From Chaos to QoS: Case Studies in CMP Resource Management

Hari Kannan[!], Fei Guo[$], Li Zhao[*], Ramesh Illikkal[*], Ravi Iyer[*], Don Newell[*], Yan Solihin[$], Christos Kozyrakis[!]
[!] Stanford University, Stanford, CA, USA
[$] North Carolina State University, Raleigh, NC, USA
[*] Intel Corporation, Hillsboro, OR, USA

Abstract

As more and more cores are enabled on the die of future CMP platforms, we expect that several diverse workloads will run simultaneously on the platform. A key example of this trend is the growth of virtualization usage models. When multiple virtual machines or applications or threads run simultaneously, the quality of service (QoS) that the platform provides to each individual thread is non-deterministic today. This occurs because the simultaneously running threads place very different demands on the shared resources (cache space, memory bandwidth, etc) in the platform and in most cases contend with each other. In this paper, we first present case studies that show how this results in non-deterministic performance. Unlike the compute resources managed through scheduling, platform resource allocation to individual threads cannot be controlled today. In order to provide better determinism and QoS, we then examine resource management mechanisms and present QoS-aware architectures and execution environments. The main contribution of this paper is the architecture feasibility analysis through prototypes that allow experimentation with QoS-Aware execution environments and architectural resources. We describe these QoS prototypes and then present preliminary case studies of multi-tasking and virtualization usage models sharing one critical CMP resource (last-level cache). We then demonstrate how proper management of the cache resource can provide service differentiation and deterministic performance behavior when running disparate workloads in future CMP platforms.

1. INTRODUCTION

As the momentum behind on-chip multiprocessor (CMP) architectures [7][12][13] continues to grow, it is expected that future client and server microprocessors will have several cores sharing the on-die and off-die resources. The success of CMP platforms depends not only on the number of cores but also heavily on the platform resources (cache, memory, etc) available and their efficient usage. Traditionally, processor and platform architectures are designed to perform well when a single parallel application is running on them. However, with the evolving software use models, CMP platforms will also be used to run multiple applications simultaneously in both client and server domains. The rapid deployment of virtualization [2][6][15][17] as a means to consolidate multiple applications on to a platform is a prime example.

When multiple applications run simultaneously on CMP architectures, the quality of service (QoS) that the platform provides to each individual application will be non-deterministic (*or chaotic*) because it depends heavily on the behavior of the other simultaneously running workloads. As expected, recent studies [3][5][9][10][14] have indicated that contention for critical platform resources (e.g. cache) is the primary cause for this lack of determinism and QoS. In this paper, we highlight this problem further and motivate the need for QoS support in CMP platforms. We focus on one of the important CMP platform resources (last-level cache space) and show case studies describing the effect of sharing this resource in multi-tasking and virtualization scenarios. Based on these observations, we investigate QoS policies and mechanisms to efficiently manage these shared resources in the presence of disparate applications (or threads).

Recent studies on (cache) resource management have either advocated the need for fair distribution [3] between threads and applications or the need for unfair distribution [5] with the purpose of improving overall system performance. In contrast, the work presented here aims to improve the performance of an individual application at the cost of the potential detriment of others with guidance from the operating environment. This is motivated by usage models such as server consolidation where service level agreements motivate the degree of performance differentiation [1][4] desired for some applications. Since the relative importance of the deployed applications is best managed by the operating software environment, we experiment with software-guided priorities (e.g. assigned by server administrators) to efficiently manage hardware resources. We compare the use of software-guided priorities (QoS-aware caches) against dedicated runs (isolated cache use), shared runs (unmanaged shared caches) as well as fair caches (equal private cache partitions per application).

In order to experiment with QoS, we have developed two QoS software prototypes (QoS-aware Linux as well as QoS-aware Xen). The primary contribution of this paper is the description of these prototypes as well as the case studies performed using them. The case studies include multi-tasking usage scenarios as well as virtualized usage scenarios and show the trade-offs between four resource management approaches (dedicated, shared, fair and QoS) on CMP platforms. We show that enabling and enforcing

software-guided priorities is much more effective in providing QoS as compared to traditional approaches like static partitioning to CMP cache management.

2. BACKGROUND AND CHALLENGES

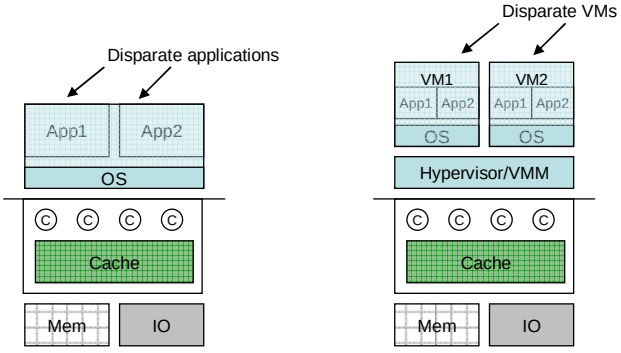
In this section, we highlight the motivation behind QoS by describing disparate threads of execution on CMP platforms and the shared resource problem.

2.1. Disparate Threads of Execution

The key trends in software that promote the use of disparate threads of execution (Figure 1) on CMP platforms are as follows:

(a) ***Multi-tasking becoming more common:*** As more threads and cores are enabled on the processor die, the compute capability is best utilized by multiple simultaneously executing tasks or applications (see Figure 1a). The behavior and platform resource usage of these simultaneous threads of execution will be quite disparate in nature (e.g. one is cache-friendly and the other is streaming in nature). At the same time, it is possible that one application is of much more importance than the other (e.g main application execution running along with a background streaming activity like network backup).

(b) Virtualized workloads becoming mainstream: While the concept of virtualization [6] has been around for a while, the recent re-emergence of virtualization as a means to consolidate workloads in the datacenter exposes the critical need to pay attention to the performance behavior of heterogeneous virtual machines running simultaneously on a server. This becomes even more important as virtualization-based usage models continue to rapidly evolve and encompass office workstations/desktops and even home PCs/laptops. In these scenarios, many disparate workloads are consolidated together and hardware performance isolation [4] becomes a desired feature for the high priority applications that can be identified by the user or system administrator.

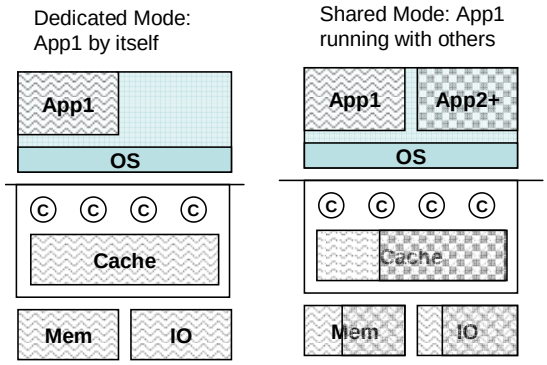


(a) Multi-Tasking (b) Virtualization/Consolidation

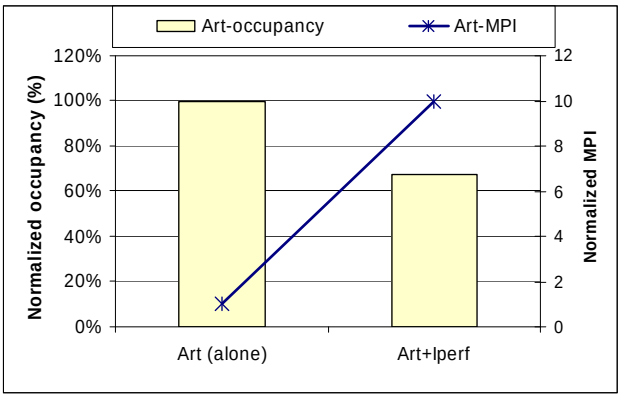
Figure 1. Disparate Threads on CMP Platforms

2.2. Example Resource Sharing Impact

In this section, we present an example of resource sharing impact based on trace-driven simulation of cache sharing in a CMP platform.



(a) Illustrating Dedicated and Shared Usage



(b) Performance Impact of Resource Sharing

Figure 2. Implications of Resource Sharing

Figure 2 illustrates the motivation for QoS and resource management. The figures show the resource and performance implications of a high priority application running in standalone (dedicated) mode versus when it is running in shared mode with a low priority application. We chose Art - a SPEC 2000 benchmark [16], to represent a high priority application, and Iperf (a network benchmark) to represent the low priority application. These experiments were conducted in a virtualized environment with each application running in a different virtual machine. This simulation study (Figure 2b) shows how sharing the cache uniformly between applications affects the performance of the high priority application (Art). The Y axis on the left depicts the cache occupancy of Art relative to the case when it runs alone. The Y axis on the right plots the MPI (Misses per Instruction) of Art, also relative to the case when it runs alone.

Figure 2b shows that the high priority application's MPI increases by 9.9X when co-scheduled with Iperf. Iperf exhibits very poor cache locality and effectively thrashes the cache, which accounts for the observed degradation in

Art's performance. In order to minimize the effects of resource contention, the priority of the application needs to be comprehended by the platform in order for it to allocate hardware resources (in this case, cache space) accordingly. In this paper, we experiment with QoS policies and mechanisms to manage the cache resource distribution between high and low priority applications. It should be noted that the need for resource management stems both from the need for providing better QoS to high priority applications as well as the need for better determinism in the platform.

3. FROM CHAOS TO QOS

In this section, we introduce the QoS policies for resource management, and the architectural modifications necessary to support these policies.

3.1. Resource Management & QoS Policies

Most modern processors do not enforce any QoS in the cache. In these systems, the amount of cache space consumed by each thread depends upon both its memory footprint and its memory accesses patterns. This leads the cache utilization to be determined solely by the individual application's demand. This has been described as "capitalistic" in a previous study [5].

We classify QoS policies into three categories: utilitarian, fair and elitist. The utilitarian policy attempts to improve the overall throughput (the greater good) of the platform by maximizing resources for the cache-friendly application and minimizing resources for the cache-unfriendly application. The fair policy attempts to equalize the cache resources provided to each of the individual applications or provide unequal cache resources to ensure equal performance degradation for each of the applications [11]. The elitist policy considers the relative priority of the applications running simultaneously and ensures that a high priority application is provided more platform resources than the low priority applications. In other words, this policy caters to the elite application(s) at the possible expense of the non-elite classes of applications.

In this paper, we consider elitist policies for resource management and compare them to fair resource management (equal resources) as well as no resource management. This study focuses on performing resource management in the last level cache. We experiment with both static and dynamic mechanisms for achieving elitist QoS.

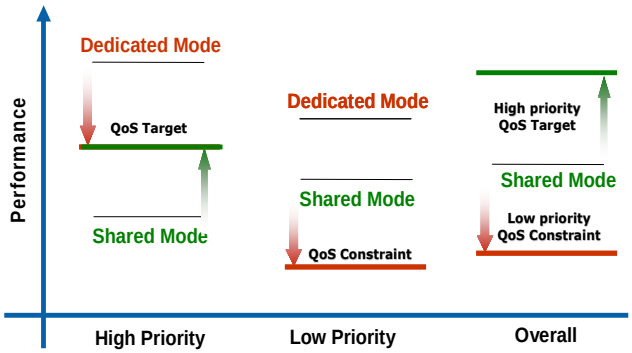


Figure 3. Platform QoS Policies

Figure 3 illustrates the impact of QoS mechanisms on both the applications, and the whole system. Static and dynamic QoS policies differ in the way the target performance and resource constraints are specified. Static policies are defined if the specified target performance does not require continuous adjustment of resources. This policy defines the QoS target and constraint in terms of the resource usage metric (e.g. cache space) provided to the high priority application and low priority applications respectively This requires mechanisms to statically distribute resources, but not dynamically alter them based on resultant performance. For the cache resource, it is intuitive that we consider space as the primary metric.

Dynamic QoS requires resource allocation to be continuously monitored based on the observed performance and the targets/constraints. The targets and resource constraints are specified in terms of process' miss rates and cache space allocation respectively. The dynamic policy we implemented involved varying both the resources provided to different processes and the targets assigned to them based on the current phase of the application. Both applications start out sharing all the available resources without any constraints. The high priority process' initial target is set to be half its observed miss rate and the low process' degradation threshold to be twice its observed miss rate. The adjustment of targets and constraints is performed every 50 million instructions. If the high priority application's miss rate is lower than the target, and the degradation threshold for the low priority application is not exceeded, then the percentage of resources assigned to the low priority application is decreased by a factor of two. If the low priority application's performance degrades beyond its target miss rate, the percentage of resources assigned to it is increased by 10%. The targets for the applications are made more aggressive (decrease the target miss rate by 10%) or conservative (increase the target miss rate by 10%) depending upon whether they are met or not.

3.2. QoS-Aware Platform Architecture

The elitist policy requires that threads or applications be classified into various priorities and architectural support

be added to control the hardware resource consumption based on the classification. Figure 4 illustrates the architecture that meets these requirements. As shown in the figure, applications are assigned various priorities by the administrator. Based on the priorities, the QoS resource table (QRT) is updated with cache space limitation data for each priority level. When threads are scheduled to run, their memory requests are tagged with priority levels and cache space limitation information. The cache subsystem is modified to maintain the priority level associated with each line. There are counters that monitor cache space used by each priority level. These counters are used by the replacement policy to evict the appropriate lines within the set. For example, when the low priority resource usage limit is reached, it requires that a low priority victim be chosen from the cache set. Similarly, the high priority process should preferentially replace the low priority process' entries since we our goal is an elitist policy. We accomplished this by modifying the LRU policy to first identify low priority lines and pick the victim among these lines for replacement, in the abovementioned scenarios.

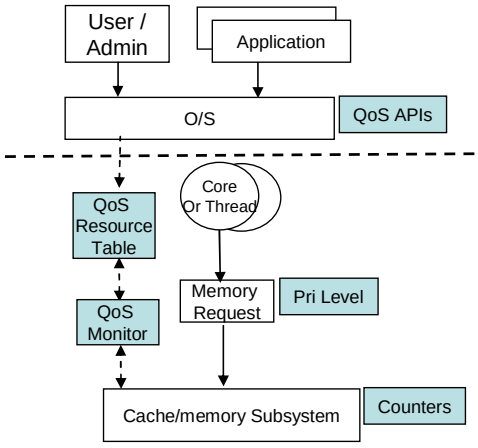


Figure 4. Architecture support for QoS

In order to enable users or administrators to specify the priorities of the application/thread/process, we require QoS support in the execution environment (OS or virtual machine monitor). In section 4, we describe in more detail the architectural support required to provide QoS, and introduce prototypes that mimic this support.

4. QoS Prototypes for Experimentation

In this section, we describe the QoS support required in systems software (OS and VMM) and present prototypes that we developed to emulate this support.

4.1. QoS-Enabled OS

The OS based platform QoS prototype is built on the Linux operating system. In order to provide QoS support, we made several modifications and additions on the 2.6.16 Linux kernel on a Fedora Core 5 host machine. These changes are illustrated in Figure 5.

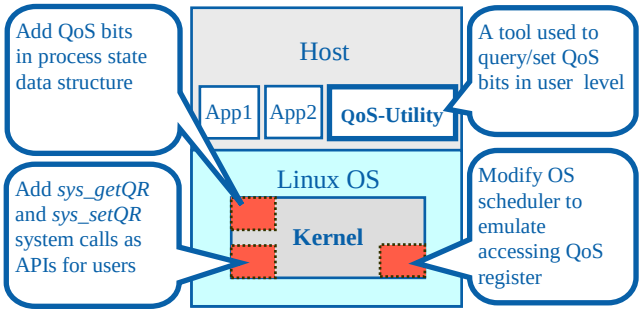


Figure 5. Illustration of QoS-Enabled Linux

The QoS software prototype implementation comprises of these major components:

(a) ***QoS bits in process state:*** QoS bits that indicate the priority level and associated information were added to each process' state. This information is saved and restored during context switches.

(b) ***QoS register emulation:*** The Linux scheduler was modified to emulate saving and restoring the QoS bits from the process state and to commit it to processor architecture state (QoS register). This was achieved by employing a special I/O instruction during every processor context switch. More specifically, we first read the QoS bits value from the process context that was switched in. Then we issued an *out* (x86 ISA) instruction that sent this value to an unused I/O port 0x200(typically, this port was reserved for joystick). This instruction was used to communicate the process' QoS value to the hardware. Port 0x200 was registered as the "QoS" port in the kernel I/O resource registration module to guarantee that it would not be used by other I/O devices.

In addition, to allow administrators to manage QoS values associated with running processes, the Linux kernel was modified to provide:

(a) ***QoS APIs for user/administrator:*** Two extra system calls were added to the Linux kernel to provide interfaces for programs to access the QoS bits which were stored in kernel address space.

(b) ***QoS utility program:*** This tool was implemented in the host Linux machine to query and modify the QoS value of the running applications.

4.2. QoS-Enabled VMM

Our virtualization software prototype is similar to the OS prototype, except the enhancements were made to the hypervisor instead of the operating system. This involved

making the following modifications to a popular hypervisor, Xen 3.0.2:

(a) ***QoS at virtual domain level***: QoS bits that indicate the priority level and associated information were added to each domain's (virtual machine's) state. This information is saved and restored during context switches.

(b) ***QoS register emulation***: The Xen scheduler was modified to emulate setting the processor architecture state (QoS register) based on the priority of the domain. We used a setup similar to that described in Section 4.1 to expose this register to the underlying hardware.

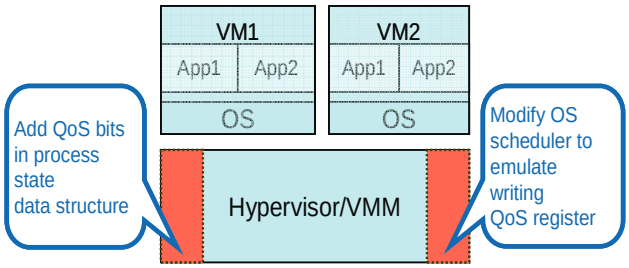


Figure 6. Illustration of QoS-Enabled Xen

With this infrastructure, it is possible to make the cache subsystem cognizant of the priority of the virtual machine it is servicing.

4.3. Experimental Framework

In order to evaluate proposed OS/VMM prototypes, we setup the experimental framework shown in Figure 7.

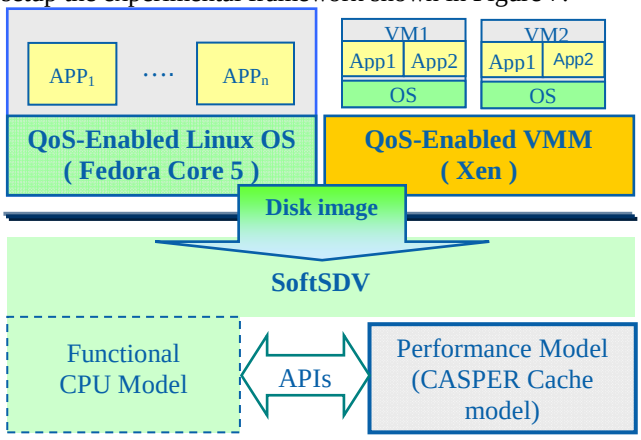


Figure 7. Simulation framework for QoS aware OS/VMM

We employed SoftSDV [18], a full system simulator that allowed us to functionally model the architecture and provided an interface for enabling performance models. We used the functional model of SoftSDV to boot a Fedora Core 5 Linux disk image for a QoS-enabled OS simulation. A disk image of Xen 3.0.2 running virtual machines with Suse 9.1 Linux was used for QoS-Enabled VMM simulation. SoftSDV provided us with APIs which we used to pass the instructions executed by the running applications or VMs from our functional model to the performance model. These instructions included the special *out* instructions mentioned in Section 4.1, which triggered the performance model to simulate the ensuing memory accesses with the specified priority. We integrated CASPER [8] - a functional cache simulator which was modified to support the cache QoS policies described in Section 3.3, into this experimental framework, for evaluating cache performance.

Parameters	**Values**
Core Number	1/2/4
L1 (Private)	Unified, non-inclusive, 32KB, 16 way, 64B Block, LRU
L2 (Shared)	Unified, 512/1024/2048/4096/8192 KB, 16 way, 64B Block

Table 1. Cache Simulation Parameters

Table 1 summarizes the parameters for our experiments. We evaluated the QoS-Enabled OS prototype using a few applications from the SPEC 2000 benchmark suite [16] - Ammp, gcc, art, applu and mcf, which show large cache sharing impact when they are co-scheduled. The standard *ref* input sets were used with these benchmarks. The QoS-Enabled VMM prototype was evaluated using Iperf (a networking benchmark) and SPEC 2000 benchmarks – art, swim, mesa and bzip2. In all experiments, we collected the cache sharing statistics for billion instructions executed by the system.

5. Preliminary Results and Analysis

In this section, we describe the experimental results for our QoS-Aware Linux and QoS-Aware Xen software prototypes.

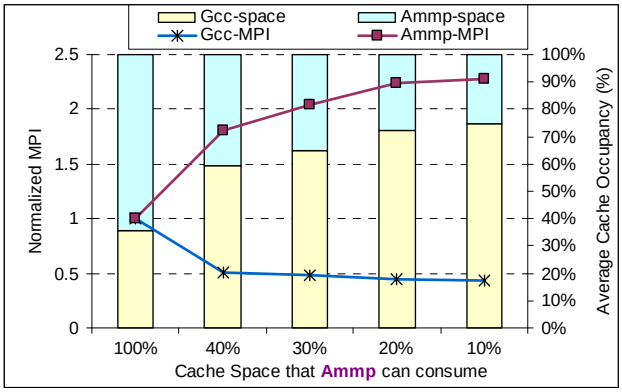
5.1. Multi-Tasking Experiments on QoS-Aware Linux

In the multi-tasking experiments, we assumed the execution platform to be two/four core CMP architecture. Two or four applications run simultaneously and share the L2 cache.

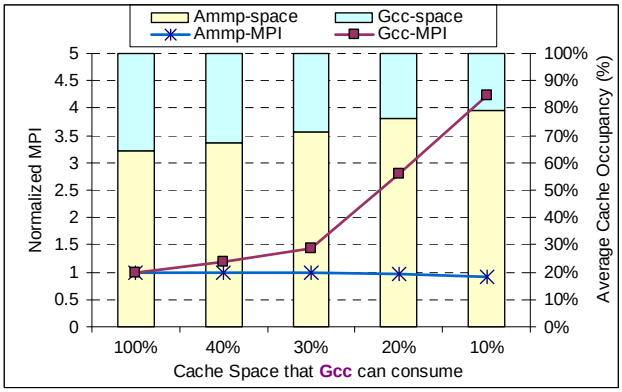
5.1.1 Two-Core CMP Scenario

Our evaluation of the static QoS policy on two-core CMP is done by varying the cache space limit for low priority applications from 10% to 40% of the cache. We compare this to the shared mode execution without QoS, which is denoted by a cache space limit of 100%. We use the resource performance metric to evaluate effectiveness in terms of the L2 cache's MPI (misses per instruction).

Figure 8 shows the impact of QoS policy when Ammp and Gcc are running simultaneously (on separate cores) and share a 512KB cache.



(a) Gcc(hi) + Ammp(lo)



(b) Ammp(hi)+Gcc(lo)

Figure 8. Impact of QoS in two-core CMP scenario

The two Y-axes represent the MPI (lines) and average cache space occupancy (bars) of the co-scheduled applications respectively. The MPI value is normalized to the case when both applications share the L2 cache without any prioritization. Figure 8a illustrates the scenario when Gcc has a higher priority than Ammp. As expected, the MPI of Gcc reduces when we reduce the cache space available for Ammp. This is accompanied by an increase in the the MPI of Ammp. The MPI reduction was seen to be as much as 57% when Ammp was constrained to occupy 10% of total cache size. A noteworthy finding was that Gcc benefited more from cache Qos than Ammp, even though Ammp occupied more cache space than Gcc in the base case with no QoS (around 65%). This conclusion was based on the fact that Ammp's MPI increased by only around 125% when it lost about 40% of total cache space (Figure 8a), while the MPI increase of Gcc was around 320% when it lost about 15% of total cache space (Figure 8b). This explains why reducing the cache space available for Gcc did not cause significant MPI reduction for Ammp (Figure 8b).

Note that although we limit the cache space for low priority application, this limitation is not a hard bound and applications can sometimes exceed the specified limits. This is because of a couple of reasons. One, sharing of data by applications, (shared libraries etc) results in processes sometimes accessing data tagged with the priorities of other processes. In our implementation, cache lines are tagged with the priority of the last application that touches the data. This accounts for the calculated low priority application occupancy potentially being larger than the prescribed space limit. Secondly, cache locality dictates that we need not always have a replacement candidate with the same priority present in every set. Since we try and constrain total cache occupancy and not the occupancy per set, such situations lead to the low priority process potentially exceeding the bounds set for it.

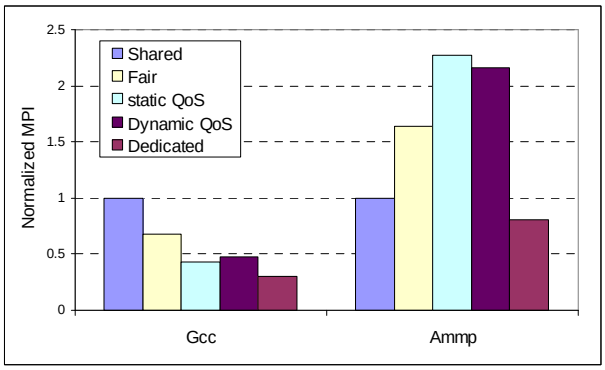


Figure 9. Performance of Gcc and Ammp running simultaneously with different execution modes

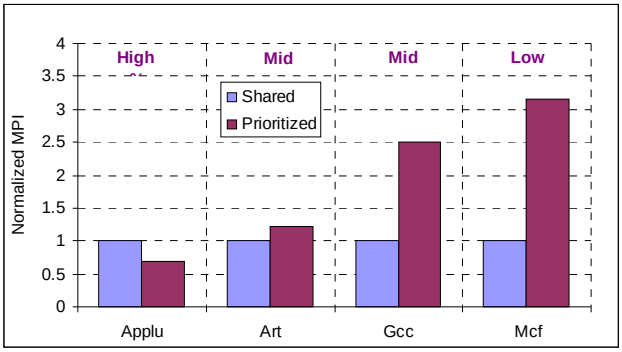
Figure 9 illustrates the MPI of Gcc and Ammp under shared mode (without prioritization), fair mode (each application occupy half of the cache), static QoS mode (ammp is low priority and is constrained to occupy 10% of the cache), dynamic QoS mode (ammp is low priority and the amount of cache it occupies is dynamically modified) and dedicated mode (applications occupy the whole cache). The MPI value of each application is normalized to the case when it runs under shared mode. Figure 9 shows that both static and dynamic QoS are more efficient policies to approach the performance improvement bound (dedicated mode) than fair QoS. For applications like Ammp that occupy large percentages of cache space in the shared mode, it is not surprising that fair QoS adversely affects performance.

5.1.2 Four-Core CMP Scenario

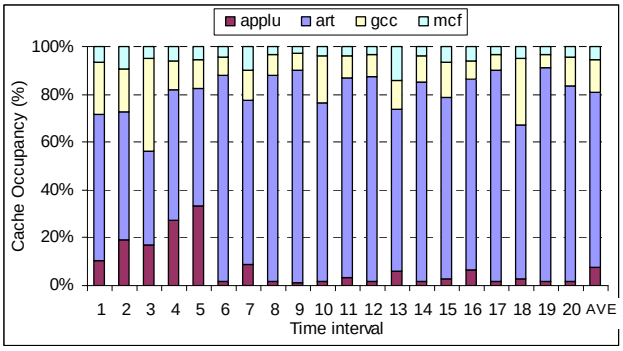
In the four-core CMP scenario, we assume that one high priority application, two mid level priority applications and one low priority application run simultaneously and share a 1MB cache. Under the prioritized running mode, the middle priority application is limited to occupy 10% of total cache space and the low priority application will bypass the L2 cache (i.e. 0%).

Figure 10 shows the impact of QoS when Applu (high priority), Art (mid level priority), Gcc (mid level priority) and Mcf (low priority) are co-scheduled. In Figure 10a, the MPI value of each application is normalized to the case when it shares the cache with other applications without prioritization. Figures 10b and 10c show the cache

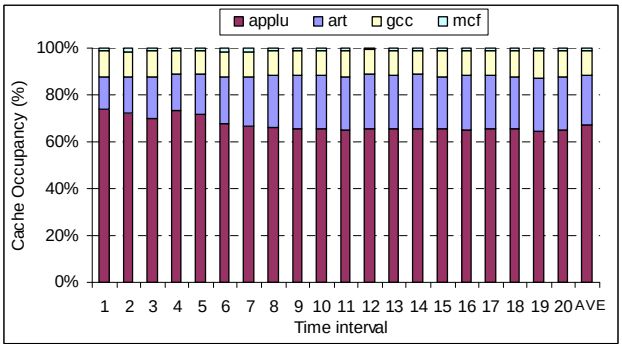
occupancies of four applications without and with cache QoS respectively. When we limit the cache space of Art and Gcc and bypass the cache accesses from Mcf, the MPI of Applu reduces by 33%. As a result, the MPI of Art, Gcc and Mcf will increase by 21%, 150% and 216% respectively. Figure 10b and Figure 10c clearly show that employing cache QoS can efficiently assign a deterministic amount of cache space to the high priority application. The cache occupancy of Applu increases from 7% to 67% on average with prioritization. Mcf's occupancy is slightly larger than 0% and Art's occupancy is around 30% of the cache, due to shared Linux libraries as explained in section 5.1.1



(a)Performance



(b) No Cache QoS



(c) With Cache QoS

Figure 10. Impact of QoS in four-core CMP scenario

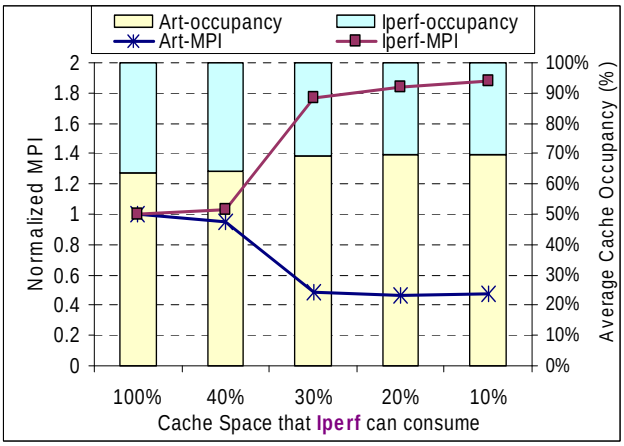
5.2. Virtualization Experiments on QoS-Aware Xen

We ran our experiments on a uni-processor with split Level-1 instruction and data caches (each 32 KB) and a unified Level-2 cache. Only the Level-2 cache was made QoS-aware. Our experiments involved running two benchmarks, in different virtual machines. One of the virtual machines was given a higher priority, allowing it to use the whole cache, while the other had a lower priority and was constrained to use only a portion of the cache. The cache size was varied between 1MB and 8MB to mimic large server workloads. We also varied the amount of cache allocated to lower priority virtual machines between 10% and 40% of the cache. We ran three sets of benchmarks which are described in this section:

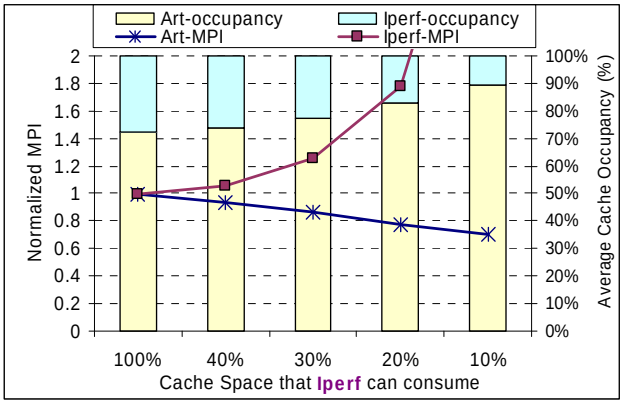
5.2.1 Co-scheduling Compute intensive and I/O intensive workloads

Compute intensive tasks typically have more uniform cache access patterns and are far more cache friendly than I/O intensive applications. Constraining the amount of cache available to the I/O application should help improve the MPI of the compute intensive process, when they are co-scheduled, since I/O applications cause an increase in the number of conflict misses, effectively thrashing the cache.

For the purpose of this study we ran Iperf and Art in a virtualized environment. We ran Iperf in Xen's Domain 0 (along with the I/O VM) and assigned it a low priority. We enabled Xen's native scheduler which allows Domain 0 to use as much as 75% of the CPU. Art was run in a different virtual machine (Domain 1) which was allowed to access the whole cache, and was scheduled for up to 25% of the CPU time. The two benchmarks were co-scheduled for a billion instructions varying both the amount of cache at Iperf's disposal and the total size of the cache.



(a) MPIs of Art and Iperf sharing a 4MB cache



(b) MPIs of Art and Iperf sharing a 2MB cache

Figure 11. Impact of QoS while co-scheduling Art and Iperf

Figure 11 shows the static QoS evaluation results from our QoS prototype experiment with an I/O application (Iperf) and a computation application (Art) sharing a last level cache. These plots also show the percentage of cache occupied by Art and Iperf at different cache sizes. When running Art (high priority) and Iperf (low priority) with a 4MB level-two cache (Figure 11a), we find that placing a limit on the cache space that the lower priority application can occupy significantly decreases the MPI of the higher priority application (reduced to around 40%). This MPI reduction is corroborated by the cache occupancy graph which shows Art's occupancy stabilizing when Iperf is limited to 30% of the 4MB cache. The reduction in Art's MPI is more than the increase in that of Iperf, indicating that resource management in this situation not only has a positive influence over the high priority application, but also helps lower the net MPI of the system (Art and Iperf running together). Another point to note from the graphs is that the occupancies of the low priority applications aren't strictly bound by the upper limits we set. As explained in section 5.1.1, this is due to sharing of VMM entries (which inherit the priorities of the processes they are invoked from) and maintaining global cache occupancy counters.

When the cache size is less than 2MB (Figure 11b), we see that the percentage occupancy of Art increases monotonically to use most of the cache. This indicates that the cache is too small to fit Art's working set completely. Constraining the amount of cache accessible by Iperf has a positive impact on Art's MPI at the cost of Iperf cache performance. The overall MPI of the system increases significantly when Iperf is constrained to run with less than 10% of the 2MB cache, indicating that resource management of small caches could adversely affect the system's performance if the lower priority application is not allocated a minimum amount of cache space.

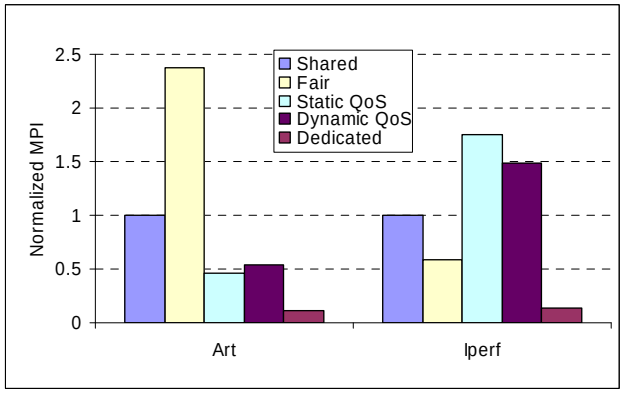


Figure 12. Performance of Art and Iperf running simultaneously with a 4MB cache with different execution modes

Figure 12 illustrates the MPI of Art and Iperf running in shared mode (without prioritization), fair mode (each application occupies half of the cache), static QoS mode (Iperf is assigned a low priority and occupies 10% of the cache), dynamic QoS mode (Iperf is assigned a low priority and the amount of cache it occupies is dynamically modified) and dedicated mode (both applications have dedicated 4MB caches). The MPI value of each application is normalized to the case when it runs in shared mode. Figure 12 shows that both static and dynamic QoS are more efficient policies to approach the maximum performance improvement point (dedicated mode) than fair QoS. In addition, dynamic QoS can be tuned to provide better performance for the platform in general and not just one application.

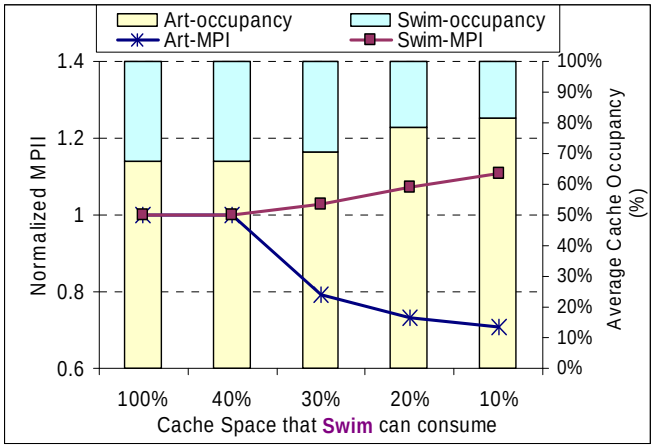
5.2.2 Co-scheduling two compute intensive workloads

In this study, we ran Swim and Art, two SPEC2000 workloads that have very different cache scaling patterns. Swim's MPI (misses per instruction) remains fairly constant as the cache size changes while Art's MPI falls rapidly as the size of the cache increases. We constrained the amount of cache the domain running Swim could use, while allowing the domain running Art access to the whole of the cache. We modified Xen to schedule both domains for equal time slices (while using the default period of the SEDF scheduler).

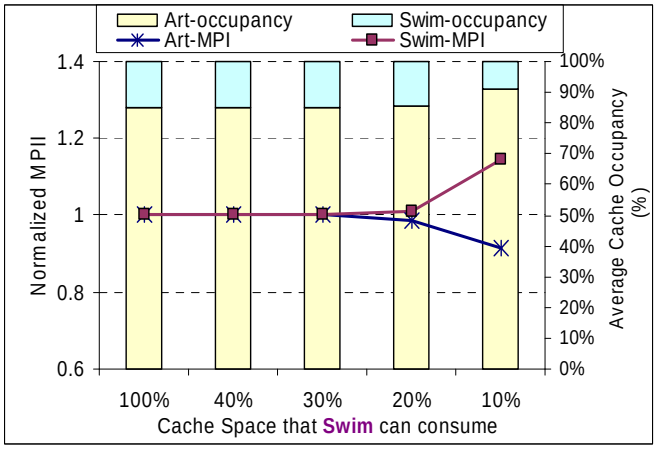
Figure 13 shows the static QoS evaluation results. These plots also show the percentage of cache occupied by Art and Swim at different cache configurations. While running Art (high priority) and Swim (low priority) with an 8MB level-two cache (Figure 13a), we find that placing a limit on the cache space that the lower priority application can occupy significantly decreases the MPI of the higher priority application (reduced to less than 70%). Cache management in this situation not only has a positive influence over the high priority application, but it also helps lower the net MPI of the system (Art and Swim running together).

At 4MB (Figure 13b), we see that Art occupies most of the cache, even in the shared case. This high and almost constant cache occupancy graph explains the lack of a radical reduction in Art's MPI at this cache configuration when Swim's cache space is changed. For cache sizes smaller than 4MB, the performance boost for Art is small due to the fact that the cache size is too small to accommodate the working sets of both Art and Swim.

These studies indicate that enabling cache-priorities at the granularity of virtual machines definitely decreases the amount of contention in the cache and allows the higher priority applications to be assured a much better quality of service. They also suggest that even overall system performance could receive a boost at certain hardware configurations, which are dependent upon the workloads in question.



(a) MPIs of Art and Swim sharing an 8MB cache



(b) MPIs of Art and Swim sharing a 4MB cache

Figure 13. Impact of QoS while co-scheduling Art and Swim

Figure 14 illustrates the MPI of Art and Swim running in shared mode (without prioritization), fair mode (each application occupies half of the cache), static QoS mode (Swim is assigned a low priority and occupies 10% of the cache), dynamic QoS mode (Swim is assigned a low priority and the amount of cache it occupies is dynamically modified) and dedicated mode (both applications have dedicated 8MB caches). The MPI value of each application is normalized to the case when it runs in shared mode. The figure shows that both static and dynamic QoS are more efficient policies to approach the maximum performance improvement point (dedicated mode) than fair QoS. This figure also illustrates the uniform behavior of Swim across the different modes, due to which static and dynamic QoS actually provide a performance boost to not only Art but the whole system.

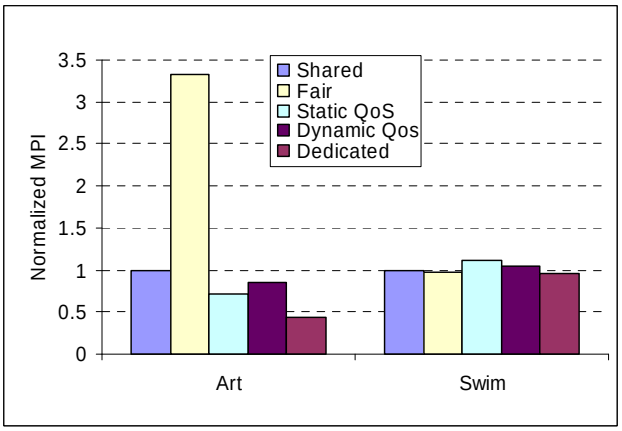


Figure 14. Performance of Art and Swim running simultaneously with a 8MB cache with different execution modes

5.2.3 Four-VM CMP Scenario

In this study, we scaled the number of virtual machines to four, with each of them running a different application, to gauge the effectiveness of our scheme in larger CMP environments. In the four-VM CMP scenario, we assume one high priority application, two mid-level priority applications and one low priority application are running simultaneously and share a 4MB cache. Under the prioritized running mode, the mid-level priority applications are limited to occupy 20% of the total cache space and the low priority application is limited to occupy 10% of the total cache space. All these applications were run with equal scheduling priority.

Figure 15 shows the impact of QoS when Art (high priority), Mesa (middle priority), Bzip2 (middle priority) and Iperf (low priority) are co-scheduled. Each of these applications run in different virtual machines which share the physical resources available to one core. In the Figure, the MPI value of each application is normalized to the case when it shares the cache with other applications without prioritization. As is evident from the figure, limiting the amount of space available to the other applications results in a 45% reduction in miss rate for Art. Mesa also surprisingly benefits from QoS which is perhaps because of the decreased interference from Iperf. The performance degradation for the other mid-level priority process - Bzip2, is minimal. Iperf's performance sees a degradation

which is consistent with its getting allotted just 10% of the cache.

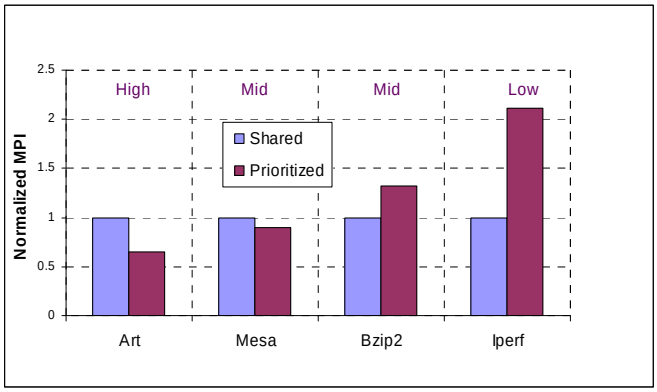


Figure 15. Impact of QoS in four-VM CMP scenario

6. SUMMARY AND FUTURE WORK

In this paper, we presented the motivation behind QoS-aware platforms and execution environments by showing case studies of CMP cache resource usage. We showed that it is important to provide better determinism in the platform especially in the context of multi-tasking and virtualization. By enabling QoS mechanisms in the hardware as well as exposing it to the systems software, we can address this requirement. We described two software prototypes (QoS-aware Linux and QoS-aware Xen) that allow experimentation with multi-tasking and virtualization usage scenarios. The preliminary case studies with these usage scenarios showed that QoS-enabled caches work better than unmanaged caches and fair caches. We also show that enabling QoS provides better determinism to the high priority application by bringing its performance closer to that of when the application is running by itself.

In future, we hope to experiment more with dynamic QoS mechanisms that improve the performance of not just one application, but the whole platform. We also plan to experiment with QoS in other CMP resources (memory, interconnect and I/O). We plan to investigate the impact of QoS for realistic applications and virtual machined deployed on future architectures. Last but not least, we found that the lack of benchmarks for virtualization and multi-tasking scenarios is a significant problem for experimenting with such usage models. This is an open problem area that is to be addressed as these usage models become widespread.

REFERENCES

[1] Azul Compute Appliance," Azul Systems, can be found at http://www.azulsystems.com/products/cpools_cappliance.html

[2] P. Barham, et al, "Xen and the Art of Virtualization", In the Proceedings of the ACM Symposium on Operating Systems Principles (SOSP), October 2003

[3] D. Chandra, F. Guo, S. Kim and Y. Solihin, "Predicting inter-thread cache contention on a chip multiprocessor architecture", In Proc. 11th International Symposium on High Performance Computer Architecture (HPCA), Feb 2005

[4] T. Deshane, D. Dimatos, et al., "Performance Isolation of a Misbehaving Virtual Machine with Xen, VMWare and Solaris Containers," , submitted to USENIX 2006; also at http://people.clarkson.edu/~jnm/publications/isolationOfMisbehavingVMs.pdf

[5] L. Hsu, S. Reinhardt, R. Iyer and S. Makineni, "Communist, Utilitarian, and Capitalist Cache Policies on CMPs: Caches as a Shared Resource," International Conference on Parallel Architectures and Compilation Techniques (PACT), 2006

[6] R. P. Goldberg, "Survey of virtual machine research," IEEE Computer, 34—45, 1974.

[7] Intel Corporation. "Intel Dual-Core Processors -- The First in the Multi-core Revolution," http://www.intel.com/technology/computing/dual-core/

[8] R. Iyer, "CASPER: Cache Architecture, Simulation and Performance Exploration using Re-streams," Intel's Design and Test Technology Conference (DTTC), 2001.

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

[10] C. Kim, D. Burger, S. W. Keckler, "Nonuniform Cache Architectures for Wire-Delay Dominated On-Chip Caches," IEEE Micro 23(6): 99-107 (2003)

[11] S. Kim, D. Chandra, and Y. Solihin, "Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture", in Proc. Of the 13th International Conference on Parallel Architectures and Complication Technique (PACT), Sep-Oct 2004

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

[13] K. Olukotun, B. A. Nayfeh , et. al., "The case for a single-chip multiprocessor," Proceedings of the 7th International Conference on Architectural support for Programming Languages and Operating Systems, October 01-04, 1996.

[14] N Rafique, et al, "Architectural Support for Operating System-Driven CMP Cache Management", International Conference on Parallel Architectures and Compilation Techniques (PACT), 2006

[15] M. Rosenblum and T. Garfinkel : Virtual Machine Monitors : Current Technology and Future Trends. IEEE Computer 38(5) : 39-47(2005)

[16] "SPEC", http://www.spec.org/cpu2000/

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

[18] R. Uhlig, R. Fishtein, et. al., "SoftSDV: A Presilicon Software Development Environment for the IA-64 Architecture", Intel Technology Journal, Q4, 1999. (http://www.intel.com/technology/itj)