CacheScouts: Fine-Grain Monitoring of Shared Caches in CMP Platforms

Li Zhao, Ravi Iyer, Ramesh Illikkal, Jaideep Moses, Srihari Makineni and Don Newell
System Technology Lab, Intel Corporation, Hillsboro, Oregon
Contact: {li.zhao, ravishankar.iyer}@intel.com

Abstract

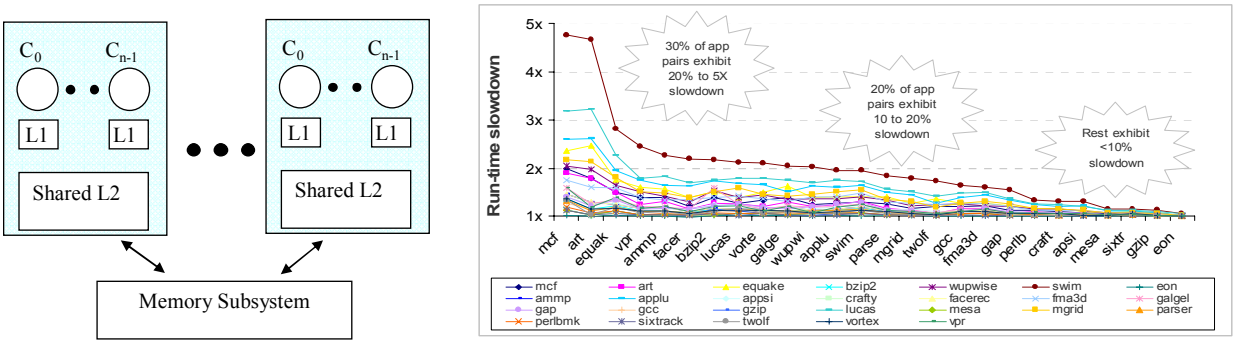
As multi-core architectures flourish in the marketplace, multi-application workload scenarios (such as server consolidation) are growing rapidly. When running multiple applications simultaneously on a platform, it has been shown that contention for shared platform resources such as last-level cache can severely degrade performance and quality of service (QoS). But today's platforms do not have the capability to monitor shared cache usage accurately and disambiguate its effects on the performance behavior of each individual application. In this paper, we investigate low-overhead mechanisms for fine-grain monitoring of the use of shared cache resources along three vectors: (a) occupancy – how much space is being used and by whom, (b) interference – how much contention is present and who is being affected and (c) sharing – how are threads cooperating. We propose the CacheScouts monitoring architecture consisting of novel tagging (software-guided monitoring IDs), and sampling mechanisms (set sampling) to achieve shared cache monitoring on per application basis at low overhead (<0.1%) and with very little loss of accuracy (<5%). We also present case studies to show how CacheScouts can be used by operating systems (OS) and virtual machine monitors (VMMs) for (a) characterizing execution profiles, (b) optimizing scheduling for performance management, (c) providing QoS and (d) metering for chargeback.

1. Introduction

With dual-core and quad-core processors [8][9] already in the marketplace for client and server platforms, we have truly entered the era of chip-multiprocessor (CMP) platforms. All major CPU manufacturers have adopted the CMP architecture and have announced plans to aggressively increase the number of cores integrated on a single chip [10] [14][15]. There are two ways of taking advantage of the high degree of thread parallelism enabled by CMP – (a) parallelize applications to be highly multi-threaded and (b) run multiple applications simultaneously. In this paper, we focus more on the latter approach and investigate multi-application workload scenarios on CMP platforms. The rapid adoption of virtualization [29] as a means to consolidate multiple applications on to a single server platform is a prime example of this approach. In addition, the usage of client platforms (desktops and laptops) is also becoming richer with multiple applications running simultaneously.

When running multiple applications simultaneously on client and server platforms, the key performance challenge is that of platform resource contention. As more cores are enabled, contention for compute resources reduces. However, the contention shifts to the rest of the platform because the compute cores continue to share critical platform resources such as cache, memory and I/O. Several recent studies [1] [4][5][7][16][18][19] have shown that contention for shared last-level cache can cause significant loss in performance, determinism and quality of service (QoS). However, today's CMP platforms have no support to monitor the shared cache resource and to provide an understanding of resource usage breakdown amongst the multiple applications running simultaneously. The knowledge of resource usage is the key to disambiguating the impact of sharing on each individual application's performance, to allow OS/VMM make optimal scheduling decisions and for metering and chargeback.

In this paper, our goal is to propose low-overhead fine-grain shared cache monitoring techniques for CMP platforms. We start by describing the four key usage models namely profiling/characterization, scheduling and performance management, QoS and metering/chargeback, and show that their monitoring requirements essentially include (a) occupancy – how much space is being used and by whom, (b) interference – how much contention is present and who is being affected and (c) sharing – how are applications/threads cooperating. We then compare and contrast several alternatives for this shared cache monitoring in terms of overheads, granularity and accuracy. We propose the CacheScouts monitoring architecture consisting of novel tagging (with software-guided IDs) and sampling mechanisms (with set sampling) to achieve per application monitoring at low overhead and with very little loss of accuracy. We



(a) Typical CMP architecture (b) Example of cache/memory contention

Figure 1. CMP architecture and cache/memory contention

evaluate CacheScouts in two important workload contexts (virtualized CMP servers running three server benchmarks and multi-application CMP client platforms running multiple SPEC CPU2000 applications) and show its efficacy in determining execution profiles. Finally, we perform case studies to show how CacheScouts monitoring data will be used dynamically for (a) optimized scheduling / performance management, (b) quality of service guarantees and (c) metering resource usage for chargeback. We believe that this paper is the first to propose shared cache monitoring mechanisms for CMP architectures, evaluate them for client/server workload environments and show their use along three very different vectors.

The rest of this paper is organized as follows. Section 2 presents important usage models that necessitate monitoring feedback and describes related work in this area. Section 3 compares/contrasts several approaches to implement shared cache monitoring and introduces the CacheScouts architecture. Section 4 evaluates the CacheScouts monitoring options for two multi-application workload scenarios. Section 5 covers detailed case studies that show how these monitoring hooks can be employed by operating systems and virtual machine monitors for better scheduling and performance management, better quality of service and more accurate metering and chargeback. Section 6 concludes the paper and covers future work in this area.

2. Motivation and Background

In this paper, we assume a typical CMP architecture as illustrated in Figure 1(a). As shown in the figure, there are N cores in each die with private L1 caches and a shared L2. Our assumptions on workload scenarios are basically multiple heterogeneous applications running simultaneously on client platforms or multiple virtual machines running simultaneously on server platforms.

When multiple applications or virtual machines (VMs) run simultaneously on such CMP architectures, contention for shared cache space and memory bandwidth is bound to occur. To understand the performance effect of this contention, we performed a simple experiment. We ran SPECCPU2000 applications on the Core 2 Duo desktop (2 cores sharing 4MB of last-level cache) in two modes: (a) dedicated mode where we run one application on one core in the platform and (b) pair-wise mode where we run the application on one of the cores and run another application on the other core. Figure 1(b) shows the slowdown in terms of the ratio of execution time in pair-wise mode over that in dedicated mode. As one can observe, the slowdown can be as high as ~4.76X (when *mcf* is running with *swim*). Overall, we observe that in 30% of the pairs, the slowdown ranges from 1.2X to 5X. We have performed similar studies with virtualized server workloads running simultaneously and observed similar slowdown effects.

Since shared cache contention is clearly high, shared cache monitoring is highly desirable. In particular, the following key usage models motivate the need for shared cache monitoring in both client and server platforms configurations:

[1] Performance Profiling and Characterization: When an application runs alone on a platform, the resources in that platform are dedicated to its execution and it is possible to characterize and model resultant application performance. Software developers tend to assume this is the case and optimize for best performance. However, when multiple applications or VMs run simultaneously (two applications running on Core 2 Duo in Fig. 1b for example), it is difficult to determine the resources like cache space that ends up being given to each individual application. The situation gets worse as the number of applications running increases and is more than the number of cores. Performance monitoring counters that are available in today's platform only provide cache misses on a per core basis. In order to better

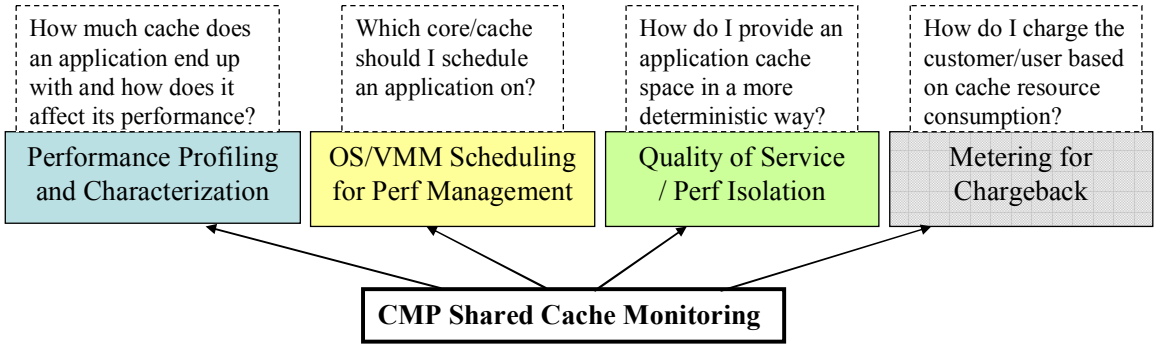


Figure 2. Usage models for CMP shared cache monitoring

characterize the effect of cache contention on each application's individual performance, the minimum amount of information needed is the cache occupancy (i.e. number of cache lines consumed) and misses on a per application basis during the execution time. This data can be then incorporated into profiling tools like Vtune [11] or Oprofile [17] for use in dynamic optimization by developer or administrator.

[2] OS/VMM Scheduling/Performance Management: The job of OS or VMM is to schedule applications or VMs respectively on the cores for maximum performance efficiency. Ideally, there are three aspects that the scheduling policy should take into account in a CMP platform with multiple shared caches: (a) it is better to schedule applications that contend less with each other on the same cache, (b) it is better to schedule applications or threads that share data on the same cache and (c) it is better to affinitize applications to caches if their working set is still available on the cache from the previous scheduling interval. However, the OS/VMM has almost no visibility into cache contention or sharing statistics in today's CMP platform. As a result, OS/VMM scheduling policies do not have the ability to determine which applications should be co-scheduled on the cores sharing the same cache. Researchers [27][3][25][18] have attempted to optimize scheduling by using crude indicators for affinity (such as time elapsed between scheduling intervals) and cache destructiveness (such as high miss rate) for example. However, all of the literature points out that this is not ideal and the crude mechanisms were used primarily due to the lack of shared cache monitoring feedback from the hardware.

[3] Quality of Service and Performance Isolation: In a client environment, there are several foreground and background applications running simultaneously. Typically, the foreground applications are given higher priority in terms of core scheduling. However, since there are multiple cores available in CMP architectures, both foreground and background application will run simultaneously. Since the background application could steal cache/memory resources and affect the foreground application, it is important to implement QoS mechanisms in the cache that provide preferential treatment and a reasonable degree of performance isolation. Similar QoS and performance isolation concerns [2][5][12][20][28] are already becoming apparent in datacenter server environments where VMs of differing priorities based on service level agreements are running simultaneously. In order to guide cache space allocation based on priorities provided by the software layer (OS or VMM) [5][12][20], it is important for the OS/VMM to know how much of the resource (cache space in this case) is being used by each application. As a result, shared cache monitoring is very critical to QoS and performance isolation.

[4] Metering for Chargeback: A common model for hosted server environments is that of running multiple customer applications (potentially in multiple virtual machines) simultaneously on the same platform and determining how the platform resources were used by each application. The resource utilization is used to determine how the customer will be charged for the hosting service. Today, the only resources considered for chargeback are number of cores, memory capacity and disk space. Since shared cache contention can affect performance significantly, it is important to take shared cache space into account when determining the chargeback. This again points to the need for shared cache monitoring and the need for determining how cache space usage plays a role in the overall chargeback model.

3. CacheScouts: Shared Cache Monitoring

In previous section, we described four usage models for shared cache monitoring techniques. While the usage models are quite different, the underlying monitoring requirements are fairly similar in nature. In this section, we start by describing the underlying monitoring requirements, CacheScouts goals and

considerations. We then delve into the overall CacheScouts architecture and compare/contrast implementation options.

3.1. Goals & Requirements

The goal of the CacheScouts architecture is to provide sufficient monitoring data as feedback to guide the questions listed in Figure 2. Based on the usage models, we believe that the following monitoring requirements should be the primary basis for CacheScouts:

[1] Cache Occupancy per Application: One of the fundamental requirements for most usage models is the ability to accurately estimate the amount of cache space consumed by each application during its runtime. This requires that the number of cache lines be counted on a per application basis and be exposed to the OS/VMM so that it can monitor the data at regular intervals.

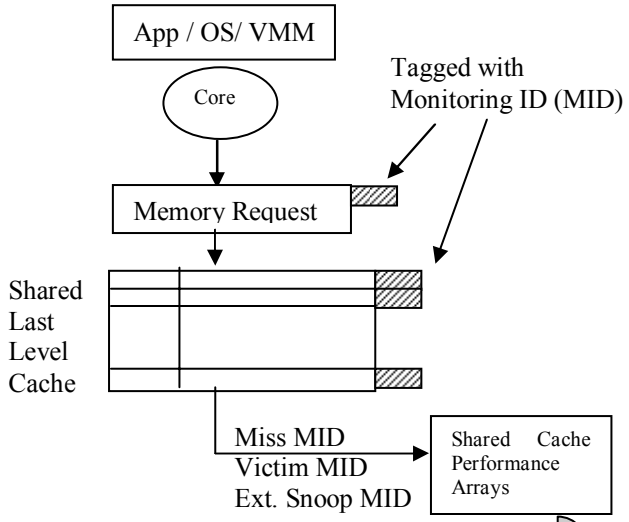
[2] Cache Interference per Application: Another important monitoring requirement is that of capturing the amount of cache interference experienced between applications that are co-scheduled on the same shared cache. For example, consider one application A1 co-scheduled with three other applications A2, A3 and A4. We need to keep track of the number of times that A1's cache lines are evicted by A2, A3 or A4. In addition, it will also be useful to keep track of how often A1 evicted A2, A3 or A4's cache lines. Ultimately a NxN (4x4 in this example) matrix of eviction would provide more detail on how the evictions are distributed.

[3] Cache Sharing per Application: If multiple applications running simultaneously are sharing data in memory, then it is important to keep track of how often a cache miss from one application finds the cache line already brought in another cache by either the same application or another application. This allows the OS to identify applications that should be co-scheduled as well as identify applications that migrate from one cache to another and suffer cold cache misses repeatedly.

3.2. CacheScouts Architecture

In order to satisfy the monitoring requirements listed in the previous section, we propose the CacheScouts shared cache monitoring architecture as illustrated in Figure 3. We describe the architecture first without constraints and then later impose the practical constraints to determine feasible implementation options. The architecture can be decomposed into two major components:

[1] Cache Line Tagging: In order to associate cache lines with the application that is using the data, it is required that we tag the cache lines with a monitoring identity (MID) either when they are allocated or when they are touched. This requires that the cache/memory request generated by the core should be tagged with the MID before it is sent to the shared last-level cache. The overhead of tagging the request is fairly low. However, the overhead of tagging each cache line with its respective MID is the major challenge that needs to be addressed for low-overhead implementation. For example, for caches with 64 byte cache lines, the overhead of a 10-bit MID (covering 1024 possible identities) incurs an overhead of ~2%. While this may seem already low, considering that the vast majority of a processor die (typically 40 to 60%) is the last-level cache, more than 1% of the die will be used up by these monitoring IDs. The basic challenge is to keep the number of MIDs low, but allow the coverage of many more applications to be tracked by the software layer. Another basic challenge is to avoid having to tag every cache line with the MID. Below, we will discuss hardware-based and/or software-guided



(a) Monitoring IDs and Cache Line Tagging

App ID	Occupancy	Interfered	Interfering	Share
1				
2				
…..				
N				

Interference / Sharing Tables

	1	2	……	N
1				
2				
…..				
N				

(b) Shared Cache Performance Arrays

Figure 3. CacheScouts Architecture

MID options and its implications on implementation options.

[2] Shared Cache Performance Arrays: In order to keep track of the occupancy, interference and sharing on a per application basis, we need a new set of shared cache performance arrays that need to be updated when cache events of interest occur. Figure 3(a) describes the flow of information from the cache to the shared cache performance arrays and Figure 3(b) shows the detailed organization of the arrays themselves. For each application (identified by MID), we maintain the following four summary information:

- **Occupancy:** The number of cache lines currently tagged by this application id. In order to accomplish this, we need to increment the occupancy counter when a new line is allocated (upon a miss event) into the cache and decrement the counter when a line is evicted (upon victimization) from the cache.
- **Interference:** There are two forms of interference: (i) how often an application gets evicted from the cache by another application versus (ii) how often it evicts a cache line of another application. The summary performance array keeps track of both as shown in Figure 3(b). This is accomplished by incrementing these counters when the miss MID and the eviction MIDs are different. In order to get a more accurate picture of which applications were causing the most interference, a more detailed NxN matrix can also be maintained shown by the second table in Figure 3(b). This table keeps track of evictions caused by application X (row) on application Y (column).
- **Sharing:** In order to determine cross-cache sharing, we also keep track of the number of cache misses that end up hitting in another cache. As a result, the sharing count needs to be incremented every time an external snoop (for a miss) returns a valid snoop result with a corresponding MID from the other shared cache in the platform. In order to determine exactly which applications were sharing, a more detailed NxN matrix is also maintained as illustrated by the second table in Figure 3(b).

The area overhead of the shared cache performance arrays can be quite significant. Maintaining the summary table for 1024 applications (or MIDs) for instance requires about 16KB (~0.4% of a 4MB cache). However, maintaining a 1024x1024 table for the same number of applications begins to be alarmingly area-intensive. As a result, it is important to reduce the number of MIDs to a more manageable quantity, but retain the accuracy and usability of these shared cache performance counters. In the next few sections, we discuss the implementation options that attempt to reduce the area overhead significantly.

3.3. CacheScouts Implementation Options

In this section, we discuss practical implementation options for the CacheScouts architecture. The main motivation is to find an option that reduces the area overhead significantly, but retains a usable level of granularity and accuracy of the performance monitoring.

3.3.1. Monitoring with Core MIDs

The simplest approach to maintaining occupancy is to maintain it per core. Note that we use the term "core" here to indicate a hardware thread. As a result, the MID would essentially be the core id and therefore be supporting very small number of IDs.

However, there are some serious limitations to this approach. If the number of applications is more than the number of cores, then it becomes very difficult to identify whether a cache line was related to the previous application context running on this core or the current one. Since cache state is persistent across multiple context switches, it is almost impossible to determine which application is currently occupying the cache. While additional counters could be maintained to address this specific problem, there are no elegant heuristics to solve the overall problem of overlaying multiple applications on to the cores without any software guidance. As a result, we rule out this hardware-only approach to monitoring shared cache usage.

3.3.2. Monitoring with Software MIDs

We believe that the most practical and usable solution for monitoring cache space is based on software-guided MIDs. This essentially allows the OS or VMM to associate an MID with each application or virtual machine when it is scheduled on to the core. The software-guided MID can be specified in a control register within the core so that it is saved and restored across context switches. Since there are still a limited number of MIDs (say 32 or 64), the OS or VMM can appropriately recycle the MIDs amongst its running applications and essentially enable sampling of a subset of applications over any given period of time.

For the rest of the paper, we assume that 64 software-guided MIDs are reasonable to employ for most commercial client and server environments. With 64 MIDs, the overhead of tagging each line in the cache is moderately high (~1.25%). It is important to look for novel mechanisms that further reduce this area overhead in the cache.

3.3.3. Using Set Sampling Mechanisms

In order to reduce the overhead of shared cache monitoring and avoid tagging every single line in the cache with a MID, we propose the use of set sampling in the cache. Set Sampling [6][26] has been shown to be very effective in accurately representing the behavior of the entire cache. Researchers have shown that much less than 10% of the sets need to be sampled to achieve significant accuracy in estimating cache behavior. Here, we take advantage of set sampling to reduce the overhead of maintaining MIDs. We choose a random subset of sets in the cache and tag all of the lines within these sampled sets with the MIDs. While we independently evaluate (in Section 4) the number of sets required for sufficient monitoring accuracy, we expect that much less 10% of the sets will be required. As a result, we can reduce the area overhead of MID-enabled caches down to at least 0.125% (instead of 1.25% for the full cache). It should be noted that the MIDs for the sampled sets will now be stored in a separate structure because extending only a few of cache line tags makes the cache layout irregular and complex. At the same time, having a separate structure is more flexible and modular as its design can be done independent of the main cache design.

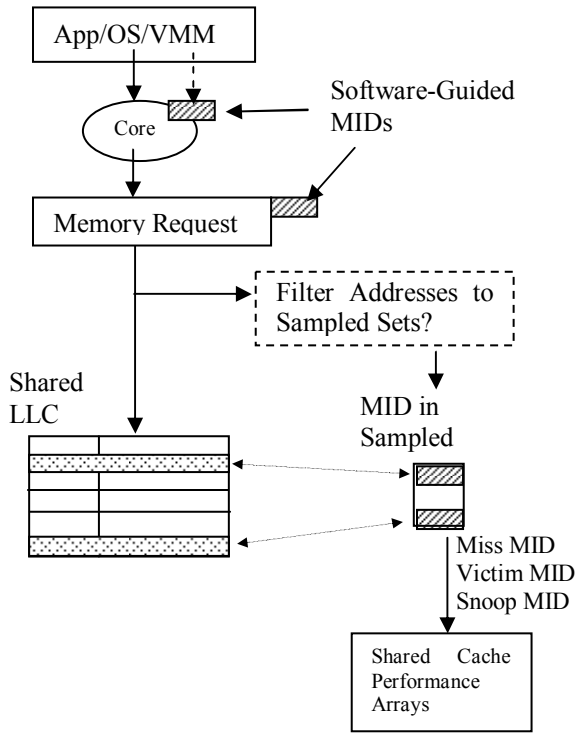


Figure 4. CacheScouts architecture using software MIDs and set sampling

Overall, the combination of software-guided MIDs and set sampling seems like a promising approach to implement the CacheScouts architecture. Figure 4 summarizes the changes to the CacheScouts architecture to implement software-guided MIDs and set sampling.

4. Evaluating CacheScouts Options

In this section, we perform a detailed study of the execution profile and cache performance behavior of two important workload scenarios based on the CacheScouts monitoring data.

4.1. Workloads and Simulations

To understand the behavior of simultaneous applications and virtual machines running together, we chose two configurations:

(1) *Multi-programmed Configuration* with 8 single-threaded SPEC CPU2000 applications [23] running on a 4-core architecture as shown in Figure 5. The 4-core architecture resembles Intel's Core 2 Duo-based quad-core desktop platform. From the SPEC CPU2000 suite, we chose the following eight benchmarks: *swim, mesa, mcf, gzip, eon, bzip2, art* and *apsi*. This mix was chosen to have some workloads that occupy lots of cache, some that are memory-intensive and some that need very little cache.

(2) *Virtualized Server Configuration* with 3 multi-threaded server benchmarks consolidated on an 8-core architecture as shown in Figure 5. The 8-core architecture resembles Intel's latest server platform with two Core 2 Duo-based quad-core processors. We chose the following 3 server benchmarks to run simultaneously.

- **SAP SD/2T:** SAP SD 2-tier [21] is a sales and distribution benchmark to represent enterprise resource planning (ERP) transactions.
- **SPECjbb2005:** SPECjbb2005 [24] models a warehouse company with warehouses that serve a number of districts.
- **SPECjappserver2004:** SPECjappserver 2004 [22] is a multi-tier server benchmark that represents J2EE application servers. SPECjappserver models the information flow between an automotive dealership, manufacturing, supply chain and order/inventory.

Since our primary interest is in cache performance, occupancy, interference and sharing, we employed a cache hierarchy simulator called CASPER [13]. CASPER was modified to maintain the monitoring information (shared cache performance arrays) on a MID basis. To simulate the scheduling of many application threads on fewer cores, we simulated a global scheduling queue of workload traces for each of

the above benchmarks. The traces were allowed to run on available cores for a typical scheduling duration, which is estimated based on the references and instructions in the traces. We ensured that the traces were long enough to capture the behavior of the applications and to warm up the cache for the associated statistics.

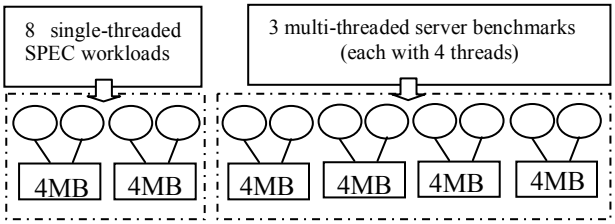


Figure 5. Simulation configurations

4.2. Capacity Profile using CacheScouts

We start by looking at the cache capacity profile for both workload scenarios. Figures 6 and 7 show the capacity data (fraction of the cache size) sampled at regular frequent intervals (the x-axis). Figure 6 shows the data for both caches on the 4-core architecture. It can be observed that the distribution of cache resource usage between the SPEC applications is quite disparate. For example, *mcf* and *art* occupy large portions of the first cache during the execution. On the second cache, *swim* and *mcf* occupy significant portions of the cache, with *art* occupying significant space only in a few samples. Comparing the two charts, we can also observe how applications (like *art* and *mcf)* move from one cache to another and affect cache behavior.

Figure 7 shows the same data for server workloads on the 8-core architecture but only for one of the four caches for brevity. The legend indicates the benchmark name (multiple times for each of the four threads) within the benchmark. The distribution of cache space is less skewed in the multi-VM server configuration since all the three server workloads are cache-sensitive and use a lot of the cache space. At the same time, in any given sample (a bar) the amount of cache can be observed to be quite un-even cross the two benchmarks simultaneously running on that cache.

Another way to look at the cache capacity is at the scheduling points of the application (i.e. when it is ready to be scheduled on the core). For example, let us revisit the cache migration behavior of *art* in Figure 6. From a visual comparison of the two charts, it is clear that *art* migrates from cache 1 to cache 2 at the 35th time sample. Since *art* has a large working set, it is preferable to affinitize it to cache 1 instead of migrating it to cache 2. In order to do so, it is important to dynamically detect the working set size and allow the scheduler to determine which application should be affinitized and which should be migrated. Today, the OS/VMM attempt to determine migration/affinitization actions based on the time elapsed between schedule points. CacheScouts provides direct cache capacity monitoring to the scheduler for this purpose.

4.3. CacheScouts Interference Profile

Figure 8 and 9 show the cache interference data for both workload scenarios. The data is shown on a per application basis and for each scheduling duration during which the application was running on the core since no interference can occur when the application is waiting to be scheduled. The bar illustrates the number of misses that occurred for that application during the scheduling duration. The bar is broken down into the following conditions that can occur on a miss: (a) "Find invalid line" -- an invalid line is found and the line is allocated without conflicting other application, (b) "Conflict Self" -- a victim is found and the victim belongs to the same application and (c) "Conflict Other" -- a victim is found and the victim belongs to a different application. Figure 8 illustrates that three of the five applications (*art*, *mcf* and *swim*) have a high miss count. It can be further observed that *swim* replaces others more often than itself, whereas *mcf* replaces itself more than others. Figure 9 shows the server data for 1 thread out of each benchmark. Among the three benchmarks, SPECjbb has the highest miss rate. However, even with a smaller miss rate, SAP is almost as destructive as SPECjbb because it interferes with others more than with itself. The measure of destruction along with the capacity can be used to do appropriate co-scheduling and migration by the operating system. In section 5, we will show that this interference and the previous capacity information can be used effectively to re-schedule applications dynamically and improve overall performance.

4.4. CacheScouts Sharing Profile

Figure 10 shows the cache sharing profile for the server workloads since we already know that SPEC applications do not share any data with each other. We can see that SPECjappserver has the maximum amount of sharing and therefore the threads belonging to SPECjappserver should be co-scheduled together. SPECjbb and SAP have small amounts of sharing and it is not clear whether co-scheduling is necessary for these workloads. In Section 5, we will show that using CacheScouts feedback, we can optimize VMM scheduling algorithms to take advantage of sharing by co-scheduling relevant threads.

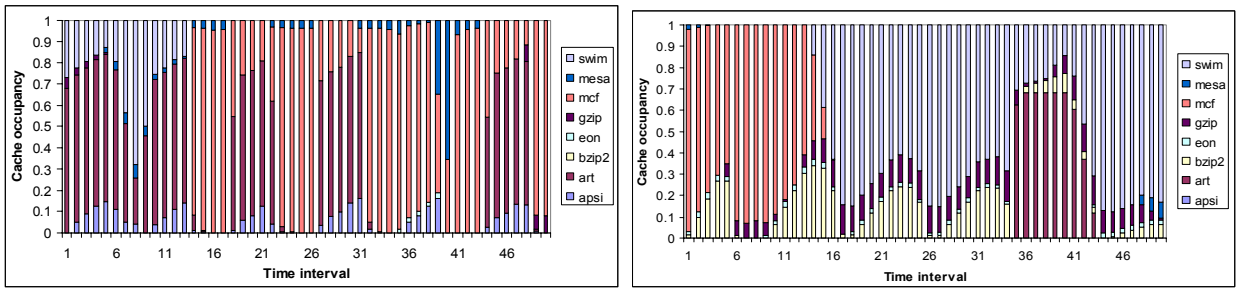


Figure 6. CacheScouts capacity profile for multi-programmed workloads (for both the caches)

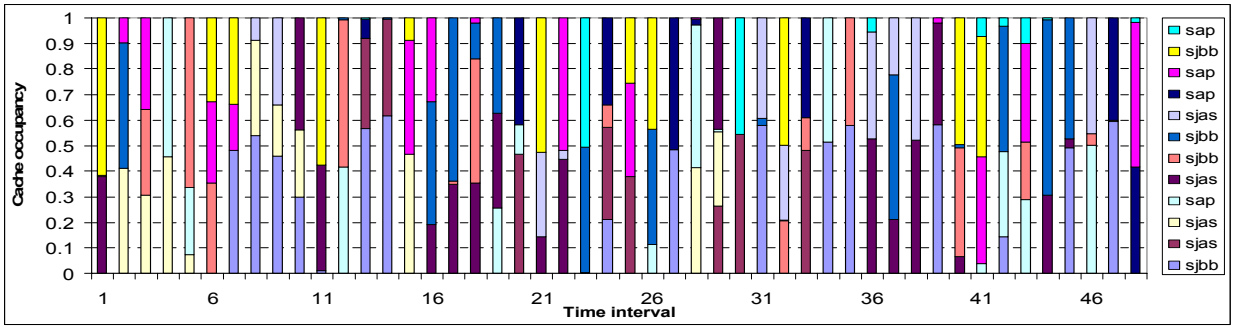


Figure 7. CacheScouts capacity profile for multi-VM server workloads (1 of 4 caches)

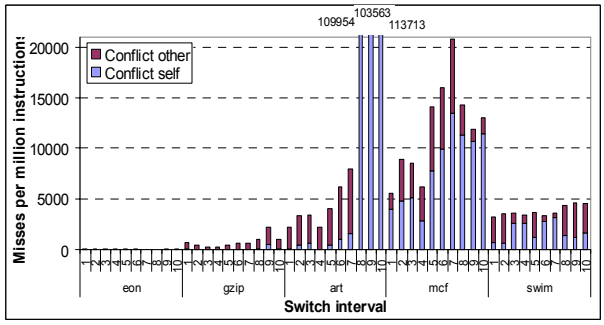


Figure 8. CacheScouts interference data for multi-programmed workloads (SPEC)

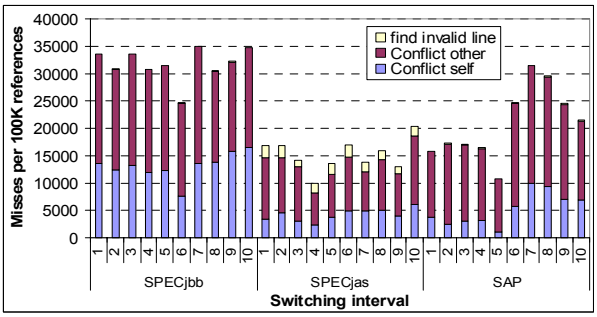


Figure 9. CacheScouts interference data for multi-VM workloads (Server)

4.5. CacheScouts Set Sampling Accuracy

When employing set sampling to reduce the overhead of monitoring, it is important to determine how many sets need to be sampled for reasonable accuracy. Figure 11 and 12 show the set sampling accuracy in

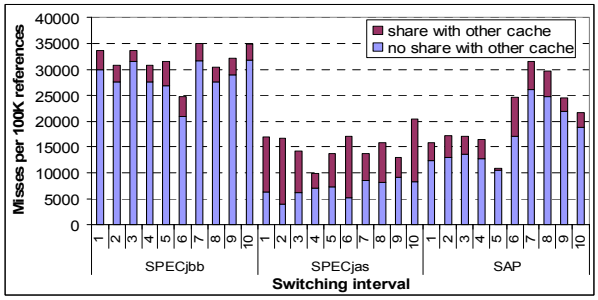


Figure 10. CacheScouts sharing data for multi-VM

reporting cache capacity information for the SPEC benchmarks and the server workloads respectively during their scheduling intervals. The data is reported in terms of average error rate (the % difference between full cache and set sampling with varying number of sets). As can be observed from the figure, the error rate in most cases reduces significantly as the number of sampled sets increases from 32 to 128 sets. It should be noted that a 4M cache with 64 byte lines and 16-way sets consists of 4096 sets. Employing 128 sampled sets out of 4096 is only 3.125%, which is much lower than the 10% that was conservatively used during the implementation discussion. For SPEC workloads, the average error rate is 6% or less when employing 128 sampled sets. It should also be noted that the error appears to be higher for workloads like eon because the fraction of cache capacity for these workloads is already very negligible. For server workloads (Figure 12), employing 128 sets provides an error rate that is 4% or lower. Since most of this data is

being used only for performance optimizations, even 90% accuracy (or less than 10% error) is expected to be more than sufficient.

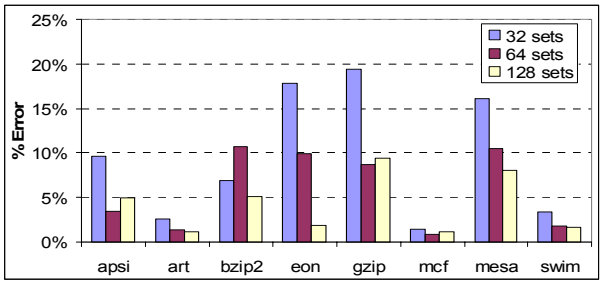


Figure 11. CacheScouts set sampling accuracy for SPEC workloads

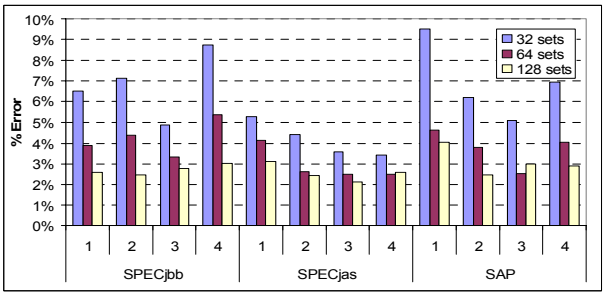


Figure 12. CacheScouts set sampling accuracy for server workloads

5. CacheScouts Monitoring Use Cases

We now look at some detailed monitoring case studies to show the value of CacheScouts monitoring feedback provided to the OS/VMM.

5.1. Simple Scheduling Optimizations

We start by first attempting two simple scheduling optimizations that require the use of CacheScouts. The first one is that of detecting sharing between threads and employing this knowledge to co-schedule sharing threads on the same cache. A CacheScouts-aware scheduler looks ahead in the queue and attempts to find a waiting task that has significant sharing with the other threads already running on other cores that share the same cache. The CacheScouts sharing matrix provides accurate information on sharing between applications and therefore this is used by the scheduler to perform this optimization. We experimented with this approach by implementing a CacheScouts-aware scheduler.

Figure 13 shows the resultant benefits as a function of the *lookahead length*. Note that a lookahead length of 0 is the base case with no optimizations. Using a lookahead length of 3 improves the miss rate of SPECjappserver by 20%, SAP by 18% and SPECjbb by a negligible amount. It should also be noted that the remote misses (misses that earlier found the data in a remote cache) are the ones that reduce significantly. The other misses stay roughly constant or increase moderately.

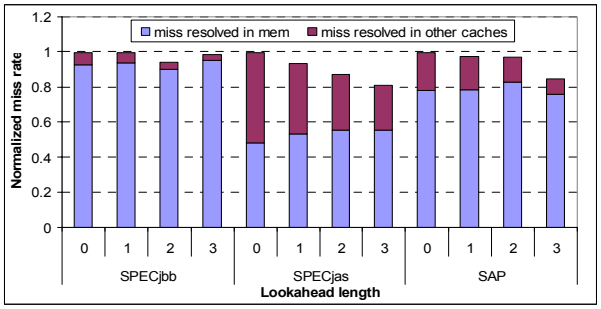


Figure 13. Benefits of sharing optimization for server workloads

We experimented with a capacity-based optimization for the SPEC workloads since we found that they use the cache resource very disparately. Here, the basic optimization is that of affinitizing threads to caches where the working sets are still remnant even after a few context switches. Typically, schedulers attempt to perform affinitization based on time elapsed between the previous context switch and the current re-schedule. However, CacheScouts provides accurate information on the working set of an application in the cache even after it is de-scheduled from the core. As a result, every time a scheduler attempts to find a task in the queue to schedule on to a core, it can look ahead deeper into the queue to find a task that still has the its working set left in the cache. By doing so, the task will not suffer a cold-cache effect and thereby improve in performance. Figure 14 shows the benefits of CacheScouts-based affinitization. From the chart, it can be seen that the application with the largest miss rate (*art*) improves by as much as 90%, which is offset by an increase in miss rate for the next dominant application (*mcf*). Overall, the average miss rate across all applications reduces by 40%, which is a significant improvement over the traditional scheduling algorithm.

We also performed another experiment by implementing CacheScouts in a full-system execution-driven simulator with timing information. We employed the capacity and interference counters available in CacheScouts to dynamically group applications by sorting them in capacity/interference order and co-scheduling one application from each end of the sorted list. This helps because each cache ends up with an application with a large working set (a very destructive application) as well as a small working set (a non-destructive application). We chose to run only four SPEC applications (*art*, *swim*, *crafty* and *eon*) simultaneously on the 4-core, 2-cache architecture

described earlier. Figure 15 shows the resultant benefits of performing this optimization. We observe performance benefits ranging from 7% to 26%. Thus the overall throughput of the platform is improved significantly by dynamic co-scheduling optimizations based on CacheScouts capacity/interference.

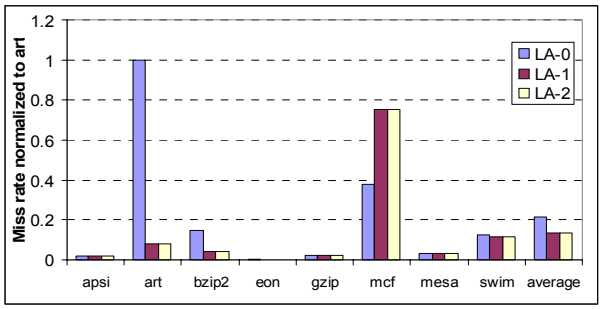


Figure 14. Benefits of capacity optimizations for SPEC workloads

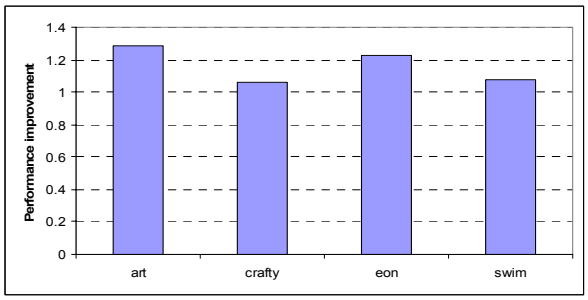


Figure 15. Capacity/Interference-Aware scheduling

5.2. Employing CacheScouts for QoS

In this subsection, we will investigate the use of CacheScouts to guide QoS and performance isolation mechanisms. QoS mechanisms have been proposed in previous literature [5][12] to control the distribution of cache resources to applications and virtual machines with varying user importance or priorities. The first step to enforcement is the knowledge of the resource utilization on a per application basis. CacheScouts enables this knowledge and allows the OS/VMM to provide cache allocation threshold hints to the platform. We set up a simple QoS experiment where we prioritize *art* over all other SPEC applications in the multi-programmed workload scenario. This is done by setting the priority of all applications except *art* to low priority and associating the low priority to limit its consumption to a 40% capacity threshold. This ensures that at least 60% of the cache is available to art at all times. The replacement policy in the cache is implemented by comparing the CacheScouts capacity counters to the threshold value of 40%. Figure 16 shows the benefit of employing CacheScouts for QoS.

We can see that the cache miss rate of *art* reduces by almost 40% at the potential expense of other application performance. For example, *mcf* miss rate increases by as much about 30% because it is constrained in cache space whenever it runs with *art*. Note that setting the limit to 40% for all other applications except *art* has the potential to affect their performance only when they are co-scheduled with *art*.

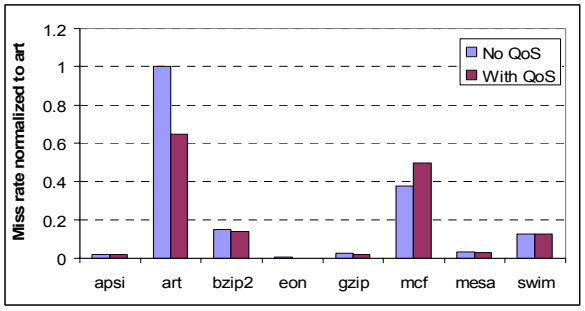


Figure 16. Capacity-Guided QoS Benefits

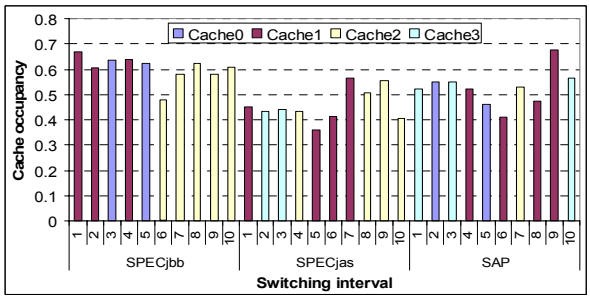


Figure 17. Metering cache usage for chargeback

5.3. Metering for ChargeBack

Today, OS/VMM provides sufficient hooks to monitor CPU utilization, memory capacity usage and disk space usage on a per application/user basis. As a result, a hosted server can charge the customers only based on these resource usage levels. However it is obvious that cache space available to an application also affects its performance significantly and is therefore a precious resource. Since CacheScouts provides cache resource usage information, it can be potentially used to charge back users.

Figure 17 shows the cache occupancy during the schedule time of one thread of each server benchmark. On average, SPECjbb occupies roughly 60% of cache space during its execution, SAP occupies roughly 50% of cache space and SPECjappserver occupies a little over 40% of cache space during its execution. Assuming that all of the applications spend an equal amount of time on the CPU, the cache resource chargeback component should exhibit a ratio of 6:5:4 ratio for SPECjbb, SAP and SPECjappserver.

6. Conclusions and Future Work

In this paper, we motivated the need for dynamic shared cache monitoring techniques in future CMP platforms. We proposed the CacheScouts architecture and implementation options. By employing software-guided monitoring IDs and set sampling, we showed that we can achieve accurate low-overhead, fine-grain cache monitoring data along the following vectors: (a) cache occupancy, (b) cache interference and (c) cache sharing. We then profiled the execution of two workload scenarios: (a) multi-programmed workloads using SPEC CPU2000 and (b) multi-VM workloads using multiple multi-threaded server benchmarks. We showed that the CacheScouts monitoring data is very useful in optimizing the scheduling policies and improving performance in these workload scenarios. So we believe the contributions in this paper will be valuable to architects as well as OS/VMM developers in the future. In the future, we plan to perform detailed evaluation of the use of CacheScouts monitoring for more workload scenarios and develop QoS-aware OS and VMM prototypes that can highlight its value. We expect similar monitoring evaluation that includes other shared resources will be valuable in the future.

7. References

[1] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting inter-thread cache contention on a chip multiprocessor architecture", 11th HPCA, 2005

[2] T. Deshane, D. Dimatos, et al., "Performance Isolation of a Misbehaving VMs with Xen, VMware and Solaris Containers,"http://people.clarkson.edu/~jnm/publications/isolationOfMisbehavingVMs.pdf

[3] A. Fedorova, M. Seltzer, M. Smith, C. Small, "CASC: A Cache-Aware Scheduling Algorithm For Multithreaded Chip Multiprocessors," http://research.sun.com/scalable/pubs/CASC.pdf

[4] L. Hsu, S. Reinhardt, et al., "Communist, Utilitarian, and Capitalist Cache Policies on CMPs: Caches as a Shared Resource", PACT, Sept 2006.

[5] H. Kannan, F. Guo, L. Zhao, et al., "From Chaos to QoS: Case Studies in CMP Resource Management," dasCMP/Micro, Dec 2006.

[6] R. Kessler, M.D. Hill, D. A. Wood, "A comparison of trace-sampling techniques for multi-megabyte caches", IEEE Transactions on Computers, 1994

[7] S. Kim, D. Chandra, and Y. Solihin, "Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture", 13th Int'l Conf. on Parallel Architectures & Complication Technique (PACT), 2004

[8] Intel Corporation. "Intel Dual-Core Processors," http://www.intel.com/technology/computing/dual-core/

[9] Intel Corporation, "World's first quad-core processors for desktop and mainstream processors," http://www.intel.com/quad-core/

[10] Intel Corporation, "Tera-Scale Computing," http://www.intel.com/research/platform/terascale/

[11] Intel Corporation, "Intel Vtune Performance Analyzer", http://www.intel.com/software/products/vtune

[12] R. Iyer, "CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms," 18th Annual International Conference on Supercomputing (ICS'04), July 2004.

[13] R. Iyer, "On Modeling and Analyzing Cache Hierarchies using CASPER," 11th MASCOTS, Oct. 2003.

[14] P. Kongetira, K. Aingaran, K. Olukotun, "Niagara: A 32-Way Multithreaded Sparc Processor," IEEE Micro. 2005

[15] K. Krewell, "Best Servers of 2004: Where Multicore is Norm," www.mpronline.com, Jan 2005.

[16] C. Liu, A. Sivasubramaniam, M. Kandemir, "Organizing the Last Line of Defense before Hitting the Memory Wall for CMPs," 10th IEEE Symp.on High-Performance Computer Architecture, Feb. 2004.

[17] Oprofile, http://oprofile.sourceforge.net/docs/

[18] S.S.Pinter, M.Zalmanovici, "Data Sharing Conscious Scheduling for Multi-threaded Applications on SMP Machines," Euro-Par 2006

[19] M. K. Qureshi and Yale N. Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches", MICRO 2006

[20] N. Rafique, W. Lim, M. Thottethodi, "Architecture Support for OS-Driven CMP Cache Management", 15th International Conference on Parallel Architectures and Compilation Techniques, Sept 2006.

[21] Sap America Inc., "SAP Standard Benchmarks," http://www.sap.com/solutions/benchmark/index.epx

[22] SPECjAppServer Java Application Server Benchmark, available online at http://www.spec.org/jAppServer/

[23] SPECint, http://www.spec.org/cpu2000/SPECint

[24] SPECjbb2005, http://www.spec.org/jbb2005

[25] G. Suh, S. Devadas, and L. Rudolph, "A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning," HPCA-8, Feb 2002.

[26] N. Thornock, "Using Set Sampling for L3 cache studies," Master's Thesis, Dept. of Computer Science, Brigham Young University, 1999

[27] J. Torrellas, A. Tucker and A. Gupta, "Benefits of cache-affinity scheduling in shared-memory multiprocessors", SIGMETRICS 1993

[28] Hsian-Fen Tsao, "IBM @eserver p5 570 Server Consolidation Using POWER5", Virtualization White Paper, IBM Corporation

[29] R. Uhlig, et al., "Intel Virtualization Technology," IEEE Computer, 2005.