, one processor core can read the data value updated by other cores by simply communicating through the L2 cache with a shared backside bus that supports a snoop-based coherence protocol. As a result, all other architectural components (e.g. the frontside bus and DRAM) below the L2 cache are all shared. These shared resources, however, are never provided for free and must be used with caution for performance assurance. For example, on-chip bus bandwidth can be increased by multiplying the bus width, yet it increases routing complexity at design time, the number of metal layers at manufacturing time, and eventually the power consumption of overall chip at run time. One can also increase the bandwidth by increasing the bus frequency. Unfortunately, it can worsen signal integrity, power consumption, etc. On the other hand, the L2 cache capacity becomes even more restrained when shared by several cores. Increasing the L2 cache size, while reducing the capacity and conflict misses, can degrade the overall system performance for a longer access latency. Given a fixed die area budget, a larger L2 implies fewer cores on die, impairing the performance per mm^2, a popular metric for comparing the effectiveness of CMPs. Another common barrier of enlarging an L2 cache is the increased power, in particular, the leakage power.

Similar to the on-chip bus, the off-chip bus bandwidth can be improved either by enlarging the bus width or by increasing the bus frequency. Nevertheless, one major drawback of increasing the off-chip bus width is the consequence of more I/O pads that do not scale with Moore's law. Increasing the off-chip bus frequency is also likely to be limited by physical design constraint, which is not considered as a scalable solution. Other design alternatives such as optical or RF/wireless inter-chip interconnect [1, 19, 3, 5] were recently investigated for overcoming this constraint.

Although different ways of sharing these resources can be sensitive to system performance, there are not too many studies that address issues with respect to how to share these resources more resiliently. Most of the prior studies focused on the impact of L2 cache space partitioning on overall system performance [11, 12, 18,

[1] *Frontside bus* denotes the bus between the L2 cache and the main memory while *Backside bus* represents the bus between L1 and L2 caches.

9, 15, 14]. When this effect is intertwined with job scheduling from the standpoint of an operating system, the problem will become much more complicated [6]. This problem is difficult to solve, not only because it is involved with so many parameters, but also it is not even clear which performance goal we are tuning for [12, 9]. This problem is even more critical in the server farms where a service provider is hosting several services for various customers who pay different premiums.

In this paper, we investigate this resource sharing problem from a different perspective. We evaluate how a cracker can exploit the shared resources of a CMP by injecting malicious threads to deprive resources of legitimate applications and cause performance degradation. In essence, we study the performance vulnerability due to a variety of Denial-of-Service (DoS) attacks on CMPs. The main contributions of this research include the following.

- Mimicking the mindset of crackers, we design malicious programs that voraciously exhaust shared resources including both *time* (link bandwidth) and *space* (memory space) and examine the performance degradation of other normal processes.
- We show that well-known hardware properties such as the LRU algorithm and cache inclusion property can be manipulated by crackers in succeeding in their malicious intention.
- We present that several malicious threads can cooperate with each other to perform a microarchitectural Distributed Denial-of-Service (DDoS) attack.
- We propose and discuss several preliminary solutions to penalize the malicious code based on monitoring results on resource utilization.

2. RELATED WORK

Iyer [11] proposed a new cache management framework to provide prioritized services on a CMP. He mainly focused on the design, implementation and performance evaluation of Quality-of-Service (QoS) priority enforcement mechanisms including cache set partitioning, selective cache allocation, and heterogeneous cache regions. This work, however, lacked an in-depth investigation of priority classification and assignment mechanism.

Kim *et al.* [12] focused on a cache partitioning algorithm to provide fairness to the processes running on a CMP. They proposed several metrics and studied relationship between fairness and throughput. These metrics included absolute and relative numbers of cache misses and cache miss rates.

Fedorova *et al.* [6] proposed an operating system scheduling algorithm to minimize the cache contention among threads. They monitored the memory re-use pattern for each thread and estimated their cache miss rates to identify and schedule symbiotic threads together for higher throughput.

Yeh and Reinman [18] proposed PDAS, a physically distributed NUCA L2 cache design with an adaptive sharing mechanism, to provide QoS on a CMP. They used miss rate and IPC as a metric to determine the cache partitioning. Hsu et al. [9] compared different policies on performance targets. They used several metrics to evaluate these policies, including miss rates, misses per cycle, and IPC. Unfortunately, they found that overall performance was seriously dependent on the used metric, and it was very difficult to pick one metric over the other. Rafique et al. [15] proposed an OS-level management of shared caches in CMPs. They forced the cache to choose the victim line based on OS-specified quotas. They, however, did not specify the metric to set the quota for each processes. More recently, Qureshi and Patt [14] proposed utility-based cache partitioning. They used sum of weighted IPCs, sum of IPCs, and harmonic mean of normalized IPCs as a metric to determine cache size allocated to each process.

Note that all above papers focused only on shared cache space, i.e. the capacity issue. They failed to address the unfair sharing problem in interconnection bandwidth. Considering that the bandwidth of on-chip or off-chip interconnection becomes more scarce for each core as the number of cores goes up in CMPs, unfair utilization of the bandwidth will eventually become a roadblock to scaling up performance. This paper focuses on both the cache capacity issue as well as the often ignored interconnection bandwidth issue.

Furthermore, most of the crackers are very determined and smart enough to exploit the vulnerability of partitioning algorithms proposed by prior works. For example, the IPC can be easily fooled by inserting dummy instructions, e.g. or r0 = r0, r0. The miss rates can also be easily manipulated by concocting the working set deliberately. Thus, a partitioning algorithm based on these metrics may result in a guarantee of a large space to the malicious thread and make these attacks more effective.

In this paper, we are interested neither in fairness nor in QoS on CMPs. In other words, we do not argue that the processor should always provide its best services to meet all the requirements for applications. Rather, we argue that we should have, at least, a hardware mechanism that can detect and prevent the performance of our system from suffering from intentional DoS attacks by well premeditated malicious applications.

3. DENIAL-OF-SERVICE ATTACKS ON A CMP

3.1 Generic DoS Attacks

Denial-of-Service is a common network attack method in which a malicious user intentionally makes a flood of requests to a targeted Internet service, rendering the victim server unavailable to legitimate subscribers. With this type of attack, lots of packets were sent to saturate one link, exhaust the memory or CPU time of the server, overflow the buffer of the network interface card of the victim server. Although link level fairness is strictly controlled by Medium Access Control (MAC) layer protocol, packets from one source can make packets from other sources unreachable to the server, due to the vulnerability of the upper level transmission control such as TCP SYN or UDP flood attack [4].

A naïve DoS attack is relatively easier to identify than a Distributed DoS attack. Several mechanisms were proposed to detect and battle against DoS attacks including complex algorithms based on pattern matching through packet monitoring. A DDoS attack, however, is very difficult to detect as in this case multiple hosts (sometimes at different locations) are compromised to exhaust the resources of the victim server by the same group of crackers. Each connection established by the compromised machines behaves exactly like a normal user making a legitimate request although aggregated requests from these compromised machines can overwhelm the capacity the victim server can sustain.

The conventional DoS attacks on the Internet exploit and devour shared yet limited resources in an illegal, unexpected manner. The network bandwidth, server CPU time, or server memory and buffer space offered by the service providers are abused and completely consumed by the crackers. This observation motivates our study on a CMP which ensembles a scaled down replica with shared backside bus and frontside bus bandwidth, cache space, and CPU time. When a malicious code is launched on one or multiple cores of a CMP, the CMP will suffer from a similar availability issue. Note that such an attack by nature is very different from fairness issue during resource allocation. The problem is much more complicated as the fairness must be guaranteed only for legitimate applications.

3.2 Microarchitectural DoS Attack

To the best of our knowledge, there are two related works regarding vulnerability of microarchitectural resources: the first one exploits the shared resources in a simultaneous multithreading processor (SMT) [7]; the second one discusses heat dissipation and its implication on performance in a processor when a malicious code continuously exercises the same portion of a processor to trigger a thermal alarm and its protection mechanism such as throttling the execution or scaling down the frequency [8]. Since an SMT is a more tightly-coupled shared architecture than a CMP, a thread on an SMT can harm the performance of other threads more seriously, by occupying execution units aggressively, flushing the pipeline, flushing the trace cache, etc [7]. However, lower-level shared resources such as bus bandwidth and memory space has been ignored in previous works. In this paper, we will show that attacks against these resources on a CMP (and/or an SMT) can also be serious.

Furthermore, note that a DoS attack on a CMP is clearly different from that on a shared-bus SMP system as follows:

- Higher level memory structure and interconnect are shared on a CMP while they are isolated and dedicated to each processor on an SMP. Consequently, a process running on a CMP needs to access the shared resources more frequently. For example, a process or thread running on a CMP will access the shared backside bus upon every L1 cache miss. The same process running on an SMP, however, only accesses the shared frontside bus upon every L2 cache miss, a much rarer event than a L1 cache miss. Thus, processes running on a CMP are more susceptible to DoS attacks.
- The main memory (a shared resource on an SMP) is a larger structure than the last level cache (a shared resource on a CMP). Thus, it is difficult for malicious threads on an SMP to occupy a large amount of main memory space fast enough to perform DoS attacks.
- Cores on a CMP share the last level cache, while processors on an SMP share the main memory. The key difference is that the allocation to and eviction from the last level cache of each cache line is determined by the hardware, e.g. the LRU algorithm, yet the allocation to and eviction from the main memory of each memory page is fully handled by the operating system, which is not precitable from the standpoint of malicious threads. Thus, SMPs are less vulnerable to DoS attacks.

4. TYPES OF ATTACKS

In this section, we describe various malicious programs designed by us to exploit the performance vulnerability. In our explanation, we first define the processor architecture used in the experiments and then show several attack mechanisms — attacks against the backside bus bandwidth, the L2 cache, and the frontside bus bandwidth. Furthermore, we present a few more elaborate methods — LRU and inclusion property aware attacks and attacks that exploit locking protocol.

4.1 CMP Model

Our CMP model, a quad-core system, is illustrated in Figure 1. The L1 instruction and data caches are private, 32KB each. A 2MB L2 cache is shared by all processor cores. Similar to the Intel® Core™ microarchitecture, the backside bus is set to 256 bits running at full processor speed. The detailed specification is shown in Table 1.

Each core accesses the shared backside bus upon every L1 cache miss and we assume that bus arbitration is controlled by an arbiter that is capable of providing bus access fairness to all cores. Each core has its own Miss Status Handler Register (MSHR) for handling L1 misses as in most conventional processors to provide non-blocking cache functionality. The shared L2 cache contains its own MSHR shared by all cores. The frontside bus is modeled similar to the frontside bus of the IA-32 architecture.

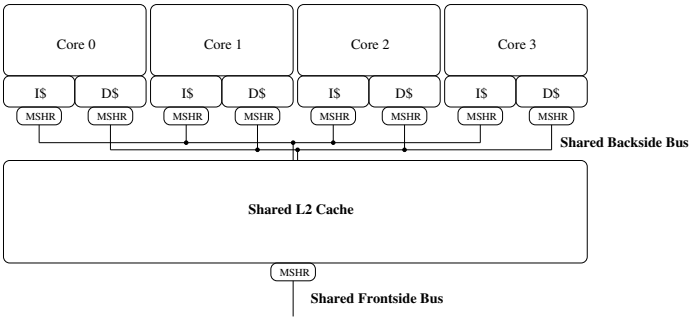


Figure 1: Experimental CMP Model

Clock frequency	2.0 GHz
Number of cores	4
Instruction issue width	3
L1 I$ (per core)	2-way set associative 32KB cache with 64B line 1 cycle hit latency
L1 D$ (per core)	2-way set associative 32KB cache with 64B line 1 cycle hit latency 8-entry MSHR
Data Bus bandwidth between L1 D$ and L2$ (shared)	64 GBps (2GHz * 256bit)
L2$ (shared)	8-way set associative 2MB cache with 64B line 14 cycle hit latency 1 shared MSHR
Bus bandwidth between L2$ and DRAM (shared)	16 GBps
DRAM latency (shared)	100 ns

Table 1: Processor configuration

4.2 Attack against BSB Bandwidth

The first malicious code is designed to thrash the bandwidth of the backside bus. To saturate this bus, we need to generate L1 data cache misses as frequently as possible, because this bus is used as the communication channel between the L1 and the L2. To generate L1 misses as many and as frequently as possible, our malicious code should be very efficient to be aggressive enough. In other words, the MSHR of a core running the malicious code must be always fully occupied. To meet this requirement, the malicious code is written with assembly language and well optimized by avoiding L1 instruction cache miss, removing the side effect of branch misprediction as much as possible, removing the instruction dependency while generating an L1 data cache miss almost every cycle, and minimizing response time from the lower level memory such as page faults. Note that the last requirement is one of the most important criteria. Without it, we will be unable to completely saturate the backside bus even with a filled MSHR.

The pseudo code of this attack is listed in Figure 2. The code constantly loads data from a 64KB array with a stride size of 64B, which is equivalent to the L1 line size. Note that the entire data memory footprint traversed by this code is 64KB, twice the L1 data cache size. Ideally, the access pattern of this code will incur an L1 data cache miss for each load instruction regardless of the set associativity of the L1 data cache.

As we will show later in the discussion of the simulation results, some interesting behavior of this code is observed. Even though

```
    allocate 64KB array
    mov $2, (pointer to this array)
    mov $3, (pointer to this array + 32K)
L1:
    lw $22, 0($2)
    lw $22, 64($2)
    lw $22, 128($2)
    ...
    lw $22, 32704($2)
    lw $22, 0($3)
    lw $22, 64($3)
    lw $22, 128($3)
    ...
    lw $22, 32704($3)
    jmp L1
```

Figure 2: Pseudo code for the attack against the backside bus

we hand-crafted this code to generate an L1 miss for each load instruction, this code does not generate cache misses as frequent as we have expected. The reason is that the backside bus is often saturated, leaving the processor incapable of posting incessant memory requests on the bus. As such, the code not only degrades the performance of itself, the bus thrashing also impairs the ability of other CMP cores sharing the same backside bus from gaining bus access, resulting in poor performance for all workloads.

4.3 Attack against the L2 Cache

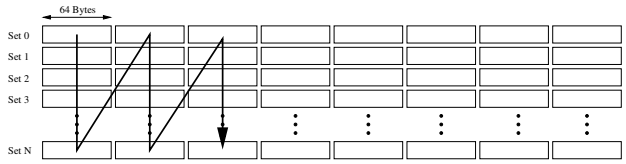
The second malicious code is designed to sweep through the L2 cache space as quickly as possible. Using this attack, the malicious thread generates a large number of L1 misses that wipe out the footprint of the victim process. Consequently, the victim process suffers from a higher L2 miss rate. To make this attack effective, the malicious code should generate L1 data cache misses fast enough so that it obsoletes the the L2 cache lines loaded by the victim process more quickly. To meet these requirements, we designed the malicious code similar to the one described in Section 4.2. The only difference is that a larger memory footprint is used — 2MB, the size of the L2 cache. Because this code inherently saturates the backside bus due to frequent L1 cache misses, the code is expected to degrade the performance of the victim process more seriously. Note that this attack might not behave exactly as what we have expected for most processors employ physically-indexed L2 cache. Nevertheless, we do not expect it alleviates the vulnerability as in typical situation the operating system does not allocate pages in random order.

4.4 Attack against the FSB Bandwidth

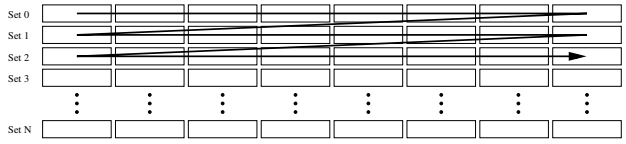
Similar to the technique (or black art) in Section 4.2 that thrashes the backside bus, to saturate the frontside bus, a malicious code needs to generate L2 cache misses as frequently as possible. The only difference from previous code is the memory footprint required, which is now 4MB, twice of the L2 cache size. Note that this attack also sweeps through the entire L2 cache space, and could saturate the backside bus bandwidth when contrived properly.

4.5 LRU and Inclusion Property Aware Attack

The fourth type of attacks is a variation of the second attack described in Section 4.3. Instead of sweeping the cache with a stride of 64B as shown in Figure 3(a), this attack successively sweeps each cache set as shown in Figure 3(b). Note that, with the L2 cache configuration shown in Table 1, all memory accesses to the address $x + n * 2^{18}$ are mapped to the same set. For example, accesses to the address 0, $1*2^{18}$, $2*2^{18}$, $3*2^{18}$, $4*2^{18}$, $5*2^{18}$, $6*2^{18}$, and $7*2^{18}$, are all mapped to set 0. The pseudo code for this attack for our 8-way L2 cache, shown in Figure 4, successively loads data from these addresses by using eight registers ($18 to $25), so that



(a) Normal stride attack



(b) LRU and inclusion property aware attack

Figure 3: LRU and inclusion property aware attack

```
    allocate 2MB array
    mov $7, (pointer to this array)
    mov $10, (0)
    add $11 = $10, 2^18
    add $12 = $11, 2^18
    ...
    add $17 = $16, 2^18
    add $18 = $10, $7
    add $19 = $11, $7
    ...
    add $25 = $17, $7
L1:
    lw $26, 0($18)
    lw $26, 0($19)
    ...
    lw $26, 0($25)
    lw $26, 64($18)
    lw $26, 64($19)
    ...
    lw $26, 64($25)
    ...
    lw $26, 4032($18)
    lw $26, 4032($19)
    ...
    lw $26, 4032($25)
    offset increase or reset
    jmp L1
```

Figure 4: Pseudo code for the LRU and inclusion property aware attack

cache lines loaded by the victim process can be eventually evicted. Once it successfully evicts those lines, it attacks the next set by changing the offset.

Considering the facts that (1) the cache inclusion property should be maintained for an efficient coherence protocol implementation, and (2) the LRU policy always evicts the least recently accessed cache line — if a cracker successively accesses the same cache set with different address, he ensures that L2 cache lines of the victim core is evicted, and that the L1 cache lines of the victim core is also invalidated. Therefore, the victim core will access the backside bus and the L2 cache much more often, which degrades the performance of the application running on the victim core.

4.6 Attack Using Locked Atomic Operation

To implement the atomic operation, a Read-Modify-Write instruction is typically provided in commercial processors to implicitly lock the bus [10, 2]. If a malicious code can successfully lock the bus, other processes running in other cores must wait until the

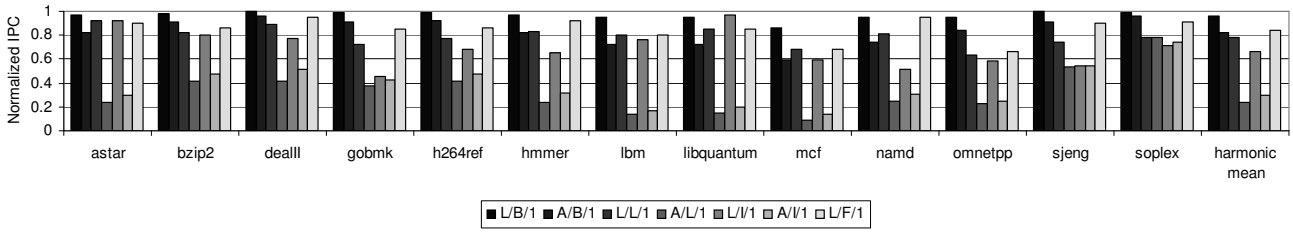


Figure 5: DoS vulnerability due to different attacks

bus lock is released. In our experiments, we assume that the hardware implementation of this operation locks the backside bus until the data is returned from the L2. Normal load operations of the first, second and third malicious codes discussed earlier are substituted with locked atomic operations to generate aggressive locked atomic operations.

4.7 Microarchitectural DDoS Attack

Similar to the attack models found in conventional Internet, we also foresee the possibility of the Distributed Denial-of-Service attacks in a future many-core processor system. These malicious codes can be scheduled simultaneously and cooperate with each other. DDoS attacks are expected to increase the latency of acquiring the backside bus, and may consume (and void) the L2 cache space more aggressively. In our experiments, we run several threads of codes discussed previously to simulate DDoS.

5. EXPERIMENTAL RESULTS

5.1 Simulation Environment

We experimented these attacks on the SESC simulator [16]. The latest SPEC CPU2006 benchmark programs are used as victim applications to be attacked by the malicious codes.[2] Our CMP model was described in Section 4.1. Our simulation results were gathered by executing 100 million instructions of each victim application after skipping the first five billion instructions. The link bandwidth and contention of each bus are modeled with an assumption of the arbitration latency to be zero. Note that this is an optimistic assumption in terms of performance impact. In reality, the performance degradation due to contention with malicious threads will be greater due to the arbitration latency.

5.2 Simulation Results

Figure 5 shows the performance results on DoS vulnerability. The baseline IPC was measured when only the victim application is running on the CMP. In this case, 3 out of 4 cores are idle. The graph shows the normalized IPC results for a victim application executed on one core together with one malicious code running on another core under different attacking scenarios. We varied the types of attacks to demonstrate the performance vulnerability of different resources. For example, **L/B/1** contains **1** malicious thread that uses normal **L**oad instructions to attack the **B**ackside bus. Table 2 details the acronyms used in the legend.

As shown in Figure 5, by running one single malicious thread, the performance of the victim application can be degraded by as much as 91%. In general, the performance sensitivity is highly dependent on the program behavior. For instance, mcf has been long known for its poor memory performance on some particular indirect load instructions. By attacking all memory-related resources

[2]Benchmark programs written in Fortran or using unsupported system calls by SESC simulator are excluded from the simulation.

[instruction]/[resource]/[# of attackers]

[instruction]
- L: attack using normal Load instruction
- A: attack using locked Atomic instruction

[resource]
- B: BSB attack
- L: BSB and L2 space attack
- F: BSB, L2 space and FSB attack

[# of attackers]
- 1: One malicious thread
- 2: Two malicious threads
- 3: Three malicious threads

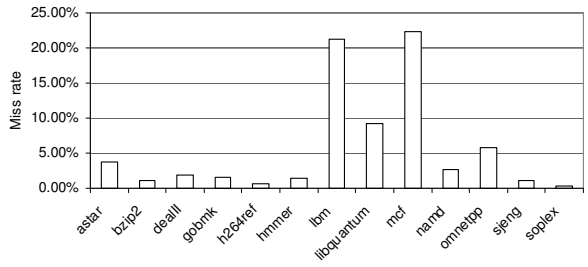
Table 2: Attack Acronyms

such as the backside bus, frontside bus and L2 space can further reduce its performance substantially as shown in Figure 5. Figure 6(a) shows the L1 miss rate of each baseline victim application to provide more insight to the vulnerability. As expected, it is clearly observed that the applications with higher L1 miss rates are usually more vulnerable to the attacks as they need to access the shared resources more often. In other words, those applications demonstrate high locality and can better endure memory latency will be less susceptible to such DoS attacks in CMPs. In addition, we also found that our LRU and inclusion property aware attacks can slightly, though not seriously, increase the L1 miss rates of the victim process in most of the cases, making the victim application more vulnerable to the attack. Future workloads containing larger working set (e.g. RMS [13] advocated by Intel) will clearly be more vulnerable to the DoS attack.

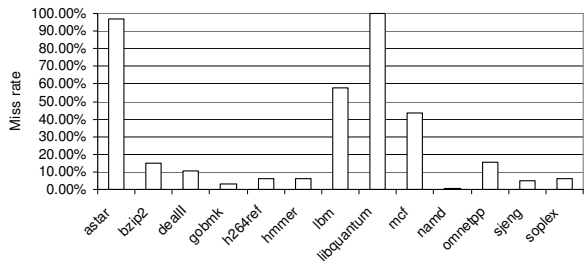
Another clear trend is that the performance of the victim process with a higher L2 miss rate (Figure 6(b)), e.g. astar and libquantum, is not affected too much by the attack against the L2 cache space. As these applications do not need the cache line again, once it is evicted from the L1 cache, their performance is less sensitive to the L2 cache space that they occupy.

Also from the figure, it is interesting to find that locked Atomic operations are very destructive when they are improperly used by the crackers. As suggested by Intel developer's manual [10], frequent use of these operations is not recommended for performance reasons. Worse yet, crackers can deliberately exploit this property to satisfy their goals.

As expected, attacking the backside bus bandwidth and the L2 space together degrades the performance more than simply attacking the backside bus bandwidth. Nevertheless, attacks that target the frontside bus bandwidth appear to be less effective in Figure 5. Our analysis shows that it is due to the lower performance of the attacking thread. Because the DRAM memory latency is fairly long, the MSHR is quickly filled by these long latency operations. As such, the attacking thread fails to thrash the backside bus bandwidth and the L2 space more timely, making these resources more available to the victim application.



(a) L1 Miss rate



(b) L2 Miss rate

Figure 6: Miss rate of the baseline victim process

Simulations with more than one attacking thread are also performed to investigate the vulnerability under a Distributed DoS attack. Figure 8(a) shows the normalized IPC numbers as a result of attacking the backside bus using one, two and three malicious threads. First of all, we found the IPC results of different attacking threads in each simulation scenario are almost same. This suggests that each core accesses the bus in a fair manner in our simulation environment. In spite of fair accesses, the victim process suffers from the longer latency caused by the saturated backside bus bandwidth. The simulation results show that all victim processes run slower as the number of attacking threads increases. This is very obvious because the probability that the victim process is allowed to access the bus is getting smaller with more malicious threads running.

Unlike the attack to the backside bus, it is not always true that a DDoS attack is more effective than a DoS attack when the attack is targeted to both the backside bus and the L2 cache space. For example, as the number of malicious threads increases, the performance degradation of sjeng decreases, but that of mcf increases, as shown in Figure 8(b) and Figure 8(c). Note that this type of attack by a single thread is already saturating the backside bus and evicting an L2 cache line loaded by the victim process. Consequently, there is no big advantage with more malicious threads. Considering the fact that these attacks are more effective only when the malicious threads evict a cache line, either from the L2 cache or from the L1 cache of the victim core, which the victim core needs to access, the effect of each attack is not highly predictable. That is why a DDoS attack cannot always be more effective than a DoS one.

A DDoS attack can work very well if the goal of the malicious threads is to saturate the frontside bus and their higher level memory resources. As stated before, this attack done only by one malicious thread suffers from an overflow in the MSHR, thus it cannot exhaust the backside bus bandwidth and L2 cache space efficiently. As the number of malicious threads increase, however, it becomes more difficult for the victim to receive enough use of these resources. That explains why a DDoS attack works well in bringing down the performance of the victim application shown in Figure 8(d).

6. SUGGESTED SOLUTIONS

6.1 Monitoring Functionality on Utilization of Shared Resources

Pure software solutions such as commercial virus scanner are not effective for microarchitectural DoS or DDoS, because malicious threads can degrade the performance of this software, too. Furthermore, it is very easy to write variations of these attacks, making the signature-based detection mechanisms less useful. Rather, we need an adaptive solution to solve this problem based on real-time monitoring. To detect malicious, atypical behavior, the hardware at least needs to monitor the utilization status of shared resources, and provides appropriate information such as bus utilization, cache sharing, etc. One major research challenge is to identify and differentiate a malicious process from other normal workloads.

6.2 Dynamic Miss Status Handler Register

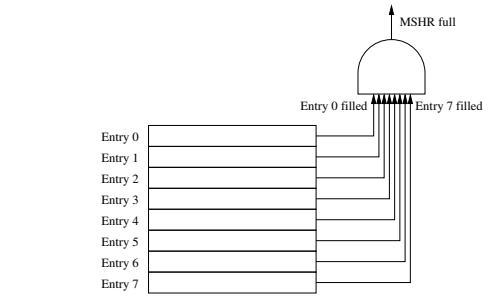
From our experiences of crafting malicious codes and observation on the simulation results, for malicious codes to succeed, the backside bus should be saturated. This is evident since the backside bus is the shared resource that every request to lower-level shared resources such as the L2 or the frontside bus must go through. A successful malicious code should generate L1 cache misses as frequently as possible to dispatch enough requests to these shared resources. Thus, to prevent DoS attacks, one possible solution is to throttle the memory activity of the malicious code.

In a uni-processor environment, this bandwidth is dedicated to only one process at a given time. Thus, the number of entries of MSHR is typically designed to fully utilize the available bandwidth. However, in a CMP where several cores (and several processes) share the same bus, architects need to develop some microarchitecture level solutions to dynamically adapt the available bandwidth for the legitimate processes.

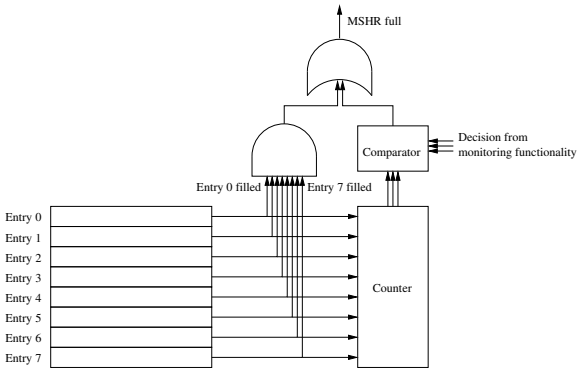
One potential solution we are investigating is to adaptively adjust the the number of outstanding memory requests based on the bandwidth utilization and/or cache space utilization, which we call Dynamic MSHR (DMSHR).[3] DMSHR, as shown in Figure 7(b), can be implemented by adding some simple logic to a regular MSHR. By monitoring the bus bandwidth utilization, the processor determines the number of MSHR entries dynamically. If a DMSHR is servicing more requests than this decision, a full signal will be set to the DMSHR to defer further requests to the processor. For example, if it is monitored that the bus has been saturated, and a suspicious thread consumes 90% of the bus bandwidth, the maximum number of DMSHR entries allowed to the core running the suspicious thread can be decreased to throttle the aggressive memory accesses.

Because a late DoS detection does not affect the functional correctness, the new MSHR full signal does not need to wait for the decision made at the current cycle. Thus, the new logic components including the counter and the comparator should not affect the latency of an MSHR full signal from the view of the processor. Compared to a conventional MSHR full signal as shown in Fig-

[3]A centralized controller such as a bus arbiter might be able to do this job too. But a centralized control will not be efficient when the number of cores increases in the future.



(a) Conventional MSHR



(b) Dynamic MSHR

Figure 7: Dynamic MSHR

ure 7(a), DMSHR introduces only one additional OR gate latency, which is not expected to affect the latency of the MSHR full signal.

The major challenge with this approach is how to accurately detect the malicious code and how to adaptively control the number of MSHR entries of the core running the malicious code. We are presently investigating methods to achieve this.

6.3 OS Level Solution

This problem can also be supervised and mitigated by the operating system. The system administrator may set some policies on the limit of resource utilization, and let the OS preemptively evict the processes that appear to be malicious. But this solution cannot be free from the false alarm problem. For example, a normal, memory-bound process can be mistaken as a malicious thread. Again, similar to what was discussed in Section 6.2, more research is needed to realize a more accurate detection mechanism.

Another OS level solution is adopting different pools based on resource usage pattern. For example, an OS can allot and dispatch processes into two different pools by tracking the dynamic use of resources by each process: one for the applications which require a small amount of resources, and another for those which require a large amount of resources. By having two separate pools, it can guarantee a faster turnaround time for the processes that are less resource bound.

7. CONCLUSION AND FUTURE WORK

Due to resource sharing among processor cores in a CMP, the performance of each process is highly dependent on the amount of resources allocated. Therefore, if these resources are exhausted by malicious threads, the performance of the overall system will be seriously degraded. The malicious threads we designed in this paper degraded the performance of a legitimate application by up to 91% on a CMP. Furthermore, several threads can be organized in a DDoS manner and degrade performance when running simultaneously.

DoS vulnerability on a different intra-chip interconnection topology, such as embedded ring, or network-on-chip in future many-core processors, would be worth researching. Due to distributed arbitration nature of ring architecture, the bandwidth of the ring would be more vulnerable to the attack. Network-on-chip, where a large number of buffers are used in cores and routers, is also more susceptible to DoS attacks. Techniques to identifying these malicious attacks and solutions for different configurations of intra-chip interconnection will be investigated in our future work.

8. REFERENCES

[1] International Technology Roadmap for Semiconductors. http://public.itrs.net.

[2] AMD. *AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions*, 2006.

[3] Alan F. Benner, Michael Ignatowski, Jeffrey A. Kash, Daniel M. Kuchta, and Mark B. Ritter. Exploitation of optical interconnects in future server architectures. *IBM Journal of Research and Development*, 49(4/5), July-September 2005.

[4] CERT Coordination Center. Denial of Service Attacks. http://www.cert.org/tech_tips/denial_of_service.html.

[5] Mau-Chung Frank Chang, Vwani P. Roychowdhury, Liyang Zhang, Hyunchol Shin, and Yongxi Qian. RF/Wireless Interconnect for Inter-and Intra-Chip Communications. *Proceedings of the IEEE*, 89(4):456–466, 2001.

[6] Alexandra Fedorova, Margo Seltzer, Christopher Small, and Daniel Nussbaum. Performance Of Multithreaded Chip Multiprocessors And Implications For Operating System Design. In *Proceedings of the USENIX 2005 Annual Technical Conference*, April 2005.

[7] Dirk Grunwald and Soraya Ghiasi. Microarchitectural denial of service: insuring microarchitectural fairness. In *Proceedings of IEEE/ACM 35th International Symposium on Microarchitecture*, pages 409–418, November 2002.

[8] Jahangir Hasan, Ankit Jalote, T. N. Vijaykumar, and Carla Brodley. Heat Stroke: Power-Density-Based Denial of Service in SMT. In *Proceedings of the Eleventh Annual Symposium on High Performance Computer Architecture*, pages 166–177, February 2005.

[9] Lisa R. Hsu, Steven K. Reinhardt, Ravishankar Iyer, and Srihari Makineni. Communist, Utilitarian, and Capitalist Cache Policies on CMPs: Caches as a Shared Resource. In *Proceedings of the 2006 International Conference on Parallel Architectures and Compilation Techniques*, September 2006.

[10] Intel. *Intel Architecture Software Developer's Manual Volume 3: System Programming Guide*, 2006.

[11] Ravi Iyer. CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms. In *Proceedings of the 2004 International Conference on Supercomputing*, pages 257–266, June 2004.

[12] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *Proceedings of the 2004 International Conference on Parallel Architectures and Compilation Techniques*, pages 111–122, September 2004.

[13] Bob Liang and Pradeep Dubey. Recognition, Mining and Syntesis. *Intel Technology Journal*, 09(02), May 2005.

[14] Moinuddin K. Qureshi and Yale N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance Runtime Mechanism to Partition Shared Caches. In *Proceedings of IEEE/ACM 39th International Symposium on Microarchitecture*, December 2006.

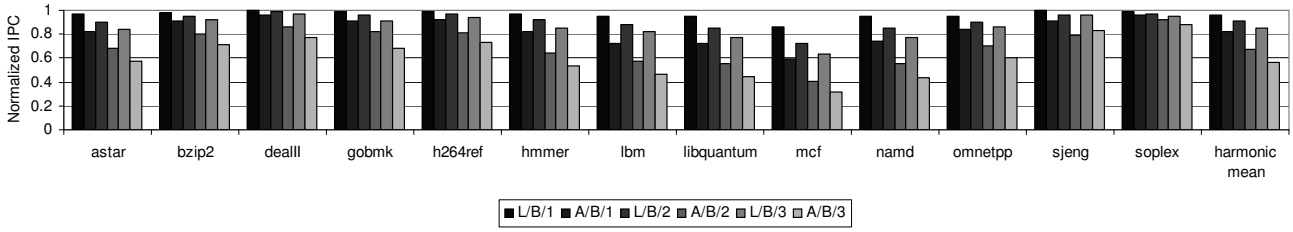
[15] Nauman Rafique, Won-Taek Lim, and Mithuna Thottethodi. Architectural Support for Operating System-Driven CMP Cache Management. In *Proceedings of the 2006 International Conference on Parallel Architectures and Compilation Techniques*, September 2006.

[16] Jose Renau, Basilio Fraguela, James Tuck, Wei Liu, Milos Prvulovic, Luis Ceze, Smruti Sarangi, Paul Sack, Karin Strauss, and Pablo Montesinos. SESC simulator, January 2005. http://sesc.sourceforge.net.

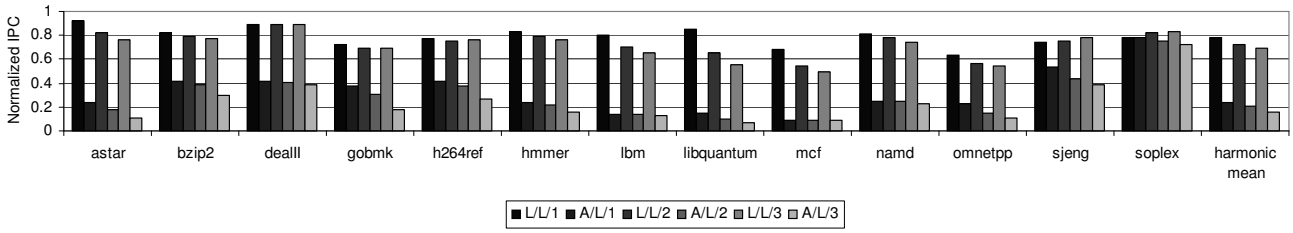
[17] Ofri Wechsler. Inside Intel® Core™ Microarchitecture: Setting New Standards for Energy-Efficient Performance. *Technology@Intel Magazine*, March 2006.

[18] Thomas Y. Yeh and Glenn Reinman. Fast and Fair: Data-stream Quality of Service. In *Proceedings of the 2005 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, September 2005.

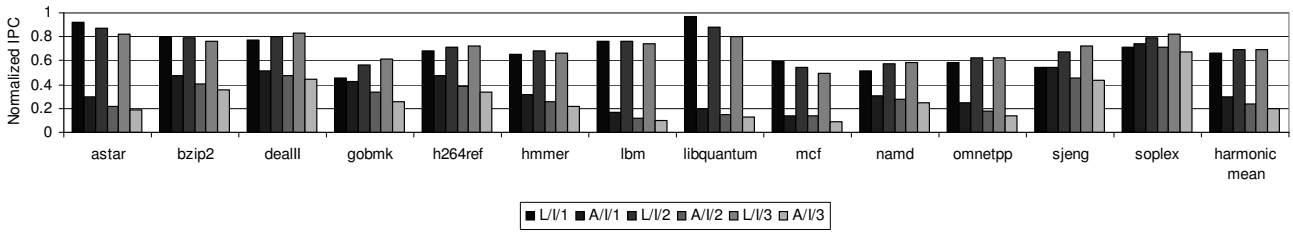
[19] Ian Young. Intel introduces chip-to-chip optical I/O interconnect prototype. *Technology@Intel Magazine*, pages 3–7, April 2004.



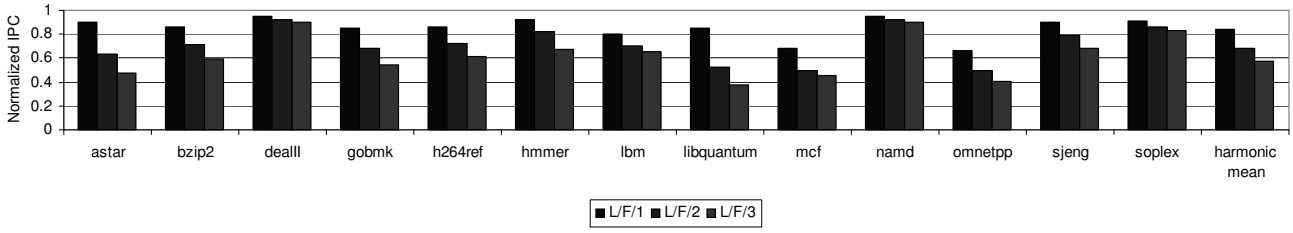
(a) Performance degradation of the victim due to attacks against the backside bus bandwidth



(b) Performance degradation of the victim due to attacks against the backside bus bandwidth and L2 space



(c) Performance degradation of the victim due to LRU and inclusion property aware attacks against the backside bus bandwidth and L2 space



(d) Performance degradation of the victim due to attacks against the backside bus bandwidth, L2 space, and the frontside bus bandwidth

Figure 8: Distributed DoS vulnerability due to different attacks