A Quantitative Study of Memory System Interference in Chip Multiprocessor Architectures

Magnus Jahre, Marius Grannaes and Lasse Natvig
Department of Computer and Information Science
Norwegian University of Science and Technology
$\{jahre|grannas|lasse\}@idi.ntnu.no$

Abstract—The potential for destructive interference between running processes is increased as Chip Multiprocessors (CMPs) share more on-chip resources. We believe that understanding the nature of memory system interference is vital to achieve good fairness/complexity/performance trade-offs in CMPs. Our goal in this work is to quantify the latency penalties due to interference in all hardware-controlled, shared units (i.e. the on-chip interconnect, shared cache and memory bus). To achieve this, we simulate a wide variety of realistic CMP architectures. In particular, we vary the number of cores, interconnect topology, shared cache size and off-chip memory bandwidth. We observe that interference in the off-chip memory bus accounts for between 63% and 87% of the total interference impact while the impact of cache capacity interference can be lower than indicated by previous studies (between 5% and 32% of the total impact). In addition, as much as 11% of the total impact can be due to uncontrolled allocation of shared cache Miss Status Holding Registers (MSHRs).

I. Introduction

Chip Multiprocessors (CMPs) or multi-core architectures are the prevalent architecture for modern general-purpose, high-performance processors. In these architectures, it is common to share some part of the hardware-controlled memory system between cores. When multiple processes are run concurrently, the presence of shared resources makes destructive interference possible. In addition, the on-chip shared resources are managed by simple hardware policies that are unaware that the requests belong to different processes. The performance effects caused by destructive interference are hard to predict since they are a consequence of the runtime interaction between the memory request streams from co-scheduled processes. Consequently, destructive interference is an undesirable property and a considerable research effort has been aimed at developing techniques that reduce its performance impact [1, 2].

Figure 1 illustrates that the current CMP memory systems are unable to provide predictable performance. To evaluate interference, we use a baseline configuration called the *private mode* where the benchmark is run in one of the processing cores while the remaining cores are idle. Consequently, it
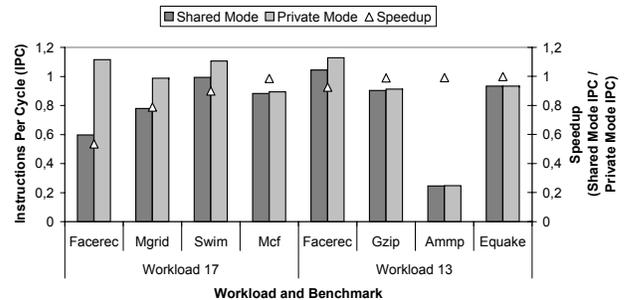
Figure 1. Performance Impact of Interference in the 4-core, Crossbar-Based CMP with 4 Memory Channels

has exclusive access to all shared resources. Conversely, all benchmarks in a workload are run concurrently and compete for access to the shared resources in the *shared mode*. Figure 1 shows the private- and shared mode IPCs of all benchmarks in two of our 40 randomly generated workloads. These measurements are taken from the 4-core, crossbar-based architecture with 4 memory channels which is the architectural configuration with the lowest amount of interference of the configurations used in this work. In workload 17, *facerec* and *mgrid* are heavily impacted by interference with a performance reduction of 46% and 21%, respectively. However, the performance of *mcf* is only reduced by 1%. This illustrates that the performance impact of interference can be substantial and that it does not affect all running processes equally. Furthermore, the performance impact of interference is unpredictable since *facerec* is only slowed down by 7% in workload 13. Since these effects are clearly undesirable, there is a need for architectural techniques that provide predictable performance and improve fairness.

Previously, cache capacity interference has received a great deal of attention [1, 3–7] while only a few researchers have proposed techniques that reduce memory bus interference [2, 8, 9]. Furthermore, there has been little interest in the details of designing a complete, thread-aware memory system [10–12]. A first step towards a unified approach to reducing interference in the hardware-managed memory system is to develop an understanding of the problem. For
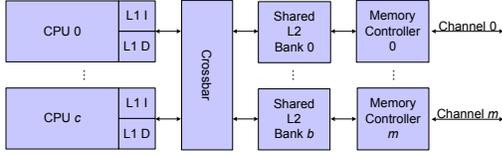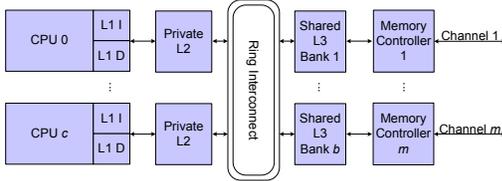
Figure 2.   Crossbar-based CMP

Figure 3.   Ring-based CMP

instance, we found that memory bus interference accounts for 64% of the total amount of interference while cache capacity interference only accounts for 25% with a powerful 4-channel memory bus in our 4-core crossbar-based CMP. When the complexity of current fair cache sharing techniques is taken into account, the fairness requirements on the system must be strict for thread-aware cache techniques to be worth the cost.

In this work, we aim to increase the understanding of the interference problem and thus help architects achieve good complexity/fairness trade-offs. This understanding is developed through detailed analysis of interference at the memory request level. Consequently, we are able to analyze both the relative interference impact of the different shared units as well as the distribution of interference penalties. Handling memory bus interference yields the largest gain, and we believe that employing a fairness-aware technique here will be sufficient for many architectures and usage scenarios. However, we have also observed interference due to shared cache Miss Status Holding Register (MSHR) allocation which must be handled if the fairness requirements are sufficiently strict. Finally, we show that the main driver of memory system interference is insufficient memory bus bandwidth. Since this parameter is limited by the number of physical pins on a chip and the electronic characteristics of the circuit board, it is likely that thread-aware memory bus schedulers will become a necessity in the near future.

II. Related Work

It is common to aim an interference reduction technique at providing fairness and/or Quality of Service (QoS). A memory system is fair if the performance reduction due to interference between threads is distributed across all processes in proportion to their priorities [5]. QoS is provided if it is possible to put a limit on the maximum slowdown a process can experience when it is co-scheduled with any other process [1]. Furthermore, the allowed slowdown can depend on the priority of the process. In other words, the objective of fairness techniques is not to remove interference completely but to equalize its impact on all running processes.

There has been a considerable amount of research on how the performance impact from interference can be reduced in the hardware-controlled, shared memory system. However, most of these studies have focused on a single component of the entire system. For example, techniques have been proposed to reduce cache capacity interference [1, 3–7], cache bandwidth interference [13] and memory bus transfer interference [2, 8, 9]. Unfortunately, a technique that reduces interference in one component is not adequate to provide interference control for the complete memory system. Consequently, a few researchers have investigated how a chip-wide resource management technique can be designed. Iyer et al. [11] proposed a high-level framework for implementing a QoS-aware memory system, while Nesbit et al. [12] proposed the Virtual Private Machines framework where a private virtual machine is created by dividing the available physical resources among applications. In addition, Bitirgen et al. [10] showed how machine learning can be applied to the resource allocation problem. The focus of these works has been to partition all shared resources amongst processes according to some allocation policy. In this work, we investigate the impact of interference and provide guidance on how trade-offs can be handled in resource allocation implementations.

III. Methodology

A. Chip Multiprocessor Architectures

There is still considerable debate regarding the high-level organization of CMPs [14–16]. Therefore, we use two different CMP architectures that are similar to current general-purpose, high-performance CMP implementations for our interference investigations. Furthermore, we scale these architectures according to the expected improvements in process technology [17]. The first CMP type uses a crossbar interconnect to connect the private L1 caches to a large, shared L2 cache as shown in Figure 2. Unfortunately, the crossbar does not scale in terms of area [18]. Consequently, we also use a different CMP model where a bi-directional ring is used as the interconnect. Since the ring has lower bandwidth than the crossbar, we add a private L2 cache to each processor to reduce the number of accesses to the interconnect. This is reasonable since the ring uses considerably less area than the crossbar. Furthermore, the number of processing cores and memory bus channels can be configured in both processor models which makes it possible to investigate the impact of memory system interference across a wide range of realistic CMP architectures. For convenience, we will refer to these architectures by the tuple

c-i-m where c is the number of cores, i is the interconnect and m is the number of memory bus channels.

B. Measuring and Reporting Interference

To gather accurate interference measurements, it is convenient to compare to a baseline where interference does not occur [19]. In this work, we create such a baseline by letting the process run in one processing core and leaving the remaining cores idle. Consequently, the process has exclusive access to all shared resources and we will refer to this configuration as the *private mode*. Conversely, all processing cores are active and the processes compete for the shared resources in the *shared mode*. Mutlu and Moscibroda observed that memory system interference is related to the memory latencies in the shared and private modes with the formula: *interference penalty = shared mode latency − private mode latency* [8].

In our CMP models, there are three shared units: the interconnect, the memory bus and the shared cache. To assess the interference impact of each of these units, we partition the memory request latency through the shared memory system as shown in Table I. For the interconnect, we divide the latency into three types: entry, transfer and delivery. The interconnect has a finite entry queue. If this queue becomes full, the interconnect can not accept any more requests and the request is delayed in the private cache MSHR. We refer to this as *Interconnect Entry Interference* if it causes a different delay in the shared mode than in the private mode. Furthermore, the shared cache can block. In this case, all requests waiting behind a request for a blocked bank are delayed since reordering requests can cause starvation. We refer to interference arising from this situation as *Interconnect Delivery Interference*. Finally, *Interconnect Transfer Interference* is the difference between the shared mode and private mode latencies when there is no cache blocking.

In the memory bus, we divide the latency into two types: entry and transfer. Again, the entry delay is the number of cycles the request is kept in an MSHR before it is accepted into the memory bus queue. If this latency is different for the shared and private modes, we refer to it as *Memory Bus Entry Interference*. In addition, *Memory Bus Transfer Interference* is the difference between the memory bus queue latency plus service latency in the two modes. Since there is no buffer allocation in the shared cache on a response, the memory bus does not have a delivery latency.

Finally, competition for space in the shared cache can lead to *Cache Capacity Interference*. Unlike the interference types discussed above, cache capacity interference does not have a latency value directly associated with it. The key observation is that if a request experiences a bus transfer latency in the shared mode and no bus transfer latency in the private mode, we have a miss in the shared cache that would have been a hit if the process had the entire cache
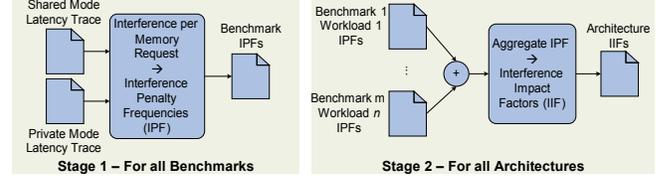

Figure 4.   Interference Measurement Workflow

to itself. The extra latency caused by this event in our CMP models is the number of cycles used to service the request in the memory bus. Consequently, the latency penalty of cache capacity interference is the sum of the bus entry latency and the bus transfer latency of the request.

Figure 4 illustrates the two stage process of gathering interference measurements and aggregating them for a single architecture. In the first stage, we create a compact representation of the measured interference for each benchmark in all workloads and architectures. First, we record the latency of all shared mode memory requests and all private mode memory requests. For all shared mode requests, we find the corresponding private mode request and compute the interference penalties for all interference types. If there are more than one request for the same address, we assume that the requests occur in the same order in both the private and shared modes. Then, we create a histogram representation of the data by counting the number of requests that experience a certain interference penalty for each interference type. For example, if a request for memory address 15 experiences 12 cycles of interconnect transfer interference, we add 1 to the interconnect transfer interference entry at position 12. We refer to this data as the *Interference Penalty Frequency (IPF)*, and stage 1 of the analysis is complete when we have created IPF files for all workloads and architectures.

Stage 2 is the process of aggregating the per benchmark IPF files into one file for each architecture. First, we sum the request counts for each interference penalty from all files belonging to the architecture of interest. For some of the interference types, it is very common to not experience interference. These entries are of little interest and will dominate the results if we use plot the number of requests per interference penalty directly. Consequently, we devise a new metric called the *Interference Impact Factor (IIF)* that balances the latency penalty of interference against the probability of it arising (i.e. $IIF(i) = i \cdot P(i)$). For example, an experiment that results in 15 requests with 3 cycles interconnect transfer interference and 100 requests in total gives $IIF(3) = 3 \cdot \frac{15}{100}$. When we have computed the IIFs for all interference penalties, stage 2 is finished. In most cases, there is a large range of possible interference values and there is a need to summarize the IIFs for a range of interference penalties into a single number. To do this, we use the *Aggregate Interference Impact Factor (AIIF)* which is

Table I
SHARED MEMORY SYSTEM LATENCY BREAKDOWN

Type	Description
Interconnect Entry	The number of cycles a request was kept in the private cache MSHR before it is accepted into a interconnect queue
Interconnect Transfer	The number of cycles spent in the interconnect queue plus the interconnect transfer latency
Interconnect Delivery	The number of cycles a request was delayed because a shared cache bank could not accept requests due to insufficient buffer space
Memory Bus Entry	The number of cycles a request was delayed in a shared cache MSHR before it was accepted into a memory controller queue
Memory Bus Transfer	The number of cycles a request spent in the memory controller queue plus the number of cycles used to retrieve the requested data from DRAM
Cache Capacity	The number of cycles used to service misses that would not occur if the process had exclusive access to the shared cache

simply the sum of the IIFs for all or a subset of the observed interference penalties (i.e. $AIIF(a, b) = \sum_{i=a}^{b} IIF(i)$).

C. Processor Model Scaling

To investigate the impact of interference in multi-core architectures, it is important that reasonable parameters are used to scale the latency, bandwidth and capacity of the various units in the memory system. To this end, we have used the International Technology Roadmap for Semiconductors [17] to estimate scaling trends and CACTI 5.3 [20] to find reasonable caches for the multi-core architectures used in this work. Table II summarizes the main multi-core model parameters. With each improvement in feature size, we double the number of processing cores but use the same core implementation. Furthermore, we follow the ITRS expectation that the interconnect transfer latency will roughly double with each technology generation. The only exception is the per hop latency of the 4-core ring architecture which we assume is limited by the cache cycle time. To account for this latency increase, we double the ring bandwidth across generations. Since the ITRS projections for off-chip bandwidth results in a large range of possible pin counts, we simulate all architectures with 1, 2 and 4 independent memory channels.

Table III contains the parameters of our scaled on-chip caches. Here, we choose to keep the percentage of the total chip area occupied by L2 and L3 caches in the ring-based CMP constant. We use the same shared cache for the crossbar based CMP, but here we only use two levels of caches. Consequently, we assume that the area made available by using a two level cache hierarchy is sufficient to implement a crossbar interconnect. To reduce the shared cache access time and increase the opportunity for cache access parallelism, we divide the shared cache into 4 banks.

D. Simulation Methodology

We use the system call emulation mode of the cycle-accurate M5 simulator [22] for our experiments. The processor architecture parameters for the simulated CMPs are shown in Table IV. Table V contains the interconnect and memory bus parameters, and the cache parameters are outlined in Table III. We have extended M5 with crossbar and ring interconnects and a detailed DDR2-800 memory

bus and DRAM model [23]. For the shared mode, we generated 40 different 4-core workloads (Table VI), 20 8-core workloads (Table VII) and 10 16-core workloads (Table VIII) by picking benchmarks at random from the full SPEC CPU2000 benchmark suite [24]. The only requirement given to the random selection process is that a benchmark can only appear once in each workload. These workloads are fast-forwarded for 1 billion clock cycles before we gather traces for 100 million clock cycles. For our interference measurement methodology to be accurate, it is critical to minimize the difference between the memory requests in the shared and private modes. To ensure this, we use static cache partitioning and an infinite bandwidth interconnect and memory bus during fast forwarding such that the simulation sample starts on a similar instruction in both modes. Furthermore, we run the shared mode experiments first and then retrieve the number of instructions the benchmark committed. Then, we run the private mode simulation for the exact same number of instructions.

Since our processor cores are out-of-order, we can get cache misses from wrong path instructions that only occur in either the private or shared mode. Secondly, the start and termination of the simulation sample is not perfectly synchronized between the two modes. Thirdly, our memory controller reorders requests to achieve high page hit rates which can affect the private cache access patterns and miss rates. For these reasons, there can be small differences between the private and shared mode memory request traces. We remove these differences by applying two preprocessing steps before analyzing the traces. Firstly, we remove the requests for addresses that only occur in the private or shared modes. Secondly, we remove the superfluous requests of the mode that has the most requests in the cases where there are a different number of requests for the same address in the shared and private modes. These steps result in the removal of 0.1% of the observed requests.

IV. RESULTS

Modern out-of-order processors and memory systems contain a substantial amount of logic dedicated to hiding memory latency. Since our interference measurement methodology is latency focused, it is necessary to verify that the observed interference result in an asymmetric per-

Table II
ARCHITECTURE PARAMETER SCALING

	Crossbar Based Architecture			Ring Based Architecture		
	4-core	**8-core**	**16-core**	**4-core**	**8-core**	**16-core**
ITRS Year of Production	2007	2010	2013	2007	2010	2013
Feature Size (nm)	65	45	32	65	45	32
Shared Cache Size (MB)	8	16	32	8	16	32
Memory Bus Channels	1, 2 or 4	1, 2 or 4	1, 2 or 4	1, 2 or 4	1, 2 or 4	1, 2 or 4
Interconnect Latency (End-to-End/Per Hop)	8/-	16/-	30/-	-/4	-/4	-/8

Table III
CACHE PARAMETERS

Cache	Size (4-core/8-core/16-core)	Associativity	Access Latency (cycles)	Cycle Time (cycles)	MSHRs / WB (per bank)	Banks	Area (mm^2)
Level 1 Private Cache	64KB	2	3/2/2	2	16MSHRs/4WB	1	2.3/1.1/0.5
Level 2 Private Cache	1 MB	4	9/6/5	4/3/2	16	1	14.6/7.0/3.6
Level 2/3 Shared Cache	8/16/32 MB	16	16/12/12	4	16/32/64	4	94.0/91.9/84.7

Table IV
PROCESSOR CORE PARAMETERS

Parameter	Value
Clock frequency	4 GHz
Reorder Buffer	128 entries
Store Buffer	32 entries
Instruction Queue	64 instructions
Instruction Fetch Queue	32 entries
Load/Store Queue	32 instructions
Issue Width	4 instructions/cycle
Functional units	4 Integer ALUs, 2 Integer Multipy/Divide, 4 FP ALUs, 2 FP Multiply/Divide
Branch predictor	Hybrid, 2048 local history registers, 4-way 2048 entry BTB

Table V
INTERCONNECT AND DRAM INTERFACE

| Parameter	Value
Crossbar Interconnect	8/16/30 cycles end-to-end transfer latency, 32 entry request queue, Pipelined (2/4/6 pipe stages)
Ring Interconnect	4/4/8 cycles per hop transfer latency, 1/1/2 pipe stages per hop, 32 entry request queue, 1/2/2 request rings, 1 response ring
Point to Point Link	4/3/2 transfer latency, 32 entry request queue
Main memory	DDR2-800, 4-4-4-12 timing, 64 entry read queue, 64 entry write queue, 1 KB pages, 8 banks, FR-FCFS scheduling [21], Closed page policy
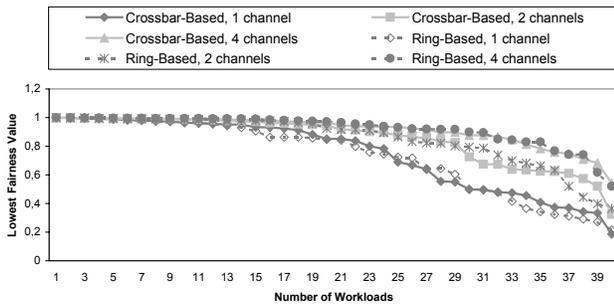
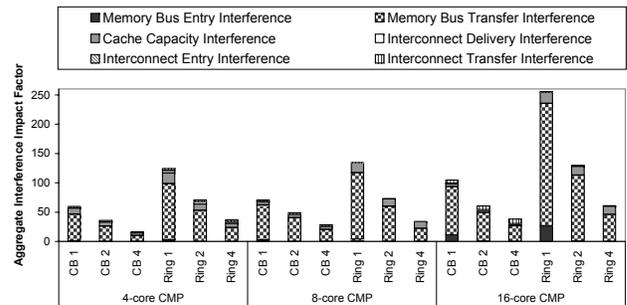Figure 5.   4-core Fairness Metric Values

Figure 6.   Interference Impact Breakdown

formance reduction. To this end, we use the fairness metric of Gabor et al. [25]. This metric expresses the difference between the largest and smallest shared mode slowdowns for one workload and provides values in the range from 0 to 1 where 1 indicates that the slowdown is the same for all benchmarks. A value of 0 indicates that at least one benchmark is not making forward progress.

Figure 5 shows the distribution of fairness metric values for all 4-core CMPs used in this work. Here, we plot the lowest fairness value observed when a certain number of workloads are taken into account for the different CMP ar-

chitectures. The main observation from Figure 5 is that many workloads have reasonably good fairness values. However, there are also workloads where interference leads to large performance differences between the benchmarks (i.e. low fairness). This supports the claim that interference-aware techniques are necessary to reduce performance variability.

Figure 6 shows the interference results for all architectures examined in this work. The main observation is that memory bus transfer interference is the major interference contributor across all architectures. This trend is also visible in Figure 5. Cache capacity interference is the second most important

Table VI
RANDOMLY GENERATED 4-CORE MULTIPROGRAMMED WORKLOADS

ID	Benchmarks	ID	Benchmarks	ID	Benchmarks	ID	Benchmarks	ID	Benchmarks
1	mesa, twolf, art, vpr	9	crafty, twolf, bzip, perlbmk	17	mgrid, facerec, mcf, swim	25	twolf, crafty, bzip, art	33	swim, gap, vortex1, perlbmk
2	art, vortex1, applu, crafty	10	eon, twolf, galgel, crafty	18	equake, applu, eon, gzip	26	applu, gap, perlbmk, crafty	34	equake, twolf, bzip, galgel
3	gap, eon, art, wupwise	11	vortex1, eon, art, equake	19	galgel, mesa, gzip, gcc	27	galgel, facerec, eon, mesa	35	applu, eon, fma3d, vortex1
4	fma3d, applu, parser, swim	12	gzip, lucas, twolf, apsi	20	art, galgel, parser, eon	28	vpr, crafty, applu, vortex1	36	lucas, ammp, twolf, fma3d
5	mcf, swim, gzip, vortex1	13	facerec, ammp, gzip, equake	21	bzip, gzip, perlbmk, eon	29	twolf, vpr, swim, wupwise	37	eon, parser, bzip, mcf
6	swim, galgel, apsi, applu	14	swim, sixtrack, mgrid, vortex1	22	vpr, swim, apsi, gcc	30	parser, mesa, vortex1, gcc	38	vpr, vortex1, wupwise, applu
7	gzip, wupwise, eon, equake	15	sixtrack, fma3d, parser, mcf	23	art, applu, perlbmk, mesa	31	lucas, mgrid, sixtrack, gap	39	lucas, mgrid, swim, gzip
8	sixtrack, gcc, facerec, perlbmk	16	twolf, galgel, crafty, applu	24	facerec, eon, bzip, mesa	32	facerec, galgel, vpr, sixtrack	40	gzip, swim, eon, fma3d

Table VII
RANDOMLY GENERATED 8-CORE MULTIPROGRAMMED WORKLOADS

ID	Benchmarks	ID	Benchmarks	ID	Benchmarks	ID	Benchmarks
1	ammp, mcf, vpr, fma3d, equake, sixtrack, galgel, bzip	6	applu, mcf, perlbmk, parser, crafty, eon, galgel, fma3d	11	ammp, lucas, wupwise, eon, twolf, fma3d, gcc, equake	16	apsi, ammp, vortex1, vpr, gap, perlbmk, art, bzip
2	crafty, vortex1, facerec, ammp, bzip, parser, mcf, perlbmk	7	fma3d, gzip, lucas, perlbmk, bzip, apsi, crafty, gap	12	mcf, galgel, gap, gzip, swim, sixtrack, vpr, fma3d	17	gzip, art, equake, facerec, eon, apsi, gcc, wupwise
3	lucas, vpr, mesa, apsi, swim, art, gzip, twolf	8	swim, gzip, ammp, facerec, perlbmk, equake, gcc, apsi	13	mesa, fma3d, gap, lucas, wupwise, galgel, sixtrack, parser	18	perlbmk, gap, parser, swim, sixtrack, fma3d, lucas, vortex1
4	art, mcf, perlbmk, wupwise, ammp, applu, mesa, swim	9	gap, mcf, vpr, apsi, vortex1, lucas, parser, applu	14	bzip, mgrid, facerec, art, eon, swim, equake, apsi	19	lucas, mesa, apsi, fma3d, mcf, parser, crafty, gcc
5	eon, apsi, equake, vpr, fma3d, facerec, gcc, vortex1	10	mcf, sixtrack, vpr, swim, gzip, mgrid, ammp, lucas	15	swim, vpr, gap, facerec, twolf, sixtrack, mcf, crafty	20	gcc, perlbmk, sixtrack, parser, vortex1, eon, facerec, galgel

Table VIII
RANDOMLY GENERATED 16-CORE MULTIPROGRAMMED WORKLOADS

ID	Benchmarks	ID	Benchmarks
1	lucas, art, ammp, bzip, sixtrack, vpr, gzip, fma3d, equake, gcc, vortex1, facerec, galgel, crafty, apsi, twolf	6	parser, mesa, bzip, vortex1, vpr, fma3d, gap, gcc, perlbmk, gzip, mcf, crafty, eon, equake, facerec, galgel
2	lucas, ammp, mgrid, bzip, swim, crafty, galgel, equake, vortex1, parser, vpr, eon, wupwise, gzip, twolf, mcf	7	gzip, sixtrack, gap, fma3d, eon, galgel, perlbmk, art, bzip, ammp, equake, lucas, parser, facerec, apsi, crafty
3	lucas, ammp, art, bzip, twolf, applu, facerec, apsi, mesa, eon, swim, galgel, gzip, crafty, gap, perlbmk	8	perlbmk, gzip, apsi, twolf, wupwise, gap, vpr, mgrid, galgel, facerec, gcc, eon, mcf, lucas, fma3d, ammp
4	crafty, twolf, mgrid, applu, wupwise, swim, parser, fma3d, mesa, perlbmk, facerec, gcc, lucas, vortex1, galgel, bzip	9	mgrid, art, facerec, gcc, vpr, gzip, parser, ammp, fma3d, galgel, crafty, applu, twolf, bzip, mcf, apsi
5	bzip, facerec, vortex1, ammp, gzip, swim, fma3d, equake, lucas, apsi, applu, vpr, perlbmk, sixtrack, mcf, mesa	10	apsi, swim, crafty, art, sixtrack, ammp, galgel, lucas, vortex1, gzip, perlbmk, vpr, gcc, mesa, gap, equake

source of interference, but its impact is considerably smaller than the impact of bus interference. In addition, there are architectures (e.g. 16-CB-4) where the impact of cache capacity interference is small. Finally, there is more interconnect transfer interference in the crossbar interconnect than in the ring for the 16-core CMP. This seemingly counter intuitive result is due to two factors. Firstly, the ring-based architecture has a private L2 cache that reduces the pressure on the interconnect. Secondly, we do not increase the number of banks in the shared cache which reduces the parallelism available in the crossbar.

Figure 7 shows the interference distribution for the 4-core CMP for both interconnects and all memory bus configurations used in this work. Here, the interference impact factors are aggregated into bins of size 300, and we remove all bins that have a AIIF value of less than 0.35 to improve readability. While Figure 6 showed that interference is reduced when more memory bus bandwidth is made available, Figure 7 illustrates that the interference distribution also changes significantly. For the bandwidth constrained architectures (e.g. Figure 7(a) and 7(d)), the interference impact increases to a maximum before it decreases. In the 4-channel architectures (Figure 7(c) and 7(f)), the largest interference impact is in the 0 to 300 bin and the impact decreases rapidly. The interference impact of the low penalty bins is significantly higher for the 4-channel architectures but the total impact is lower because of the distribution's short tail.

Figure 7 illustrates that the cache capacity interference impact is heavily dependent on the amount of memory bus

(a) 4-Ring-1   (b) 4-Ring-2   (c) 4-Ring-4

(d) 4-Crossbar-1   (e) 4-Crossbar-2   (f) 4-Crossbar-4
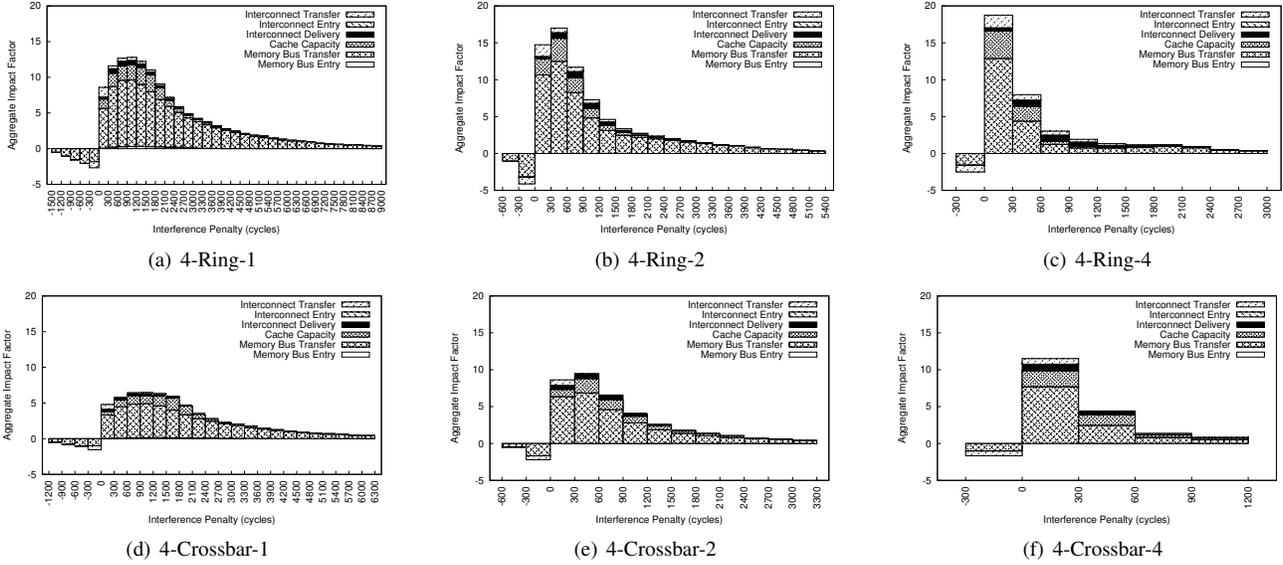
Figure 7.   4-core CMP Interference Impact (*cores-interconnect-channels*)

interference. The reason is that the cost of cache capacity interference is the memory bus service time of the additional requests. Furthermore, the impact from interconnect transfer interference is small across all architectures. Although this interference type occurs very frequently, the interference penalty is small which results in a low interference impact. In addition, there is some interconnect delivery interference in all architectures which is due to shared cache blocking. The impact from this type of interference is large enough that it most likely must be dealt with in architectures with strict QoS requirements.

There is also a considerable amount of constructive interference. With the 4-Ring-1 architecture (Figure 7(a)), constructive memory bus interference leads to a noticeable impact in the -1500 to -1200 cycles bin. This can be explained by taking into account that our memory controller allows some requests to skip past the queue to achieve higher page hit rates and better memory bus utilization [21]. For the interconnect transfer interference, the impact from constructive interference is much lower. In this case, the constructive interference is due to some benchmarks having significant interconnect delays when they have the memory bus to themselves. In the shared mode, memory bus interference reduces execution speed enough that the interconnect congestion disappears which results in lower transfer delays in the shared mode.

To illustrate the impact on interference by increasing the number of processing cores, we show the results of two 16-core ring-based architectures in Figure 8. Here, we use a bin size of 500 and only show bins that have an AIIF value of 1.0 or more. As expected, Figure 8(a) shows that there is a large amount of interference if the memory bus bandwidth is not scaled with the number of cores. Furthermore, memory bus
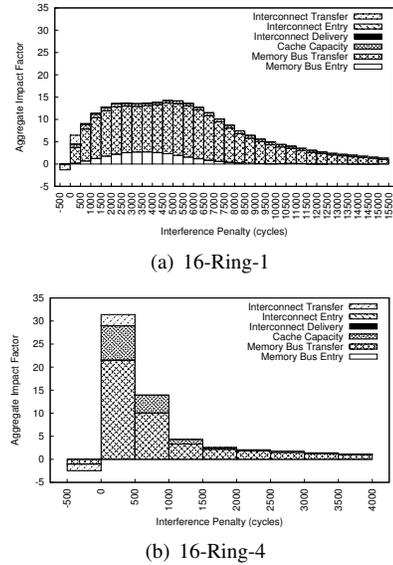

(a) 16-Ring-1

(b) 16-Ring-4

Figure 8.   16-core Ring Interference Impact

entry interference has a considerable impact for this architecture. Consequently, a significant part of the interference is due to shared cache misses not being accepted into the memory bus queue because it is full. This further illustrates the need for fair buffer management observed in all 4-core architectures. Figure 8(b) shows the effect of increasing the number of memory bus channels to 4. Here, the distribution has a considerably shorter tail. However, the impact of the 0 to 500 cycle bin is large which indicates that low-penalty interference is frequent. In other words, providing more resources reduces the impact of interference but does not

remove it. This indicates that fairness techniques are useful even when there are no severe performance bottlenecks.

V. CONCLUSION AND FURTHER WORK

In this work, we have shown that the impact of interference will increase as more cores are added to the chip by investigating a variety of realistic CMP architectures with 4, 8 and 16 cores. Consequently, techniques that reduce this interference are needed in future CMPs. We found that memory bus interference is the major source of interference and it is responsible for between 63% and 87% of the total interference impact depending on the architectures. Furthermore, it is unlikely that this situation will improve in the future as memory bus bandwidth is limited by the number of physical pins on a chip and the electronic characteristics of the circuit board. We also observed that cache capacity interference can be a relatively small part of the total interference impact (between 5% and 32%). Consequently, adding a fair memory controller might be sufficient to achieve acceptable fairness and QoS for many near-term architectures. However, we have also observed architectures where 11% of the total interference impact is due to the shared cache MSHR allocation policy for which no solutions are currently known.

In this work, we have developed an understanding of memory system interference that can be useful for future research. However, we have only investigated CMPs where no fairness techniques have been implemented. A possible avenue of further work is to investigate how implementing fairness techniques in one shared unit will influence the interference impact of the other shared units. For instance, a cache capacity sharing technique might reduce the overall number of cache misses enough to reduce the impact of memory bus interference. On the other hand, it can potentially increase the number misses by limiting the cache space available to a process which might result in more memory bus interference. In addition, we observed that shared cache blocking and memory controller blocking can be important contributors to interference in certain architectures. One possible solution to this problem is to allocate MSHR entries and memory bus queue space per thread. However, this must be done carefully to ensure that the provided resources are utilized efficiently.

REFERENCES

[1] J. Chang and G. S. Sohi, "Cooperative Cache Partitioning for Chip Multiprocessors," in *ICS '07: Proc. of the 21st Annual Int. Conf. on Supercomputing*, 2007, pp. 242–252.

[2] O. Mutlu and T. Moscibroda, "Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems," in *ISCA '08: Proc. of the 35th An. Int. Symp. on Comp. Arch.*, 2008, pp. 63–74.

[3] R. Iyer, "CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms," in *ICS '04: Proceedings of the 18th An. Int. Conf. on Supercomputing*, 2004, pp. 257–266.

[4] F. Guo, Y. Solihin, L. Zhao, and R. Iyer, "A Framework for Providing Quality of Service in Chip Multi-Processors," in *MICRO 40: Proc. of the 40th An. IEEE/ACM Int. Symp. on Microarchitecture*, 2007.

[5] S. Kim, D. Chandra, and Y. Solihin, "Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture," in *PACT '04: Proc. of the 13th Int. Conf. on Parallel Architectures and Compilation Techniques*, 2004, pp. 111–122.

[6] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan, "Gaining Insights into Multicore Cache Partitioning: Bridging the Gap between Simulation and Real Systems," in *HPCA '08: Proc. of the 13th Int. Symp. on High-Perf. Comp. Arch.*, 2008.

[7] N. Rafique, W.-T. Lim, and M. Thottethodi, "Architectural Support for Operating System-driven CMP Cache Management," in *PACT '06: Proc. of the 15th Int. Conf. on Parallel Architectures and Compilation Techniques*, 2006, pp. 2–12.

[8] O. Mutlu and T. Moscibroda, "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors," in *MICRO 40: Proc. of the 40th Annual IEEE/ACM Int. Symp. on Microarchitecture*, 2007.

[9] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith, "Fair Queuing Memory Systems," in *MICRO 39: Proc. of the 39th An. IEEE/ACM Int. Symp. on Microarch.*, 2006, pp. 208–222.

[10] R. Bitirgen, E. Ipek, and J. F. Martinez, "Coordinated Management of Multiple Resources in Chip Multiprocessors: A Machine Learning Approach," in *MICRO 41: Proc. of the 41th IEEE/ACM Int. Symp. on Microarchitecture*, 2008.

[11] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt, "QoS Policies and Architecture for Cache/Memory in CMP Platforms," in *SIGMETRICS '07: Proc. of the 2007 ACM SIGMETRICS Int. Conf. on Measurement and Modeling of Comp. Sys.*, 2007, pp. 25–36.

[12] K. Nesbit, M. Moreto, F. Cazorla, A. Ramirez, M. Valero, and J. Smith, "Multicore Resource Management," *IEEE Micro*, vol. 28, no. 3, pp. 6–16, 2008.

[13] K. J. Nesbit, J. Laudon, and J. E. Smith, "Virtual private caches," in *ISCA '07: Proc. of the 34th An. Int. Symp. on Comp. Arch.*, 2007, pp. 57–68.

[14] H. Hofstee, "Power Efficient Processor Architecture and the Cell Processor," *HPCA 11: 11th Int. Symp. on High-Performance Comp. Arch.*, pp. 258–262, 2005.

[15] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: A 32-Way Multithreaded Sparc Processor," *IEEE Micro*, vol. 25, no. 2, pp. 21–29, 2005.

[16] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: a Many-core x86 Architecture for Visual Computing," in *ACM SIGGRAPH 2008*, 2008, pp. 1–15.

[17] ITRS, "International Technology Roadmap for Semiconductors - 2007 Edition," http://www.itrs.net/.

[18] R. Kumar, V. Zyuban, and D. M. Tullsen, "Interconnections in Multi-Core Architectures: Understanding Mechanisms, Overheads and Scaling," in *ISCA '05: Proc. of the 32nd Int. Symp. on Comp. Arch.*, 2005, pp. 408–419.

[19] S. Eyerman and L. Eeckhout, "System-Level Performance Metrics for Multiprogram Workloads," *IEEE Micro*, vol. 28, no. 3, pp. 42–53, 2008.

[20] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi, "CACTI 5.1," HP Laboratories Palo Alto, Tech. Rep., 2008.

[21] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory Access Scheduling," in *ISCA '00: Proc. of the 27th An. Int. Symp. on Comp. Arch.*, 2000, pp. 128–138.

[22] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, "The M5 Simulator: Modeling Networked Systems," *IEEE Micro*, vol. 26, no. 4, pp. 52–60, 2006.

[23] *DDR2 SDRAM Specification*, JEDEC Solid State Technology Association, May 2006.

[24] SPEC, "SPEC CPU 2000 Web Page," http://www.spec.org/cpu2000/.

[25] R. Gabor, S. Weiss, and A. Mendelson, "Fairness and throughput in switch on event multithreading," in *MICRO 39: Proc. of the 39th Int. Symp. on Microarchitecture*, 2006, pp. 149–160.