Managing Shared L2 Caches on Multicore Systems in Software

David Tam*, Reza Azimi*, Livio Soares*, and Michael Stumm*

*Department of Electrical and Computer Engineering
University of Toronto, Toronto, Canada M5S 3G4
Email: {tamda, azimi, livio, stumm}@eecg.toronto.edu

Abstract—**Most of today's multi-core processors feature shared L2 caches. A major problem faced by such architectures is cache contention, where multiple cores compete for usage of the single shared L2 cache. Uncontrolled sharing leads to scenarios where one core evicts useful L2 cache content belonging to another core. To address this problem, we have implemented a software mechanism in the operating system that allows for partitioning of the shared L2 cache by guiding the allocation of physical pages. This mechanism, which can also be applied to virtual machine monitors, provides isolation capabilities that lead to reduced contention. We show that this mechanism is effective in reducing cache contention in multiprogrammed SPECcpu2000 and SPECjbb2000 workloads. Performance improvements of up to 17% were achieved without adversely affecting co-scheduled applications.**

In order to effectively size L2 cache partitions, a quantifiable metric is needed to properly predict performance as a function of L2 cache size. For page management, Miss Rate Curves (MRCs) have proven to be useful for this purpose. However, for L2 cache sizing, we have found L2 MRCs to be inadequate and have found instruction retirement Stall Rate Curves (SRCs) to be more effective, where the stalls are caused by memory latencies.

I. INTRODUCTION

On-chip shared L2 cache architectures are common in today's multi-core processors, such the Sun Niagara, IBM Power5, and Intel Core architecture. Shared caches have important advantages such as increased cache space utilization, fast inter-core communication (via the high-speed shared L2 cache), and reduced aggregate cache footprint through the elimination of undesired replication of cache lines.

A major disadvantage of shared L2 caches, however, is that uncontrolled contention can occur by allowing CPU cores to freely access the entire L2. As a result, scenarios can occur where one core constantly evicts useful L2 cache content belonging to another core without obtaining a significant improvement itself. Such contention causes increased L2 cache misses which in turn leads to decreased application performance.

Uncontrolled L2 sharing also reduces the ability to enforce priorities and to provide Quality-of-Service (QoS). For example, a low priority application running on one core that rapidly streams through the L2 cache can consume the entire L2 cache and remove most of the working set of higher priority applications co-scheduled on another core.

Many researchers in the architecture community have recognized the problem of uncontrolled contention in the L2 cache and have explored different hardware support for dynamically partitioning the L2 cache [1], [2], [3], [4], [5], [6], [7]. Some of these hardware solutions are effective and may eventually appear in future processors. We argue that an alternative, operating system-level solution is viable by exploiting the hardware performance monitoring features available in many existing microprocessors.

In this paper, we present such a software solution based on a low-overhead and flexible implementation of cache partitioning through physical page allocation on a real operating system (Linux) running on a real multi-core system (IBM Power5). We show how to exploit the hardware performance monitoring features of the microprocessor to aid in determining the L2 partition size that should be given to each application. More specifically, for each application, we generate L2 Miss Rate[1] Curves (MRCs) and instruction retirement Stall Rate Curves (SRCs), where the stalls are caused by memory latencies. We use these curves to predict L2 contention to guide the CPU scheduling algorithm in deciding which subsets of applications should be co-scheduled.

In our experimental analysis, we first show the negative impact of uncontrolled sharing of the L2 cache and then show how our software cache partitioning algorithm can eliminate this negative impact. We used SPECcpu2000 and SPECjbb2000 as our workloads, running Linux 2.6.15 on an IBM Power5 CMP system. Our experimental results indicate that by carefully partitioning the L2 cache and co-scheduling compatible applications appropriately, improvements as high as 17% in total system IPC can be achieved without adversely affecting any of the co-scheduled applications.

One of the insights we obtained by experimenting on a real system was that L2 MRCs alone were inadequate in predicting co-scheduled performance impact on our system. We found that an application's rate of instruction retirement stall due to memory latencies, as a function of L2 cache size, to be more effective. This instruction retirement stall rate curve (SRC), incorporates factors such as (1) the L2 cache miss rate; (2) instruction retirement stall sensitivity to L2 cache misses; (3) non-uniform access latencies to lower levels of the memory hierarchy, such as the L3 victim cache, local main memory, and remote main memory; and (4) shared

---

[1]We use miss *rate*, rather than miss *ratio*, since *rate* incorporates time and gives an indication of access intensity that can be easily compared among applications.

memory bus contention [8]. On our system, we found that the instruction retirement stall rate due to L1 data cache misses was sufficient to reasonably predict performance as a function of L2 cache size, and was directly obtainable from the hardware performance counters. Section VI provides more details.

II. RELATED WORK

Many researchers in the architecture community have recognized the cache contention problem in shared L2 caches and have proposed hardware support for partitioning the cache [1], [2], [3], [4], [5], [6], [7]. Some of these hardware solutions are effective, with reasonable complexity and resource consumption, and may be eventually implemented in real processors in the future. Our work explores an alternative solution that is entirely based on software using hardware performance monitoring features of today's microprocessors. Our software-based approach has the advantage of being implementable and deployable today. Moreover, it is more flexible as it can be built with standard hardware performance monitoring features and does not add to the design complexity of already complex microprocessors. While the hardware solution proposed by Qureshi and Patt [6] can achieve up to 23% performance improvement on a simulated platform, our software-based solution running on a real system is able to achieve up to 17% improvement.

The work closest to our approach is by Cho and Jin, who propose a software-based mechanism for L2 cache partitioning based on physical page allocation [9], [10]. However, the major focus of their work is on how to distribute data in a Non-Uniform Cache Architecture (NUCA) to minimize overall data access latencies. In contrast, we concentrate solely on the problem of uncontrolled contention on a shared L2 cache. Furthermore, we have implemented our solution in a real environment based on features available in today's processors. This enables us to examine the impact of the cache partitioning on real processor performance using hardware performance counters. Similar to their philosophy, we advocate low-overhead, flexible software solutions that help to simplify the hardware. Due to their target platform, they used a simulation environment (SimpleScalar) that does not take the interference of the operating system into account.

III. DESIGN & IMPLEMENTATION

To provide software-based L2 cache partitioning, we simply apply the classic technique of OS page-coloring [9], [10], [11], [12], [13]. When a new physical page is required by a target application, the OS allocates a page that maps onto a section of the L2 cache assigned to the target application. By doing so for every new physical page request of the target application, we isolate L2 cache usage of the application.

Fig. 1 illustrates the page-mapping technique in general. In a physically indexed L2 cache, every physical page has a fixed mapping to a physically contiguous group of cache lines. The figure shows that there are several physical pages labeled *Color A* that all map to the same group of physically
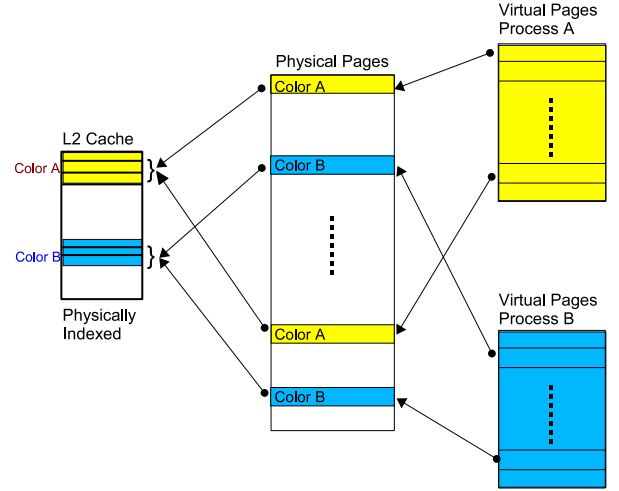
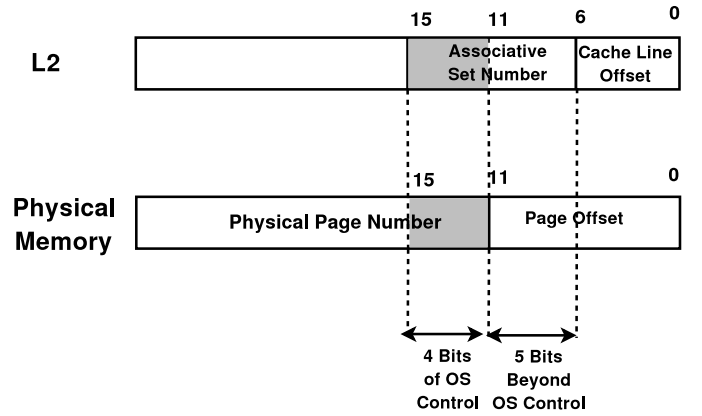

Fig. 1. Page and cache line mapping.

Fig. 2. Bit-field perspective of partitioning on the Power5.

contiguous L2 cache lines labeled *Color A*. For the Power5 processor used in our experiments, there are 32 physically contiguous cache lines to a page because the page size is 4 KB and the cache line size is 128 bytes. The figure also shows that physical pages of the same color are given to the same application. For example, physical pages of *Color A* have been assigned solely to application process *A*. The OS is responsible for this mapping of virtual-to-physical memory pages and it is this capability that enables control of L2 cache usage and isolation. We implemented this mechanism in Linux 2.6.15 by modifying the physical page allocation component of the OS. For non-targeted applications, the OS allocates any free physical page when requested, thus these non-targeted applications would not adhere to any cache partition restrictions.

Although we implemented this mechanism at the OS level in this paper, this solution could also be applied at the level of a virtual machine monitor.

A. Partitioning on the Power5

Using the page-mapping technique, the Power5 processor is able to support 16 distinct colors, which we will refer to as *partitions*. Fig. 2 illustrates a bit-field perspective of the page-mapping technique. The upper 4 bits of the L2 cache set number field overlap with the bottom 4 bits of the physical page number field. Since the operating system has direct control of the physical page number field, it has 4 bits of influence on the L2 cache set number. The lower 5 bits of the L2 cache set number, which are beyond the direct control of the operating system, means that there are 32 sets per partition.

Note that there are no bits in the physical address that are related to set-associativity because eviction within each set is managed online, at run-time by the hardware using an LRU policy.

The L2 cache on the Power5 is physically implemented as 3 symmetric slices, where the slice number is determined by a hash function using bits 8 to 27 inclusively. Unfortunately, 4 of these bits are beyond the direct control of the operating system, meaning that slice usage appears uniformly random. Having directly control of the L2 slice usage would have enabled us to support 48 partitions.

Our L2 cache partitioning mechanism also causes the L3 victim cache of the Power5 to be divided into 16 partitions. The derivation is similar to the L2 cache derivation and is therefore not shown. Each partition in the L2 has a direct mapping to a corresponding partition in the L3 victim cache.

B. OS Implementation

Although partitioning physical memory is a fairly simple concept, its implementation in the Linux kernel must be done carefully to prevent any negative performance side effects. In our first attempt, we simply used a single free list of physical pages for each CPU core. Having a single free list, however, incurred expensive linear search of the potentially long free list frequently.

Another problem with having a single free list is that upon application termination, a large number of pages are freed and put at the head of the free list. Since these pages were assigned to the recently terminated application, they may not be suitable for another application running on a different core. As a result, a linear traversal of the free list scans through a potentially large number of unsuitable pages before it finds the first suitable page.

To address this issue, we converted the single free list into multiple free lists (still on a per CPU basis). Each list contains free physical pages that map to a designated section of the L2 cache. Having multiple free lists, each corresponding an L2 partition, dramatically accelerates the process of finding a suitable page, as it removes the need for linearly searching the free list. A simple Round-Robin scheme is used when multiple free lists are eligible to select a free page from. Since the Power5 L2 cache can be divided by software to have a maximum of 16 partitions, we had 16 free lists for each CPU in the system.

When a large number of physical pages are requested at once, Linux allocates a group of physically contiguous pages as long as the groups are powers of 2 in size. To support this, Linux maintains lists of groups of contiguous free pages of size 1 to 1024 pages (i.e., level 0 to level 10). For page allocation of levels higher than 0 we use the single free list of the target level (i.e., for levels higher than 0, we do not use separate lists per partition). We traverse the list to find a suitable page group that maps to the target L2 partition. Since allocations at higher levels is fairly rare, we do not foresee this case impacting performance significantly.

When a new physical page is being allocated for an application and its assigned partition free list is empty, then the allocator must request additional free pages from the Linux buddy allocator. A problem may arise in that none of the pages returned by the buddy allocator have the color of the target partition. Even repeated attempts may be unsuccessful. We employ a configuration parameter, `MaxTry`, to limit the number of such attempts. If, after `MaxTry` attempts, the partition free list remains empty, the physical page is allocated from another partition free list chosen randomly. The default value for `MaxTry` is set to 100.

IV. Experimental Setup

The multiprocessor used in our experiments is an IBM OpenPower 720 computer, as specified in Table I. It is an *8-way* Power5 consisting of a 2x2x2 SMPxCMPxSMT configuration[2]. Each chip has 1.875 MB of shared L2 cache that is shared between the on-chip cores. There is an off-chip 36 MB L3 victim cache per chip. As mentioned previously, Linux 2.6.15 was used and modified to allow for L2 cache partitioning. With the given hardware, a total of 16 partitions are possible on the Power5 processor, each of size 120 KB in the L2 cache and 2.25 MB in the L3 victim cache.

To create a controlled execution environment for our experiments, the Linux task scheduler was modified to completely disable the default reactive and pro-active task migration mechanisms and policies. Our partitioning mechanism has no compatibility issues with process migration across cores since physical-to-virtual page mappings remain unchanged. In addition and for the same reasons, our mechanism is independent of the number of cores sharing the L2 cache.

The workloads used were SPECjbb2000 and SPECcpu2000. The IBM J2SE 5.0 JVM was used to run SPECjbb (1 warehouse configuration). For SPECcpu, 20 out of the 26 applications were run using the standard *reference* input set. (The remaining 6 applications, which were mostly Fortran-based, did not compile successfully.) To simulate a multiprogrammed server environment, various combinations of these applications were run together.

V. Results

With software-based partitioning, we have the ability to study the impact of L2 cache size on execution time. Fig. 3

---

[2]2 chips x 2 cores per chip x 2 hardware threads per core.

| Item	Specification
# of Chips	2
# of Cores	2 per chip
CPU Cores	IBM Power5, 1.5 GHz, 2-way SMT
L1 ICache	64 KB, 128-byte lines, 2-way associative, per core
L1 DCache	32 KB, 128-byte lines, 4-way associative, per core
L2 Cache	1.875 MB, 128-byte lines, 10-way associative, 14 cycle latency, per chip
L3 Victim Cache	36 MB, 256-byte lines, 12-way associative, 90 cycle latency, per chip, off-chip
RAM	8 GB (2 banks x 4 GB), 280/310 cycle latency local/remote
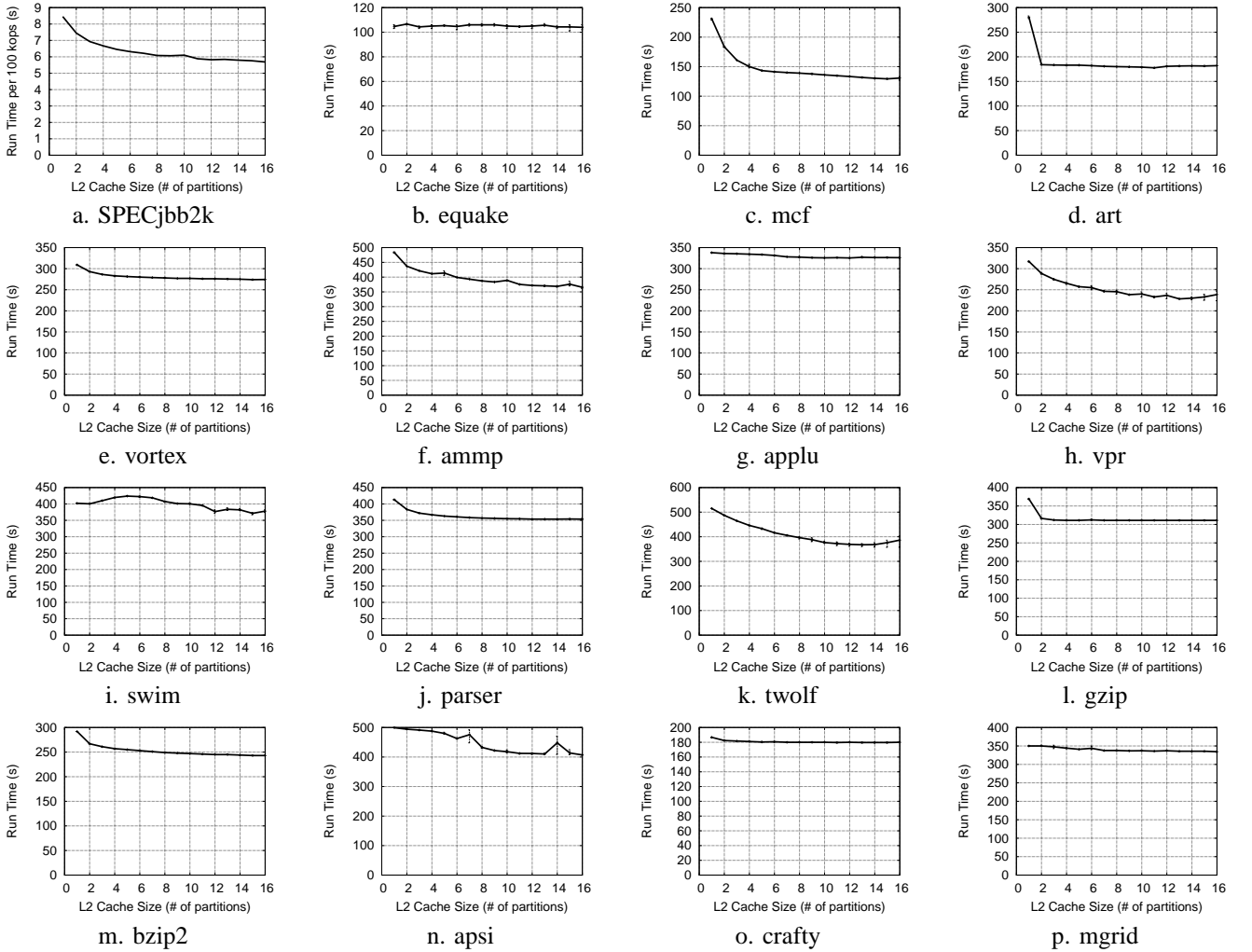
Fig. 3.   Single-programmed application performance as a function of L2 cache size.

shows the impact of varying the L2 cache size using our partitioning mechanism. Each application was run alone on a single core. Applications *gap*, *wupwise*, *mesa*, *gcc*, and *sixtrack* are not shown because they had flat curves similar to *mgrid* in Fig. 3(p). The results of Fig. 3 are similar in spirit to the initial graphs shown by Qureshi and Patt [6] but our results come from a software implementation running on a real system. We show application-reported run times for the entire run of the application, which includes all overheads. Our results here are difficult to compare against those obtained by Qureshi and Patt because we use a 1.875MB L2 cache that is partitioned at physical page granularity while Qureshi and Patt used a 1MB L2 cache partitioned by hardware at cache line, set-associative way granularity. Furthermore, they generated their results by simulating a fairly short fraction of application execution (250 million instructions) whereas we ran applications for much longer periods (several tens of billions of cycles).

Our results indicate that for some applications, having a small fraction of the cache is sufficient to achieve performance

close to the performance achieved with the entire cache. For example, SPECjbb requires 8 partitions, *mcf* requires 4 partitions, and *art* requires 2 partitions.

Most graphs show monotonically decreasing execution times as the cache size is increased, as expected. However, there are a few exceptions. For instance, *swim* shows increasing execution times as the cache size is increased from 1 to 5 partitions, and *twolf* shows increasing execution times as the cache size is increased from 12 to 16 partitions. We are currently investigating such anomalous cases using detailed hardware performance monitoring tools.

It is important to note that with the initial single-programmed results shown in Fig. 3, the impact of co-scheduling two or more applications on a single chip without software-based cache partitioning cannot be easily predicted. One important characteristic that is missing from Fig. 3 is the L2 cache usage demands of each application. For example, an application could exhibit streaming behavior consisting of high L2 cache access frequency and no reuse frequency, leading to a high miss frequency.

Fig. 4 shows the impact of software cache partitioning on performance for seven combinations of multiprogrammed workloads. Each application was run on its own core but within the same chip so that the L2 cache is indeed shared[3]. The units shown are IPC improvement per billion cycles as reported by the hardware performance counter tools developed by our research group in previous work [14], [15]. The performance is normalized to the performance of the same combination of applications without partitioning. We skip the first 30 billion cycles of execution and capture IPC data for the next 60 billion cycles using hardware performance counters.

The bottom x-axis shows the number of partitions (N) given to one application, while the remaining $16 - N$ partitions are given to the second application, as indicated by the top x-axis. For example, when SPECjbb is given 12 partitions in Fig. 4(a), *equake* is given the remaining 4 partitions. The graph indicates that SPECjbb can achieve a throughput improvement of up to 8% (12 partitions) while *equake* is penalized by less than 5%. If SPECjbb is intended to be the high priority application while *equake* is the low priority application, then these priorities could be enforced with software-based cache partitioning. As an extreme example, SPECjbb could be given 14 partitions with the remaining 2 partitions given to *equake*, resulting in a 13% improvement to SPECjbb while penalizing *equake* by 8%. For the SPECjbb+equake combination, we used a `MaxTry` value of 25000, rather than the default. Using a lower value caused the performance of SPECjbb to begin showing degradations from 11 to 15 partitions. This occurred because *equake* would exceed the `MaxTry` 100 threshold frequently since it was allowed only 1 to 5 partitions. Upon this occurrence, *equake* would obtain physical page belonging to SPECjbb instead.

Fig. 4(b) indicates that the performance of *mcf* can be

improved by up to 11% (14 partitions) without noticeably affecting *art*. In Fig. 4(c), *vortex* can be improved by 5% (10 partitions) without affecting *art*. If *art* is a lower priority task, then *vortex* can be improved by up to 8% while penalizing *art* by 3%. In Fig. 4(d), *ammp* can be improved by 5% (14 partitions) while penalizing *applu* by 2.5%. In Fig. 4(e), *twolf* can be improved by 8% (13 partitions) without penalizing *gzip*. The drop in IPC for *twolf* at 14 and 15 partitions is due to the interference from *gzip* upon exceeding the `MaxTry` 100 threshold. A similar situation occurred in Fig. 4(f) as well, from 1 to 4 partitions in *vpr*, which helped *vpr* without significantly affecting *swim*. Finally, Fig. 4(g) also shows the same phenomenon in *swim* between 1 to 4 partitions.

Although not shown in this paper, we observed no impact on the L1 instruction cache in the SPECcpu applications. However for SPECjbb, as the size of the L2 cache was decreased from 5 partitions to 1 partition, we observed a noticeable increase in the instruction retirement stall rate due to L1 instruction cache misses.

VI. QUANTIFYING CHIP SHARING INTERFERENCE

Understanding and characterizing the performance impact of sharing resources on a CMP chip is an essential part of (a) predicting which combinations of applications exhibit performance interference and, (b) quantifying the potential performance improvements of controlling resource sharing, in this case, L2 cache sharing. In this section, we demonstrate that cache partitioning can recover up to 70% of degraded IPC due to chip sharing. Furthermore, we detail how hardware performance counters can be used to predict the potential performance interference between applications executing on different cores of the same chip.

Fig. 5 shows the performance degradation suffered by applications when executing as a pair, compared to executing in single-programmed mode. Each application in the multi-programmed pair is executed on its own core but within the same chip, thus sharing the L2 cache. Two different setups are plotted: (1) the reduction of IPC, normalized to single-programmed mode, of applications executing with no cache partitioning, and (2) the reduction of IPC, normalized to single-programmed mode, of applications executing with a "best", fixed, manually selected partition size.

With most combinations shown, cache partitioning is able to significantly reduce the IPC degradation due to chip sharing of one of the applications, while possibly slightly degrading the IPC of the second application. The worst degradation seen is in *equake* when run together with SPECjbb; *equake*'s IPC suffers a 4% decrease, while enabling a 9% improvement of SPECjbb's IPC. The best improvement is seen in *twolf* when run together with *gzip*. This combination shows that cache partitioning can recover up to 70% of degraded IPC due to chip sharing.

The main reason behind most of the benefit seen by controlling L2 cache sharing is the fact that while some applications are memory intensive in their behavior, they may not benefit from using the entire L2 cache. This is the case, for example,

---

[3]Examining the impact of SMT, by running both applications on the same core, is beyond the scope of this paper.

a. SPECjbb+equake b. mcf+art c. art+vortex d. ammp+applu

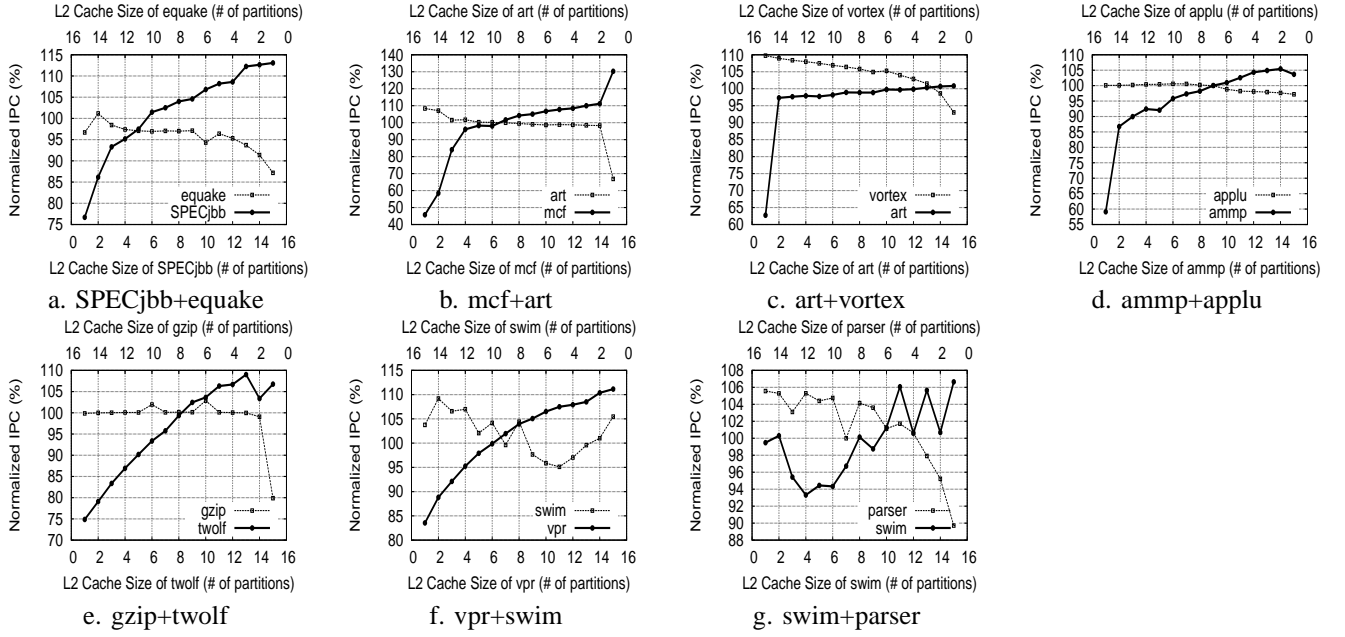e. gzip+twolf f. vpr+swim g. swim+parser

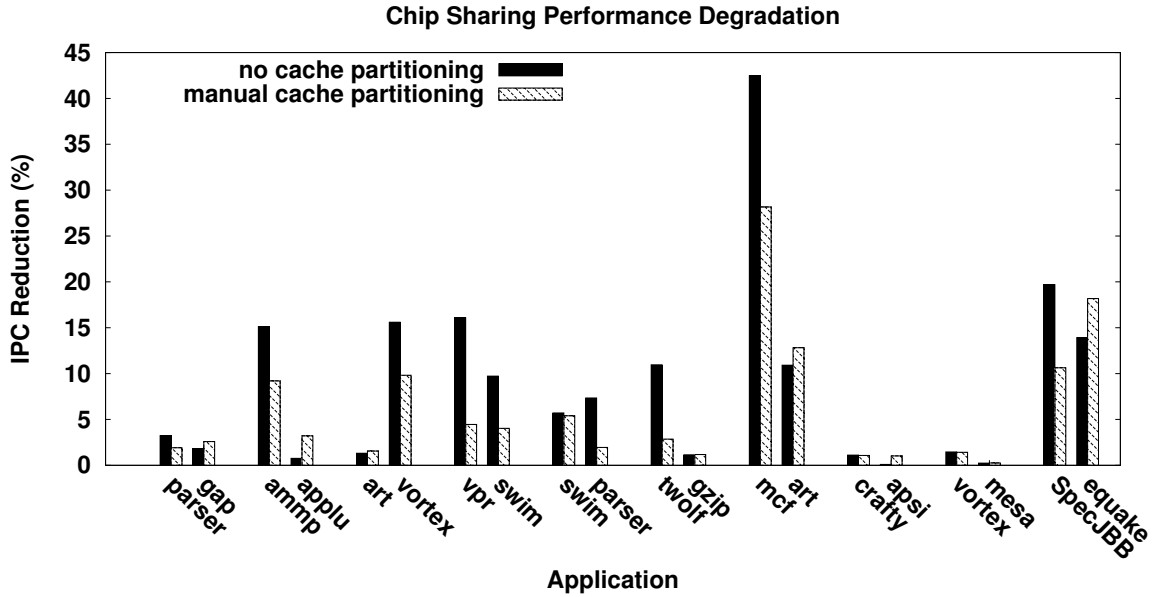Fig. 4. Multiprogrammed workload performance as a function of L2 cache size.

Fig. 5. IPC comparison of applications executing in isolation on a CMP chip, versus executing while sharing the chip.

for the *art* application. Fig. 6 shows the number of cycles in which instruction retirement is stalled due to L1 data cache misses in a billion cycles, with varying L2 cache partition sizes, as collected by the Power5 hardware performance counters. In this figure, each application is executed alone on the chip. Fig. 6(a) shows the variation of memory access related stalls for *art*. There are two notable observations. First, the run time curve shown in Fig. 3(d) closely resembles curve 6(a), demonstrating that for this application, memory stalls seen by the core are strongly related to its performance. Second, it is clear that giving more than 2 L2 cache partitions to

art does not improve its memory performance. The same characteristics apply to *swim* and *gzip*, as can be seen by comparing Fig. 3(i) and Fig. 3(l) to the various combinations in Fig. 5, and described in various papers regarding their access intensities [7], [8], [16], [17].

The curves for *mcf* and *vortex* in Fig. 6, however, show a different behavior. The number of memory related stalls monotonically decreases as the number of partitions grows. As can be seen in Fig. 5, both *mcf* and *vortex* show performance benefits with cache partitioning when executing along side of *art*, because the partitioning isolates the lack of locality seen in
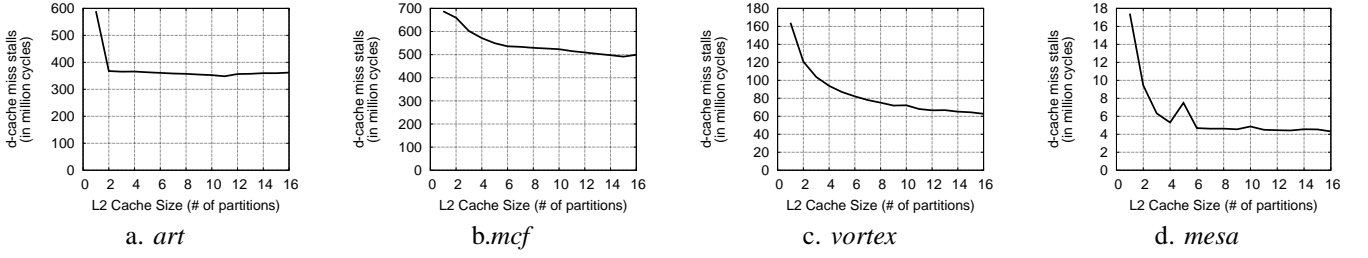
Fig. 6. Data cache stall rate curve (SRC): Number of cycles in which instruction retirement is stalled due to L1 data cache misses per billion cycles, as derived from the PowerPC performance counters. Single-programmed mode.
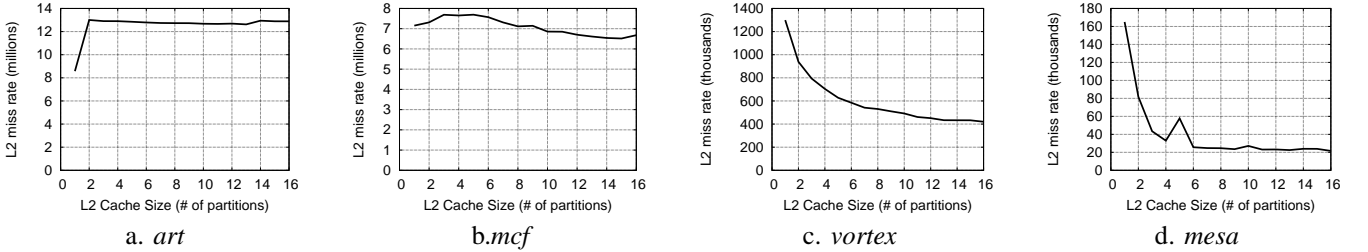

Fig. 7. L2 miss rate curve (MRC) of varying L2 cache partition sizes, as reported by the PowerPC performance counters. Single-programmed mode.

art and avoids the replacement of useful cache lines belonging to the other application, which otherwise would have been the case with the LRU hardware mechanism.

Finally, the performance of a few application combinations are not affected when sharing the same chip. This is the case for the *apsi+crafty*, *vortex+mesa*, and *gap+parser* combinations shown in Fig. 5. This can also be explained by analyzing the instruction retirement stalls due to L1 data cache misses. As can be seen in Fig. 6(d), although *mesa* shows sensitivity to varying cache partitions, instruction retirement is stalled due to L1 data cache misses for, typically, only around 5% of the cycles. This indicates that *mesa* has very low cache requirements and is unlikely to replace important cache lines from applications executing on a sibling core when using the default LRU hardware mechanism.

A. Stall Rate Curve Versus Miss Rate Curve

Fig. 7 shows the L2 miss rate curves for the same four applications in Fig. 6. While the stall rate curves (SRCs) in Fig. 6 directly measure instruction retirement stalls caused by the memory hierarchy, the L2 miss rate curves in Fig. 7 measure only a single component of the performance picture, which is the *rate* of misses experienced at the L2 cache only.

It is interesting to note that in some scenarios, the L2 miss rate is not sufficient to accurately predict the performance impact of memory operations because it does not account for the penalty of misses. In a multi-level cache hierarchy, the penalty of an L2 cache miss can vary dramatically depending on the source from which the cache miss is served. For example, when varying from 1 to 2 partitions with *art*, although there is a significant performance increase and L1 data cache stalls drop, the L2 miss rate curve does not show a corresponding decrease. Rather, it shows an increase in the miss rate. By
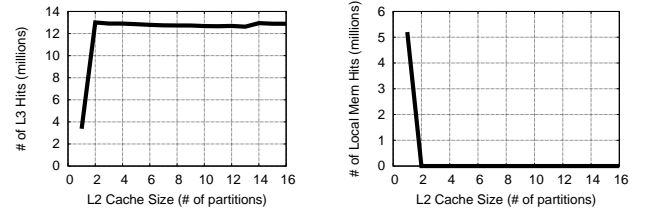

Fig. 8. L3 victim cache and local memory hit rates for *art*, per billion cycles, as reported by the PowerPC performance counters. Single-programmed mode.

examining the L3 victim cache and local memory hit rate curves, shown in Fig. 8, it is apparent that the source of the performance improvements is due to the source in which data is acquired. In the 2 partition case, most L2 cache misses are resolved in the L3 victim cache. In contrast, in the 1 partition case, most L2 cache misses are resolved in the local main memory due to L3 partitioning.

VII. DISCUSSION

Due to current hardware indexing of cache lines, software cache partitioning is compatible with larger page sizes up to a certain extent. As the size of a page grows by doubling its size, the number of possible partitions in the L2 decreases by half. However, if the size of a page causes the number of possible partitions to drop below two, then cache partitioning would no longer be possible.

Our partition mechanism does not create load imbalance on the main memory banks of the system since our Power5 system make use of standard interleaved memory design.

In this work, we have assumed that per application L2 MRCs and instruction retirement stall rate curves (SRCs), where stalls are caused by memory latencies, are available to the operating system as they are obtained during profiling runs

and stored in a repository. In order to add a new application to the repository, these curves must be calculated by running the application (or at least a representative portion of it) several times (16 in our setup). Ideally, one might want to calculate an application's L2 MRC online with low overhead. Unfortunately, this is not possible given the existing hardware performance monitoring features of today's microprocessors. Nonetheless, there is much room for speeding up the process of calculating the L2 MRC. For instance, Berg and Hagersten use a software approach based on data address *watchpoints* to calculate MRC online with the runtime overhead of 40% [18].

Secondly, we have assumed the application's L2 MRC and SRC are stable throughout the execution of the application. In reality, each application goes through several *phases* that may have different memory access patterns. To react to such phase changes, dynamic repartitioning of the L2 cache may be required which may potentially incur significant copying of data from one color to another. However, if program phases are long enough to offset the overhead of repartitioning, our software-based approach is still applicable.

VIII. CONCLUSION & FUTURE WORK

We have demonstrated a software-based cache partitioning mechanism and shown some of the potential gains in a multiprogrammed computing environment. Our mechanism allows for flexible management of the shared L2 cache resource. Although we have implemented this mechanism at the operating system level in this paper, it can also be applied at the virtual machine monitor level.

Our experience on a real system has led us to the insight that instruction retirement stall rate curves (SRCs), where stalls are caused by memory latencies, provide more useful information for our purposes than L2 cache miss rate curves (MRCs).

We are currently investigating many other possible combinations of workloads. Other potential workloads that we are considering include SPECcpu2006, and SPECweb2000 multiprogrammed with TPC-C.

We plan to extend the basic mechanism by creating a continuous optimization system that (1) dynamically determines the optimal partition size in an automated, online, low overhead manner using hardware performance monitoring facilities, and that (2) dynamically adjusts the number of partitions given to an application in an online, low overhead manner.

ACKNOWLEDGEMENTS

REFERENCES

[1] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting inter-thread cache contention on a chip multi-processor architecture," in *HPCA*, 2005.

[2] F. Guo and Y. Solihin, "An analytical model for cache replacement policy performance," in *SIGMETRICS*, 2006.

[3] R. Iyer, "CQoS: a framework for enabling QoS in shared caches of CMP platforms," in *ICS*, 2004.

[4] H. Kannan, F. Guo, L. Zhao, R. Illikkal, R. Iyer, D. Newell, Y. Solihin, and C.Kozyrakis, "From chaos to QoS: Case studies in CMP resource management," in *dasCMP*, 2006.

[5] S. Kim, D. Chandra, and Y. Solihin, "Fair cache sharing and partitioning in a chip multiprocessor architecture," in *PACT*, 2004.

[6] M. Qureshi and Y. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Micro*, 2006.

[7] E. Suh, L. Rudolph, and S. Devadas, "Dynamic partitioning of shared cache memory," *The Journal of Supercomputing*, vol. 28, no. 1, pp. 7–26, April 2004.

[8] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith, "Fair queuing cmp memory systems," in *Micro*, 2006.

[9] S. Cho and L. Jin, "Managing distributed, shared L2 caches through OS-level page allocation," in *Micro*, 2006.

[10] ——, "Better than the two: Exceeding private and shared caches via two-dimensional page coloring," in *Workshop on Chip Multiprocessor Memory Systems and Interconnects*, 2007.

[11] B. Bershad, D. Lee, T. Romer, and J. Chen, "Avoiding conflict misses dynamically in large direct-mapped caches," in *ASPLOS*, 1994.

[12] W. Lynch, B. Bray, and M. Flynn, "The effect of page allocation on caches," in *Micro*, 1992.

[13] T. Sherwood, B. Calder, and J. Emer, "Reducing cache misses using hardware and software page placement," in *Supercomputing*, 1999.

[14] R. Azimi, M. Stumm, and R. Wisniewski, "Online performance analysis by statistical sampling of microprocessor performance counters," in *ICS*, 2005.

[15] D. Tam, R. Azimi, and M. Stumm, "Thread clustering: Sharing-aware scheduling on SMP-CMP-SMT multiprocessors," in *EuroSys*, 2007.

[16] J. L. Henning, "SPEC CPU2000: Measuring CPU performance in the new millennium," *IEEE Computer*, vol. 33, no. 7, pp. 28–35, July 2000.

[17] M. Moreto, F. Cazorla, A. Ramirez, and M. Valero, "Explaining dynamic cache partitioning speed ups," *Computer Architecture Letters*, vol. 6, no. 1, Jan-Jun 2007.

[18] E. Berg and E. Hagersten, "Fast data-locality profiling of native execution," in *SIGMETRICS*, 2005.