QoS Policies and Architecture for Cache/Memory in CMP Platforms

Ravi Iyer[1], Li Zhao[1], Fei Guo[2], Ramesh Illikkal[1], Srihari Makineni[1], Don Newell[1],
Yan Solihin[2], Lisa Hsu[3], Steve Reinhardt[3]
[1]Intel Corporation [2]North Carolina State University [3]University of Michigan, Ann Arbor
Contact Email: (ravishankar.iyer@intel.com, li.zhao@intel.com)

ABSTRACT
As we enter the era of CMP platforms with multiple threads/cores on the die, the diversity of the simultaneous workloads running on them is expected to increase. The rapid deployment of virtualization as a means to consolidate workloads on to a single platform is a prime example of this trend. In such scenarios, the quality of service (QoS) that each individual workload gets from the platform can widely vary depending on the behavior of the simultaneously running workloads. While the number of cores assigned to each workload can be controlled, there is no hardware or software support in today's platforms to control allocation of platform resources such as cache space and memory bandwidth to individual workloads. In this paper, we propose a QoS-enabled memory architecture for CMP platforms that addresses this problem. The QoS-enabled memory architecture enables more cache resources (i.e. space) and memory resources (i.e. bandwidth) for high priority applications based on guidance from the operating environment. The architecture also allows dynamic resource reassignment during run-time to further optimize the performance of the high priority application with minimal degradation to low priority. To achieve these goals, we will describe the hardware/software support required in the platform as well as the operating environment (O/S and virtual machine monitor). Our evaluation framework consists of detailed platform simulation models and a QoS-enabled version of Linux. Based on evaluation experiments, we show the effectiveness of a QoS-enabled architecture and summarize key findings/trade-offs.

Categories and Subject Descriptors
B.3.2 [**Hardware**]: Design Styles of Memory Structures – *cache memories.*

General Terms: Algorithms, Management, Measurement, Performance, Design, Experimentation

Keywords: Quality of Service, CMP, Cache/Memory, Performance, Service Level Agreements, Resource Sharing Principles

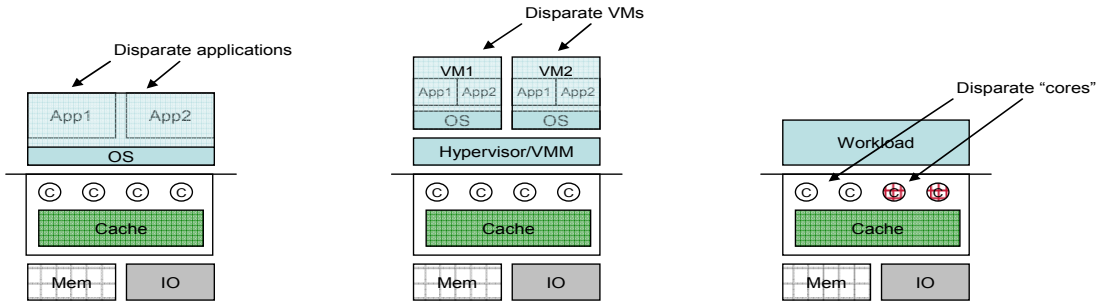
1. INTRODUCTION
As the momentum behind chip multiprocessor (CMP) architectures [7][12][18] continues to grow, it is expected that future microprocessors will have several cores sharing the on-die and off-die resources. The success of CMP platforms depends not only on the number of cores but also heavily on the platform resources (cache, memory, etc) available and their efficient usage. In general, CMP architectures are being designed to perform well when a single parallel application is running on them. However, CMP platforms will also be used to run multiple applications simultaneously. The rapid deployment of virtualization [2][21][23][31] as a means to consolidate multiple applications onto 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 not be deterministic because it depends heavily on the behavior of the other simultaneously running workloads. As expected, recent studies [3][5][9][10][19][27] 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 two important platform resources – cache (space) and memory (bandwidth) – in our investigation and identify QoS policies and mechanisms to efficiently manage these resources in the presence of disparate applications (or threads).

Recent studies on partitioning of (cache) resources 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 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 isolation [1][4] desired for some applications. Since the relative importance of the deployed applications is best known in the operating environment, we introduce the need for software-guided priorities (e.g. assigned by administrators) to efficiently manage hardware resources. While the objective of priorities may be intuitive, the considerations, trade-offs and implications of these priorities are far from obvious. In this paper, we describe the issues involved and the basis for prioritization (i.e. how priority classes are specified and what they mean to the resource distribution).

The primary contribution of this paper is the design and evaluation of several priority-based resource management policies for an effective QoS-aware cache/memory architecture. The proposed QoS policies differ in terms of prioritization goals (high priority targets and low priority constraints), the monitoring metrics used (resource vs. performance-based) and the nature of resource assignment (static vs. dynamic). We evaluate the QoS-aware architecture and policies in the context of virtualization-based consolidation usage models, heterogeneous CMP

SIGMETRICS'07, June 12-16, 2007, San Diego, CA, USA.



(a) Multi-Tasking Scenarios (b) Virtualization/Consolidation (c) Heterogeneous Cores

Figure 1. Disparate Threads of Execution on CMP Platforms

architectures and traditional multi-tasking environments. Our evaluation framework is based on detailed trace-driven simulations as well as a prototype QoS-aware version of Linux.

We show that the proposed QoS-aware memory architecture can improve the performance of the high priority application significantly in the presence of other applications. We also show that dynamic policies are important since they allow the hardware to guide the resource allocation based on actual performance changes as well as constraints.

The rest of this paper is organized as follows. The case for QoS and related work are presented in Section 2 and 3. Section 4 covers the QoS design space (goals, targets and considerations). Section 5 introduces QoS policies and the QoS-aware architecture. Section 6 presents the evaluation framework and analyzes effectiveness of proposed QoS policies. Section 7 summarizes findings and presents a direction for future work.

2. A CASE FOR QOS IN CMP PLATFORMS

In this section, we motivate QoS needs by describing disparate threads and the shared resource problem.

2.1 Disparate Threads of Execution

As shown in Figure 1, the key trends that point to disparate CMP threads of execution are as follows:

(a) ***Multi-tasking becomes more common:*** As more threads/cores are enabled on 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 can be quite disparate (e.g. cache-friendly versus streaming). It is also possible that one application is of more importance than another (e.g. business-critical application executing with network backup).

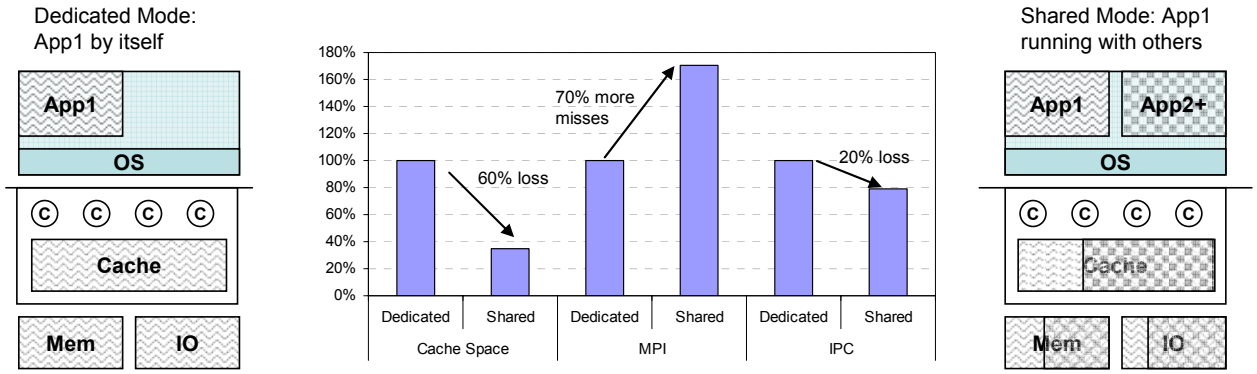
(b) Virtualized workloads becoming mainstream: While the concept of virtualization [6] has been around for a while, the recent re-emergence of virtualization as a means to consolidate workloads in the datacenter reflects the need to pay attention to the performance behavior of virtual machines running heterogeneous workloads simultaneously on a server, as shown in Figure 1b. This becomes even more important as virtualization-based usage models continue to rapidly evolve and encompasses office workstations/desktops and even home PCs/laptops. In these scenarios, many disparate workloads are consolidated together and performance isolation [4] is desired for the high priority applications that can be identified by user or administrator.

(c) Heterogeneous CMP architectures are attractive: In addition to diverse workloads, we are also at a point in the CMP evolution where not only heterogeneous cores [13] but also co-processors and engines (e.g. TCP offload, graphics, crypto) are being explored for integration on the die. These diverse "threads" of execution (as illustrated in Figure 1c) are known to have different behavior as compared to typical applications running on the general-purpose cores on the die. Depending on the workload, it is also possible that either the general purpose application or the special-purpose function is more important to optimize.

2.2 The Shared Resource Problem

Cache and memory are two key platform resources that affect application performance. While memory has always been shared in multiprocessor platforms, the emergence of CMP architectures now makes cache (typically the last level in the hierarchy) also a shared resource amongst the threads on the die. In addition to cache and memory, other resources that are shared include interconnects (on-die and off-die) as well as I/O devices. While we expect that all shared resources will require priority-based management in the platform, we focus in this paper primarily on cache and secondarily on memory. The resource characteristics that need to be provisioned for QoS differ significantly between cache and memory as considered in this paper. For cache, it is the space shared by disparate threads, whereas for memory, it is the bandwidth that is shared between disparate threads running simultaneously.

Figure 2 illustrates the motivation for QoS in this context. 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 other low priority applications. We chose an OLTP trace (TPC-C like) to represent a high priority application and a Java workload trace (SPECjbb2005 like) and a networking workload trace (NTttcp) to represent the low priority applications. We ran the high priority application in isolation (dedicated mode) and along with the low priority applications (shared mode). The study showed how sharing cache affects the performance of the high priority application (OLTP). The cache performance of the high priority application reduced significantly (~1.7X increase in MPI) since the cache space available to this application was only 35%. In order to minimize the loss of performance (~20%), the priority of the application needs to be comprehended by the platform in order to re-allocate cache resources. In this paper, we investigate QoS policies and mechanisms to manage the cache (and memory) resource distribution between high and low priority applications.



(a) Dedicated Mode Execution (b) Dedicated vs. Shared Performance Implications (c) Shared Mode Execution

Figure 2. Disparate Threads and Shared Resources: Illustrating the need for QoS

3. RELATED WORK

Previous research on partitioning for cache/memory architectures can be found in [5][9][10][17][19][19][33] The primary optimization goal for the techniques presented in most of these papers has been either fairness [10][17][33] or improving overall throughput [19]. For example, Kim et al. [10] proposed a fair policy that 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. However, their dynamic algorithm requires an offline profile for each thread's execution. Yeh et al. [33] presents a dynamic scheme which is based on on-line statistics and re-configures the cache partitioning accordingly. To provide fairness as well as QoS on available memory bandwidth across threads, Nesbit et al. [17] proposed a memory scheduler by adapting fair queuing techniques from network research. To improve overall performance through unequal distribution of cache resources, Qureshi et al [19] presents a low overhead online mechanism to keep track of cache utility for each thread and direct cache space distribution.

Although QoS has been studied for a long time in real-time environments [24], networking [35] and multimedia [15], it has only been recently introduced to the CMP architecture[9]. Iyer [9] described the need for a priority-based framework to introduce QoS in CMP caches. However, the paper did not present a detailed description of policies, metrics or complete architectural support. In this paper, we present a complete proposal with optimization goals, policies and metrics as well as hardware support in the CMP platform.

4. QOS GOALS AND CONSIDERATIONS

In order to design appropriate QoS policies and mechanisms it is important that the goals, metrics and constraints are considered and incorporated.

4.1 QoS Philosophy & Goals

The first step is to ensure that the goal of the QoS policy is well defined. Hsu et al. [5] describe various optimizations goals and policies for cache sharing in a CMP platform. Three types of policies are described – capitalistic, communist and utilitarian. The capitalist policy is essentially the baseline LRU-based cache implemented in most processors today. This policy allows each individual thread to grab cache space based on the frequency of the access. As a result, the faster it generates memory accesses, the more allocation it is able to accomplish. 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. Finally, the communist policy attempts to equalize the cache resource or performance available to each of the individual applications.

In this paper, our basic philosophy is different in that it 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. As a result, we propose a new policy that could be called "elitist", as it caters to the elite application(s) at the possible expense of the non-elite classes of applications. Elitist policies have several key considerations and requirements: (a) the classification of applications into elite vs. non-elite, (b) the nature of the QoS that an elite application should be provided and (c) the extent to which non-elite applications is allowed to suffer. In following subsections, we will further discuss considerations and propose solutions that address them.

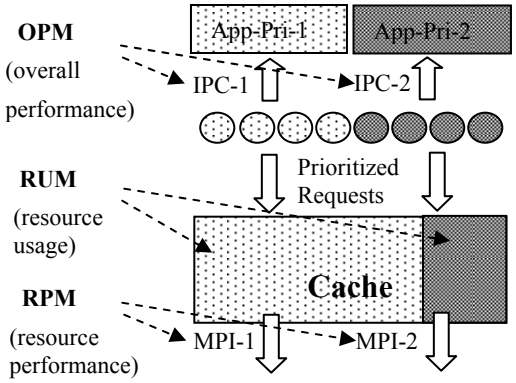


Figure 3. Metrics for QoS Policies (E.g. Cache)

4.2 QoS Metrics

The purpose of the elitist QoS policy is to improve the performance of the high priority application at the expense of

others. To specify and/or measure the efficacy of a QoS policy, we propose three types of metrics (see Figure 3):

- **Resource Usage Metrics (RUM):** The underlying mechanism to improve the performance of the high priority application is the amount of resource it is provided. So RUM (e.g. cache space) could be used as a metric to measure both resource QoS as well as its contribution to overall QoS if multiple resources are involved. Specifying the usage needs for all of the platform resources also enables creation of a virtual platform architecture tailored to the application or virtual machine of interest.
- **Resource Performance Metrics (RPM):** Providing more resources (measured by RUM) does not always ensure that the application performance will improve. For example, there are applications that are streaming in nature where providing additional cache space does not help. As a result, it may be better to use resource performance (e.g. misses per instruction for the cache resource) as a metric as opposed to resource usage itself.
- **Overall Performance Metrics (OPM):** The contribution of a certain resource and its performance to the overall platform performance depends highly on the platform as well as application characteristics. Ultimately, it is best to measure the overall performance (e.g. IPC).

4.3 QoS Targets & Constraints

To define an appropriate QoS policy and mechanism, it is important to understand the targets and the constraints. The target of the QoS policy is the extent to which the high priority application should be improved whereas the constraint ensures that the low priority application does not suffer beyond a certain point. In order to understand this further, let's define the target first. Figure 4 describes the bounds on high-priority applications, low-priority applications and overall performance of the platform.

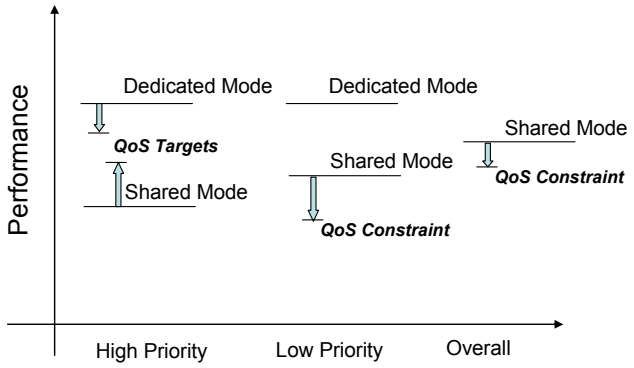


Figure 4. QoS Targets and Constraints

As illustrated in Figure 4, the high priority application performance is essentially bounded by its performance in dedicated mode versus shared mode. The QoS target is to achieve somewhere in between. At one extreme, to achieve the dedicated mode performance, all resources have to be provided to the high priority application. At the other extreme, if shared mode performance is sufficiently close to dedicated mode performance, QoS may not be required. Another consideration is that the QoS target for the high performance application can be specified as either a distance from the dedicated mode execution or a distance from the shared mode execution. Online monitors or offline profiling may be required to measure these bounds and provide guidance on setting targets.

While attempting to achieve high performance for the high priority application, it may be important to control the degradation of the low priority applications or the overall performance of the platform. As shown in Figure 4, the QoS constraint is this degradation threshold that is allowable in the platform. It would be ideal if the low priority is less affected if fewer resources are provided to it, but that may not always be the case. If the QoS target and QoS constraint are set independently, we expect that only best effort QoS can be supported by the platform. If hard guarantees are required on the resource itself, the choices that need to be made may be quite different. In this paper, we focus on best effort QoS and show how best we can provide resource and performance benefit for high priority applications while meeting the constraints.

5. QOS POLICIES AND ARCHITECTURE

In this section, we outline proposed QoS policies based on specific goals, metrics, targets and constraints. We also present a QoS-aware memory architecture that forms the underlying infrastructure to implement these policies.

5.1 Proposed QoS Policies

The proposed QoS policies differ based on whether they require static or dynamic resource allocation, whether they consider targets or constraints, and the metric on which they are based. While the QoS policies can be defined for any number of applications and priority levels, we discuss the policies in the context of one high-priority level and one low-priority level. The two primary policies (static vs. dynamic) and associated sub-policies are described in the subsections below.

5.1.1 Static QoS Policies

We define a policy as static if the hardware mechanism required for it does not need to perform dynamic resource assignment. In other words, static policies are defined such that the specified target and constraint do not require continuous adjustment of resources. As a result, a static policy specifies the QoS target and/or constraint in terms of the resource usage metric (RUM; e.g. cache space) provided to the high priority application and low priority applications respectively. It should be noted that while the hardware policy is static, the software policy can choose to dynamically modify the prioritization.

For the cache resource, we need to be able to specify the cache space allowable for each priority level and ensure that the priority level does not exceed this threshold during execution. It is also important that cache and memory QoS work cooperatively. If a low priority application is constrained in cache space, it will start to occupy more memory bandwidth and thereby affect the high priority applications and potentially cause priority inversion. The resource metric of interest for memory QoS is bandwidth. To control memory bandwidth allocation, we control the rate at which requests from different priority levels are allowed to be issued to the memory subsystem. Nesbit et al. [17] describe an interesting re-design of the memory controller to achieve the goal of bandwidth guarantees. To avoid significant re-design, we evaluate a simpler approach. Re-ordering of requests is a common optimization in the memory controller for improving memory efficiency and performance [16][22][36]. Based on this re-ordering optimization, we control memory bandwidth by allowing

requests from a high priority application to bypass requests from a low-priority application. The extent to which requests can be bypassed (e.g. 5 high priority requests before serving one low priority requests) indicates the ratio of bandwidth (e.g. 5:1) that is provided to the incoming priority levels. It should be noted that we take this approach because (a) we are more interested in bandwidth differentiation rather than bandwidth guarantees and (b) it is very simple to implement.

5.1.2 Dynamic QoS Policies

Dynamic QoS requires resources to be continuously re-allocated based on the resultant performance and the targets/constraints. In this paper, we evaluate the ability to do this dynamic re-allocation in hardware. However, it should be noted that it is possible to accomplish the re-allocation in software as well if all of the monitoring feedback is provided to the execution environment (OS or VMM). The targets and constraints can be specified in terms of resource performance (*Dynamic QoS RPM*) or overall performance (*Dynamic QoS OPM*). Also, depending on whether the constraints are used at all or how they are specified, the sub-policies can be further sub-categorized into (a) *Target* – where a target is specified for the high priority application and the constraint is ignored, (b) *LoPriConstraint* – where instead of a high-priority target, a constraint is specified that the low-priority application should not degrade below a certain resource or overall performance level, and (c) *OverallConstraint* – where instead of a high priority target being specified, a constraint that the (resource or overall) performance of the overall platform should not drop below a certain threshold is specified. In these policies, the amount of resource as well as the resultant performance provided to a high priority application or a low priority application need to be monitored at regular intervals in the platform. If the performance of the high priority application is lower than the target or the degradation threshold for the low priority application or for overall platform is not crossed, then the amount of resources assigned to the high priority application is increased by a pre-specified amount. The architecture and implementation for such QoS policies are discussed below.

5.2 A QoS-Aware Memory Architecture

In this subsection, we present a layered QoS architecture that implements static and dynamic cache resource assignment for the QoS policies. Our proposed QoS-aware memory architecture consists of three primary layers: priority enforcement, priority assignment and priority classification. For simplicity, we first assume that there is only one high priority thread and one low priority thread. We later describe extensions to the architecture for more priority levels and more applications. Figure 5 shows the three layers and the hardware support for each layer.

5.2.1 Priority Classification

The priority classification layer is responsible for identifying and providing the QoS information: the priority levels of each application (0 for high and 1 for low) and the associated targets/constraints. We expect that the metric of choice (RUM vs. RPM vs. OPM) is ultimately standardized and exposed to the user in a consistent manner. As shown in Figure 5, this layer requires support in the execution environment (either OS or hypervisor) as well as the processor architecture. Operationally, support (in the form of a QoS API) is required for the user or administrator to supply the required QoS information to

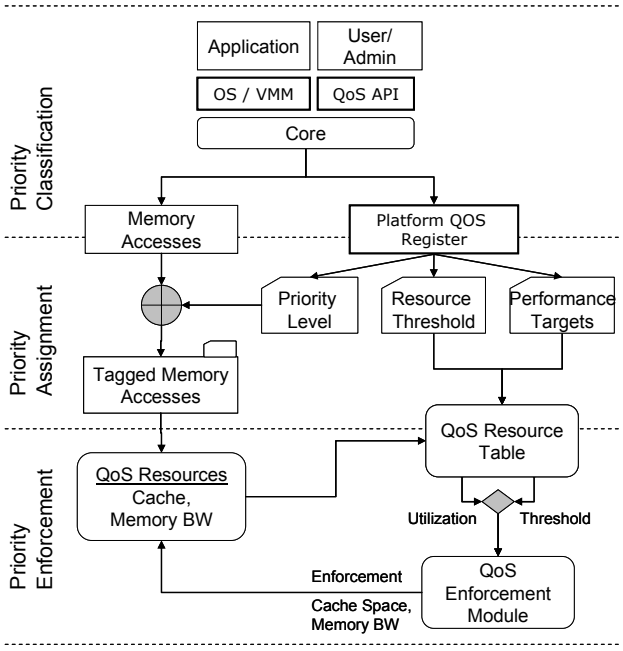


Figure 5. QoS Architecture Layers and Components

the execution environment. The support in the execution environment is the ability to maintain the QoS information in the thread state and the ability to save and restore it in the processor's architectural state when the thread is scheduled to run. The support in the processor is essentially a new control register called Platform QoS Register (PQR) in order to maintain the QoS information (in the architectural state) for the runtime. The execution environment sets the PQR with the platform priority level of the currently running application at schedule time. In addition, this register will also be used to convey the mapping of priority levels into resource thresholds (for static QoS) and the mapping of priority levels to targets/constraints (in case of dynamic QoS).

Priority Level	Resource Targets	Performance Targets
Two bits to indicate four Application Priority levels – 00: Priority A – 01: Priority B – 10: Priority C – 11: Priority D	15 bits for static resource Targets for a priority level - 2 bits for resource type - 13 bits for Threshold - Cache space (%) - Memory BW (%)	15 Bits to specify the Dynamic QoS Targets/Constraints - 1 bit for Target/Constraint - 4 bits for performance metric id - 10 bits for performance value

Figure 6. QoS Information Encoding in PQR

5.2.2 Priority Monitoring & Assignment

Figure 5 also illustrates the components of priority assignment layer in the QoS-aware memory architecture. Figure 6 shows a potential encoding of QoS information that needs to be passed down from the execution environment to the platform. The first field indicates the priority level of the executing thread. This information is written into the PQR every time a new thread is scheduled on to the processor. The second field is used for static

QoS and indicates the thresholds for each resource type (cache, memory, etc) for each priority level. For example, it specifies the space threshold for cache and the bandwidth threshold for memory in discrete quantities or as ratios respectively. This information is only provided when the execution environment initializes or needs to modify the resource thresholds for the priority levels supported by the platform. Whenever this field is updated in the PQR, the information is passed to the QoS Resource Table (QRT) which will be described later. For dynamic QoS, the third field is used to specify the target and/or constraint for the priority level. It is desirable to specify the target/constraint in terms of a multiplier to the actual performance (RPM or OPM). For example, if the target is specified as 1.2, then the goal is to achieve 20% more performance (IPC) when using OPM as the metric. In the PQR, one bit is used to indicate target vs. constraint, a few bits are used to indicate which performance metric (RPM vs. OPM, cache vs. memory, etc) and then a subset of the multiplier values are encoded in the remaining bits. Whenever the PQR is written, these resource and performance targets are passed down to the QoS Resource Table and the QoS Enforcement module if necessary. Once the mapping of priority level to resource or performance targets are established, the next step is to ensure that every memory access is tagged with the priority level. By tagging each memory access with the associated priority level (from the PQR), monitoring and enforcement of resource usage is made possible locally within each subsystem (cache, memory, etc). This is described in more detail in the next section.

5.2.3 Priority Enforcement

Figure 5 illustrates the priority enforcement layer in the architecture and shows the components involved. The inputs to the enforcement layer are the tagged memory accesses and the QoS resource table. As shown in Figure 7, each line in the cache is tagged with the priority level in order to keep track of the current cache space utilization per priority level. The QoS resource table uses this information to store the cache utilization per priority level. This is done simply by incrementing the resource usage when a new line is allocated into the cache and decrementing the resource usage when a replacement or eviction occurs. The QoS resource table also maintains the number of memory accesses (cache misses and writebacks). By doing so, it can also keep track of the bandwidth consumption in memory.

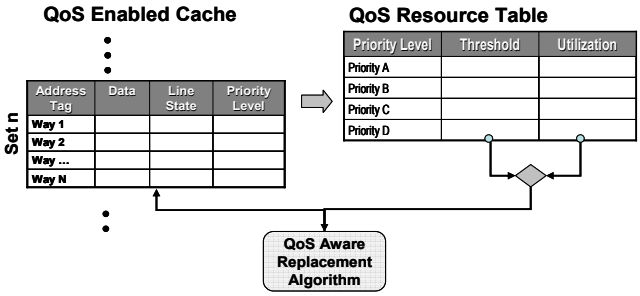


Figure 7. QoS-Enabled Cache Enforcement

For priority enforcement, there are two more functions that are critical: (a) *Static QoS:* to make sure the resource utilization stays within the static QoS threshold specified in the QoS Resource Table and (b) *Dynamic QoS:* to dynamically adjust the threshold to satisfy the performance target/constraint specified. The static QoS policy is achieved by modifying the replacement policy to be QoS aware. For each priority level, the utilization and the static QoS thresholds are available (in the QRT) on a per priority basis. If the utilization is lower than the specified threshold, then the replacement policy works in normal mode using the base policy (like LRU). When the utilization reaches the static QoS threshold, the replacement policy overrides the LRU policy to ensure that it finds a victim within the same priority level. In some corner cases, it is possible that the set does not contain a line from the same priority level even though the utilization threshold has been reached. In these cases, a victim is chosen from a lower priority level or at random if none is available. Similar cache partitioning mechanisms have been presented before [9][19][27][28].

At each interval, perform the following check:
Target-Only: If ($perf_{hi}$ < $target_{hi}$) {reduce lo threshold; increase hi threshold}
Lo-Constraint: If ($perf_{lo}$ > $target_{lo}$) {reduce lo threshold; increase hi threshold}
Overall-Constraint: If ($perf_{ovl}$ > $target_{ovl}$) {reduce lo threshold; increase hi threshold}
All-of-the-Above: Else {restore previous interval thresholds}

Figure 8. Basic 2-Priority Dynamic QoS Heuristic

To enforce dynamic QoS policy, the QoS Enforcement Module (QEM shown in Figure 5) monitors the performance (cache misses as well as cycles per instruction) at frequent intervals. A basic description of a dynamic QoS heuristic (for 2 priority levels) is shown in Figure 8. As long as the target is not achieved or the constraint is not violated (as specified by the dynamic QoS policy described in section 5.1.2), the QEM modifies the resource thresholds by reducing it for low priority and increasing it for high priority. The granularity at which the resource threshold increases/decreases is a parameter that is pre-specified (e.g. 10% of cache size). The QEM can also address underutilization issues by modifying resource thresholds. In the memory subsystem, similarly, the QEM can decide to change the bandwidth ratio (e.g. from 3:2 to 4:1). If the target is reached, then no changes are made to the thresholds subsequently. If the constraint is violated, the setting for the previous interval is restored. For implementing this, the QoS resource table is extended to maintain current and past resource utilization and thresholds, resource performance and overall performance. Since the table is small (< 1KB), the overhead of maintaining this information is negligible.

To extend the dynamic QoS heuristic for multiple priority levels, additional parameters are needed: (a) separation level and (b) split ratio. The separation level indicates that all priority levels below it will have resources reduced and the ones above it will be provided those resources. The split ratio indicates how these resources will be stolen and distributed amongst the priority levels. In the evaluation section, we will show an example with three priorities (where the separation is set to the lowest priority level and the split ratio is varied).

6. QOS EVALUATION & PROTOTYPING

In this section, we present two approaches (trace-driven simulation & software prototyping) to evaluate QoS.

6.1 Simulation-Based Evaluation

In this subsection, we describe the trace-driven simulations for evaluating QoS policies and architecture.

6.1.1 Simulation Framework

We developed a trace-driven platform simulator called ManySim [34] to evaluate CMP architectures and QoS policies. ManySim simulates the platform resources with high accuracy, but abstracts the core to optimize for speed. The core is represented by a sequence of compute events (collected from a cycle-accurate core simulator) separated by memory accesses that are injected into the platform model. ManySim contains a detailed cache hierarchy model, a detailed coherence protocol implementation, an on-die interconnect model and a memory model that simulates the maximum sustainable bandwidth specified in the configuration. The CMP architecture (a somewhat scaled down version of a Niagara-like [11][14] architecture) used for evaluation is shown in Figure 9. There are two cores, with each one having four threads and its own private L1 and L2. Both cores share the last level cache (L3) where the QoS policies are enforced. When we run multi-threaded applications with two priority levels, each core is running a different application (the first four threads run the high priority application whereas the second four threads run the low priority application). When we run three applications with three different priority levels, the high, mid and low priority applications are running on the first three threads, the next three and the last two threads respectively.

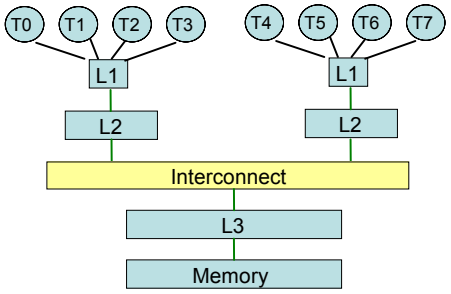


Figure 9. Evaluated CMP Environment

Table 1 summarizes the simulation configurations. As shown in the table, we model CMP architecture with a 3-level cache hierarchy and simple in-order cores. In our simulation, the 8 threads share a 1MB last-level cache. The available memory bandwidth is set to be 8 GB/s. For QoS evaluation, ManySim was modified to allow cores and traces to be tagged with priorities. The priorities were then provided to the cache/memory subsystem. The QoS hardware components (QRT, QEM, QoS-aware replacement, etc) were implemented in ManySim. The specific cache QoS policies evaluated using Manysim are:

- Static QoS
- Dynamic QoS + MPI_Target
- Dynamic QoS + Overall_MPI_Constraint
- Dynamic QoS + LoPriority_MPI_Constraint
- Dynamic QoS + IPC_Target
- Dynamic QoS + Overall_IPC_Constraint
- Dynamic QoS + LoPriority_IPC_Constraint

We also evaluate memory QoS for the base architecture with and without cache QoS. The specific QoS parameters used are described along with the results. In all cases, we model cache and memory contention effects accurately.

Table 1. ManySim Simulation parameters

Parameters	Values
Core	4GHz, In-order, 4 threads
L1 I/D cache	32 Kbytes, 4-way, 64-byte block
L2 cache	128K bytes, 8-way, 64-byte block
L2 cache hit time	10 cycles
MSHR size	16
L3 cache	1M bytes, 16-way, 64-byte block
L3 cache hit time	50 cycles
Interconnect bandwidth	128GB/s
Memory access time	400 cycles
Memory bandwidth	8GB/s
Queues and Other Structures	Memory Queue (16) L3 MSHR (16) Coherence Controller Queue (16) Interconnect Interface (8 entries)

6.1.2 Workloads & Traces

We chose a few commercial multi-threaded server workloads (OLTP, SPECjbb) and a networking workload (NTttcp). Running these simultaneously allows us to experiment with virtualization-based consolidation usage models (two or three workloads running simultaneously) as well as heterogeneous CMP architectures (when one of the workloads is NTttcp) described earlier in Section 2.1.

OLTP: For representing OLTP, we used traces from multi-threaded TPC-C-like workload [30], which is an online-transaction processing benchmark that simulates a a warehouse environment where a user population executes transactions against a database. The benchmark is based on order-entry transactions (new order, payment, etc).

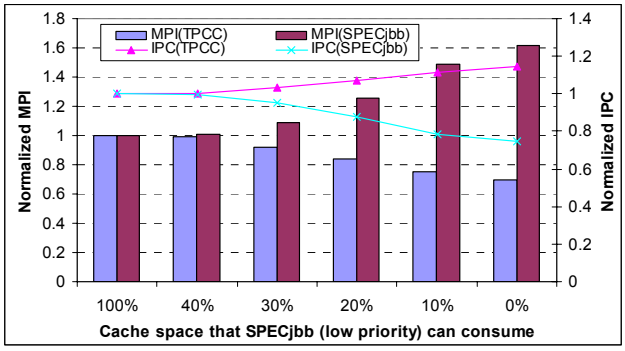
Java: SPECjbb2005 [26] is a Java-based multi-threaded server benchmark that models a warehouse company with warehouses that serve a number of districts (much like TPC-C). This workload stresses the performance of JVM internals (garbage collection, runtime optimization, etc).

NTttcp: NTttcp is commonly used to test network I/O (packet processing for transmit/receive) and contains a lot of transient/streaming data as a result. This is a Windows version for ttcp [29] micro-benchmark.

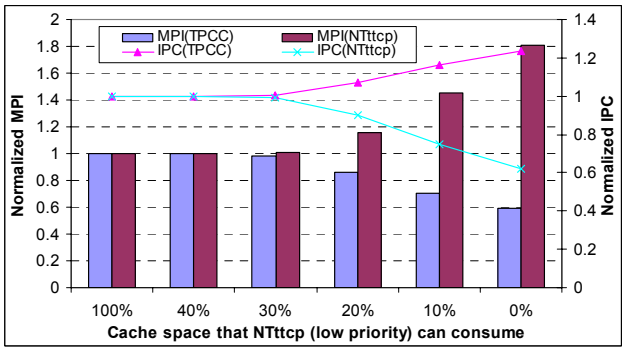
6.1.3 Evaluating Static QoS Impact

The static QoS policy essentially allows more cache (space) and memory (bandwidth) resources to be assigned to the high priority application. For cache, we do so by limiting the amount of space in the cache that the low priority application can allocate. As a result, our evaluation of the static QoS policy is done by varying this cache space limit for low priority applications from 0% to 40% of the cache size. We compare this to the shared mode execution without QoS which is denoted by a cache space limit of 100%. We use the two RPM/OPM primary metrics to evaluate effectiveness: resource performance denoted by MPI (misses per instruction) and overall performance denoted by IPC (Instructions per Cycle).

Figure 10 shows the impact of the QoS policy when executing TPCC with SPECjbb and TPCC and NTttcp. The two y-axes illustrate the MPI (bars) and IPC (lines) respectively, normalized to the case where both workloads share the cache without any prioritization. We find that as we reduce the cache space available for SPECjbb (in Figure 10a), the MPI for TPCC is decreased. When SPECjbb can only take up to 10% of the total cache size, the MPI for TPCC is reduced by about 25%, and as a result the IPC for TPCC is increased by 11.4%. On the other hand, as expected, the MPI for SPECjbb is increasing and its IPC is reduced. One of the important findings from this study is that since the low priority application gets affected, it is important to enforce constraints on how degraded its performance can get.



(a) TPCC (hi) + SPECjbb (lo)

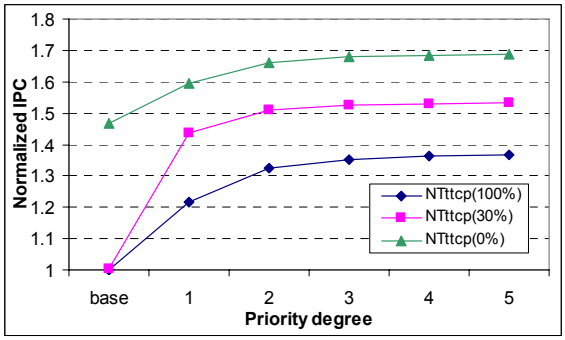


(b) TPCC (hi) + NTttcp (lo)

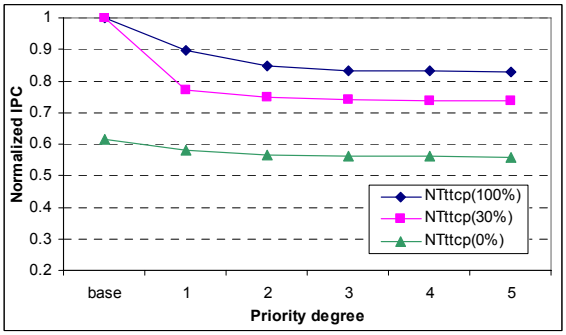
Figure 10. Impact of Cache QoS (Static Policy)

We next evaluate the impact of QoS policies for memory bandwidth prioritization when running TPCC and NTttcp (TPCC having higher priority). Note that although we have the results of all the combinations for different workloads, we pick one of them for brevity. It should also be noted that we simulated a bandwidth-constrained environment to study memory QoS to mimic pin constraints on future generations of large-scale CMP platforms. For memory bandwidth, the resource is controlled by allowing multiple high priority requests in the queue to be serviced before a pending low priority request. The number of requests that are serviced for high priority before a low priority request is labeled as the priority degree (same as bandwidth ratio). The base case is where the memory requests are serviced in order of arrival.

Figure 11 shows the impact of the static memory QoS policy on the IPC of the high priority application (in (a)) and the low priority application (in (b)). In the figure, we show three curves denoted by 100% (shared mode execution without cache QoS), 30% (shared mode execution with 30% cache space limit on low priority application) and 0% (shared mode execution with low priority applications bypassing the last-level cache). It should be noted that there are two "opposite" memory effects occurring as more cache space is provided to the high priority application: (a) fewer high priority misses go to the memory subsystem and as a result, the dependency on memory bandwidth is lower, and (b) more low priority misses go to the memory subsystem and as a result, it occupies more of the memory bandwidth.



(a) TPCC (high priority) Performance



(b) NTttcp (low priority) Performance

Figure 11. Memory QoS Impact

As shown in Figure 11 (a), even a small increase in priority degree (value of 1) improves the IPC of TPCC (high priority workload) by as much as 21%. As we increase the priority degree further, the IPC increases more slowly and remains constant once the priority degree reaches 3. When QoS policy is enforced on cache, the IPC is increased by another 20% (NTttcp-30% case) and 16% (NTttcp-0% case) respectively except for the base case. As shown in Figure 11(b), the IPC for low priority workload (NTttcp) is reduced. However the reduction in low priority performance is not as much as the increase for the high priority application. When priority degree is 1, the IPC is reduced by 10%. Similar behavior can be seen for other two curves. Another notable behavior is that when NTttcp is fully bypassing the cache (0%), the bandwidth priority does not have much impact. This occurs partially because TPC-C has more space in cache and is less sensitive to changes in memory latency.

6.1.4 Evaluating Dynamic QoS Impact

For dynamic QoS, we only present the impact of the policy on cache for lack of space. The dynamic policies evaluated differ in target/constraint metric (RPM vs. OPM) as well as the type of target or constraint used. Table 2 shows the parameters used for the six resultant policies. These policies are compared to the base

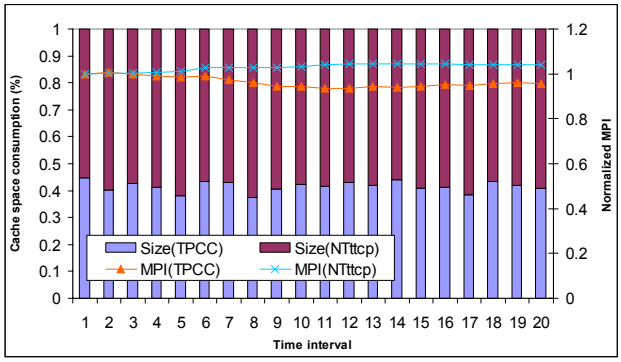
case where all the workloads have the same priorities. For two priority levels, we only show the results for TPC-C running with NTttcp for brevity. Figure 12 shows the implications of using dynamic QoS based on MPI as the RPM metric. The x-axis shows execution timeline broken into intervals of 300K references.

Table 2: Dynamic QoS Parameters

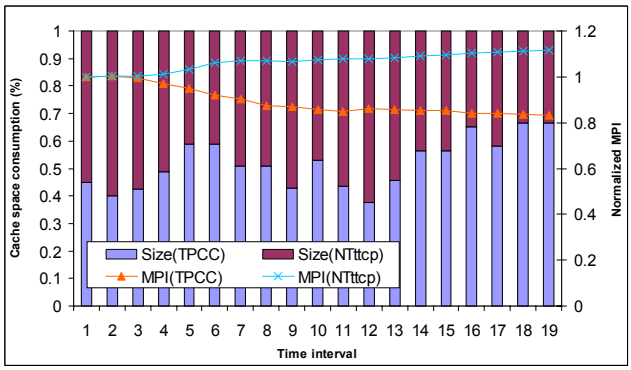
Policy	**RPM** (MPI multiplier)	**OPM** (IPC multiplier)
Target	0.8x (high priority)	1.2x (high priority)
Overall Constraint	1.1x (overall)	0.9x (overall)
Low-Pri Constraint	1.2x (low priority)	0.8x (low priority)

During a given interval, the resource allocation is constant. At the end of each interval, the QoS enforcement module re-evaluates the performance based on targets/constraints and re-allocates the cache space as appropriate. The two y-axes show the cache space utilized (bars) by each application and the normalized MPI (lines). Figure 12a shows the execution when no QoS is performed. The workloads exhibit a relatively steady cumulative MPI and cache space utilization with NTttcp occupying almost 60% of the cache space and TPC-C occupying 40%. Figure 12b shows the impact of target-only QoS, where a target of 0.8x MPI is achieved for the TPC-C workload without considering the effect on the NTttcp workload. The graph shows that cache allocation changes occurred during the run (as seen in the bars) whenever the MPI in a given interval is lower than the target. Figure 12c shows the effect of the applying overall MPI constraints on the performance. Since the overall MPI does not increase beyond the 1.1x constraint, we do not see significant amount of dynamic cache space re-allocation. Instead we find that the high priority application is constantly given additional resources (almost all of the cache) and benefits from a MPI reduction of more than 30%. Finally, Figure 12d shows the implications of applying a low priority constraint of not exceeding 1.2x MPI. As shown, the cache space is re-allocated a few times to ensure that this constraint is not violated. At the same time, the cache performance of the high priority application is improved significantly (MPI reduction of 20%). Although not shown due to space limitation, our experiments on IPC-based QoS show similar behavior and confirm that using IPC can be reasonable especially if only one resource is affecting overall performance.

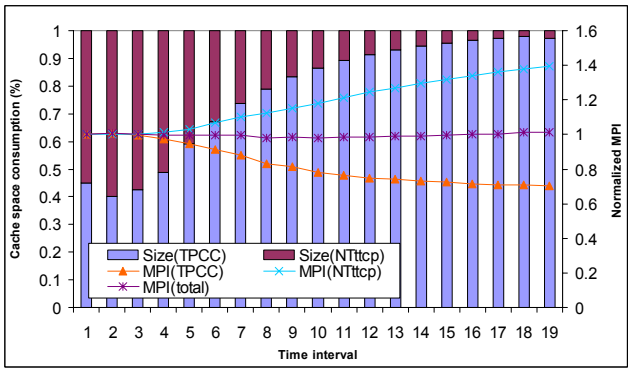
We also look at the QoS impact on all three applications running simultaneously, with TPCC having the highest priority, SPECjbb having the mid-level priority and NTttcp having the lowest priority. Figure 13 shows the data for the dynamic QoS policy with the following overall constraint: as long as the overall MPI does not increase by 2%, we steal resources from low priority applications and allocate those to mid and high priority applications. To do so, the first step is to reduce the cache space threshold for low priority application by a pre-determined amount (20% in this case). This freed up space is then made available to both high and mid-level priority applications. In order to ensure that the high priority is given more of this space than mid priority, we define a *split ratio*. The split ratio indicates the percentage of the freed up space provided to the mid-level priority application. For example, if the split ratio is 20%, then 20% of the space is



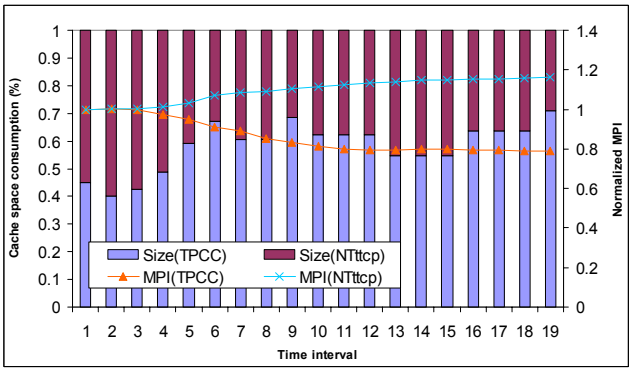
(a) No Cache QoS



(b) Cache QoS with Target-Only



(c) Cache QoS with Overall Constraint



(d) Cache QoS with Low Priority Constraint

Figure 12: Dynamic QoS using RPM

given to mid-priority and 80% is given to high priority. In Figure 13, we show the base case (no QoS) and three dynamic QoS cases, where we vary the split ratio. We can see that with the QoS policy, the cache space for TPCC is increased significantly, and as a result, its MPI is reduced about 20%. As expected, this is at the cost of NTttcp, whose MPI is increased by 25%. For the mid-level priority application, SPECjbb, its space consumption increases by only a small amount when the split ratio is 20% and 35%, and therefore the MPI does not change. However, when the split ratio is increased to 50%, its space consumption increases significantly, and its MPI reduces by 6%. In any case, the split ratio indicates a trade-off of space allocation between the high and mid-level priority applications.

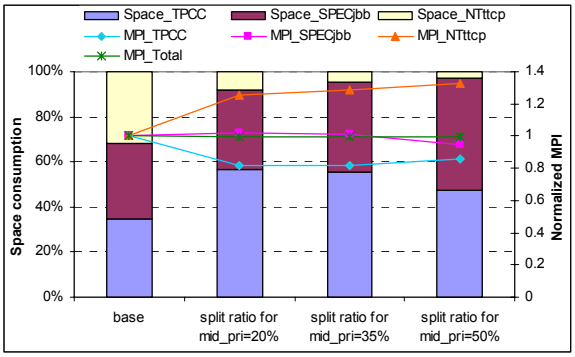


Figure 13. Dynamic QoS (RPM) on 3 priority levels

6.2 QoS Prototyping

To evaluate our QoS-aware architecture more realistically, we developed a software prototype and ran a limited set of experiments on a full system simulator. In this section, we describe this effort and our initial findings.

6.2.1 QoS-Aware Operating System

Our software prototype is a QoS-aware Linux operating system. This was accomplished by modifying the 2.6.16 Linux kernel to provide the following QoS support:

(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/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 to processor architecture state (Platform QoS Register) and vice-versa. 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 wouldn't be used by other I/O devices.

In addition, to allow administrators to pass QoS values for running processes, the Linux kernel was modified:

(a) QoS APIs for user/administrator: Two extra system calls were added to the kernel to provide access to 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.

6.2.2 Full-System Simulation

In order to evaluate the QoS-aware Linux OS on a QoS-aware architecture, we employed SoftSDV [32], a full-system simulator that allows us to model the architecture functionally and enables a performance model. We use the functional model of SoftSDV to boot the Fedora Core 5 Linux, including our QoS-enabled kernel. The functional model passes instructions executed by the applications running to the performance model. These instructions include the *out* instruction, which triggers the performance model to record the priority values into the architectural state. For the performance evaluation, we integrated a cache simulator [8] into SoftSDV. The cache simulator was modified to support static and dynamic cache QoS.

6.2.3 QoS Evaluation

We first look at a dual-core CMP with a shared last-level cache. This cache is an 8-way 256KB cache and is scaled down from 1M since the number of threads is reduced by a factor of 4 compared to the configuration used in the previous subsection. We choose two single-threaded applications from SPECint2000 benchmark suite [25] -- ammp and gcc, which show large cache sharing impact when they are co-scheduled. The standard *ref* is used as the input set. During the execution of these two benchmarks, we collect cache sharing statistics for about 100M instructions (after sufficient warmup).

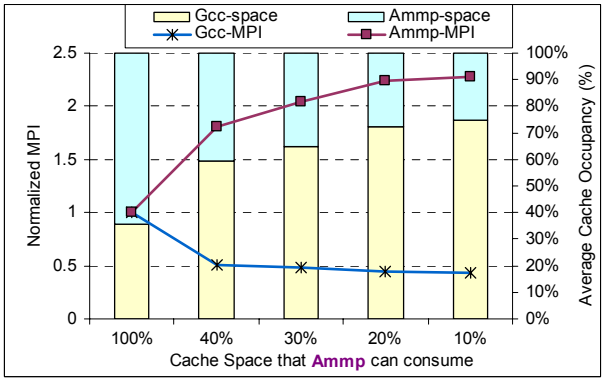


Figure 14: Impact of static QoS in two-core CMP

Figure 14 shows the static QoS evaluation results from our QoS prototype experiment when running gcc (high priority) and ammp (low priority) simultaneously. The two Y-axes represent the MPI (lines) and average cache space occupancy (bars) of the two applications respectively. The MPI value is normalized to the case when both applications share the L2 cache without any prioritization. As seen from the figure, the MPI of gcc reduces when we reduce the cache space available for ammp. This is accompanied by an increase in the MPI of ammp. The MPI reduction for gcc is about 57% when ammp was constrained to occupy 10% of total cache size. 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 can be because 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.

Next, we look at a four-core CMP platform, where we run one high priority application, two mid level priority applications and one low priority application which share a 1MB last level cache. It should be noted that we did not evaluate both mid-level priority applications contending within the same priority level. Instead we had two mid-priority levels with identical cache space thresholds (similar to having one middle-level priority with fair allocation within the priority level). For this scenario, each mid-priority application is limited to occupy 10% of total cache space and the low priority application will bypass the L2 cache (i.e. 0%) and essentially only use L1 cache. Figure 15 shows the impact of QoS when applu (high priority), art (mid level priority), gcc (mid level priority) and mcf (low priority) are co-scheduled. The MPI value of each application is normalized to the case when it shares the cache with other applications without prioritization. We can see that when we limit the cache space of art, gcc and mcf, the MPI of applu (high priority) reduces by 33% and the MPI of art, gcc and mcf increase by 21%, 150% and 216% respectively.

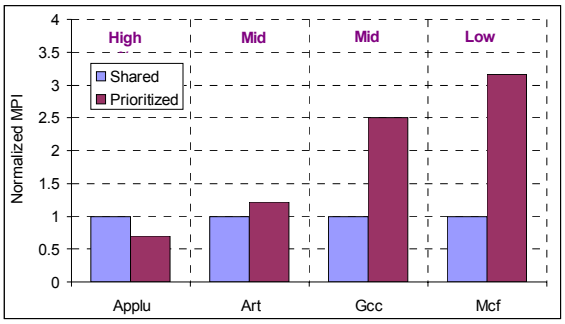


Figure 15: Impact of Static QoS in 4-core CMP

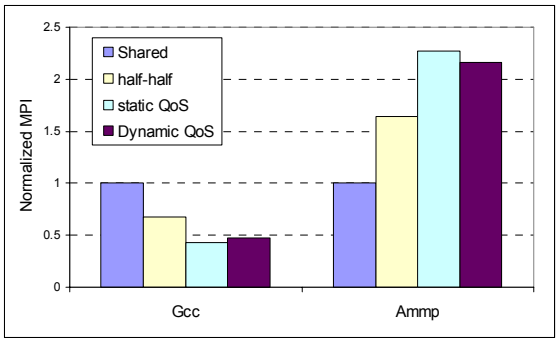


Figure 16: Comparison of Several QoS schemes

Figure 16 compares the MPI running gcc (high priority) and ammp (low priority) simultaneously as follows: shared mode (without prioritization), half-half mode (each application gets 50% of cache), static QoS mode (ammp is constrained to occupy only 10% of cache) and dynamic QoS (amount of cache is dynamically modified to improve high priority). The MPI value of each application is normalized to the case when it runs under shared mode. We can see that both static and dynamic QoS schemes efficiently improve the performance of gcc (high priority) while adversely affecting ammp (low priority).

7. CONCLUSIONS AND FUTURE WORK

In this paper, we presented the need for QoS in CMP architectures. We showed that efficient management of resources such as cache and memory are required to cater to high and low priority threads running simultaneously.

We described the philosophy and goals for investigating QoS policies by introducing resource-based and performance-based metrics. We presented the key considerations and metrics involved when defining QoS policies. The primary consideration in enabling QoS is to decide whether to enable it on a resource-basis or a performance-basis. We proposed several policies (static and dynamic) for resource-based as well as performance-based QoS. We showed the implementation requirements for a QoS-aware memory architecture for CMP platforms.

Through a detailed simulation-based evaluation, we showed that the QoS policies on cache and memory can be quite effective in optimizing the performance of high priority applications in the presence of other low priority workloads. We showed that significant performance improvements (20 to 30% reduction in MPI and 10 to 20% improvement in performance) can be achieved by providing additional cache space or memory bandwidth to the high priority application. The policies allow significant flexibility in modifying the amount of benefit achievable for the workload scenarios of interest. We also showed that considerations such as side-effects on overall performance or low priority performance can be addressed by enabling dynamic policies and implementing QoS enforcement modules. Last but not least, we validated our QoS architecture by implementing a software prototype and running it on a QoS-aware full-system simulation. Preliminary results from the prototype also show promising benefits for multi-tasking scenarios.

Future work in this area is as follows. We plan to investigate architectures and execution environments with many workloads running executing. In particular, we would like to apply the resource-based and performance-based approaches to virtual machine environments. We would also like to further experiment with our prototyping environment for application or VM scheduling implications. It is also important to evaluate dynamic software QoS approaches where the OS passes dynamic QoS hints to the platform.

8. ACKNOWLEDGMENTS

We are very grateful to the anonymous reviewers for their comments and feedback on the paper. We would also like to thank Jaideep Moses for his help in developing the ManySim simulator.

9. REFERENCES

[1] Azul Systems. *Azul Compute Appliance*. http://www.azulsystems.com/products/cpools_cappliance.html

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

[3] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multiprocessor architecture", In *Proc. of 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*. http://people.clarkson.edu/~jnm/publications/isolationOfMisbehavingVMs.pdf.

[5] L. Hsu, S. Reinhardt, R. Iyer and S. Makineni. Communist, Utilitarian, and Capitalist Policies on CMPs: Caches as a Shared Resource. In *Proc. of 15th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sept 2006.

[6] R. P. Goldberg. Survey of virtual machine research. *IEEE Transactions on Computers*, 1974.

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

[8] R. Iyer. On Modeling and Analyzing Cache Performance using CASPER. In *Proc. of 11th International Symposium on Modeling, Analysis and Simulation of Computer & Telecom Systems*, Oct 2003.

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

[10] S. Kim, D. Chandra, and Y. Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *Proc. of 13th Int'l Conf. on Parallel Arch. & Complication Techniques(PACT)*, Sept 2004.

[11] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-Way Multithreaded Sparc Processor.In *Proc. of Annual International Symposium on Microarchitecture(MICRO)*, Mar 2005.

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

[13] R. Kumar, D.M. Tullsen, N. P. Jouppi, P. Ranganathan. Heterogeneous Chip Multiprocessors. *IEEE Transactions on Computers*, 2005.

[14] J. Laudon. Performance/Watt: The New Server Focus. In *1st Workshop on Design, Architecture and Simulation of CMP (dasCMP)*, Nov 2005.

[15] K. Lee, T. Lin and C. Jen. An Efficient Quality-Aware Memory Controller for Multimedia Platform SoC. *IEEE Trans. On Circuits and Systems for Video Technology*, May 2005.

[16] C. Natarajan, B. Christenson, and F. Briggs. Performance Impact of Memory Controller Features in Multiprocessor Server Environment. In *3rd Workshop on Memory Performance Issues*, 2004.

[17] Kyle J. Nesbit, et al. Fair Queuing Memory Systems. In *Proc. of Annual International Symposium on Microarchitecture (MICRO)*, June 2006.

[18] K. Olukotun, B. A. Nayfeh , et. al. The case for a single-chip multiprocessor. In *Proc. of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct 1996.

[19] M. K. Qureshi and Y. N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches In *Proc. of Annual Int'l Symposium on Microarchitecture (MICRO)*, June 2006.

[20] N. Rafique, W.T. Lim and M. Thottethodi. Architectural Support for Operating System-Driven CMP Cache Management. In *Proc. of the 15th International Conference on Parallel Architectures and Compilation Technology (PACT 2006)*, Sept 2006.

[21] P. Ranganathan and N. Jouppi. Enterprise IT Trends and Implications on Architecture Research. In *Proc. of the 11th International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2005.

[22] S. Rixner, W. J. Dally, U. J. Kapasi, et al. Memory access scheduling. In *Proc. of the International Symposium on Computer Architecture (ISCA),* June 2000.

[23] M. Rosenblum and T. Garfinkel. Virtual Machine Monitors: Current Technology and Future Trends. *IEEE Transactions on Computers*, 2005.

[24] L. Sha, R. Rajkumar and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, Sept 1990.

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

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

[27] H. S. Stone, J. Turek, and J. L. Wolf. Optimal partitioning of cache memory. *IEEE Transactions on Computers*, Sept 1992.

[28] G. Suh, S. Devadas, and L. Rudolph. A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning. *In Proc. of International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2002.

[29] "Test TCP (TTCP) Benchmarking Tool", http://www.pcausa.com

[30] "TPC-C Design Document", http://www.tpc.org/tpcc/

[31] R. Uhlig, et al., "Intel Virtualization Technology," *IEEE Transactions on Computers*, 2005.

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

[33] T. Y. Yeh and G. Reinman. Fast and Fair: Data-stream Quality of Service. In *Proc. of International Conference of Compilers, Architecture and System For Embedded Systems (CASES)*, July 2004.

[34] L. Zhao, J. Moses, R. Iyer, et al. Architectural Evaluation of Large-Scale CMP Platforms using ManySim. In *Intel's Design & Test Technology Conference* (DTTC), Aug 2006.

[35] H. Zhang. Service Disciplines for Guaranteed Performance Service in Packet-switching Networks. In *Proc. of IEEE*, Oct. 1995.

[36] Z. Zhu and Z. Zhang. A Performance Comparison of DRAM Memory System Optimizations for SMT Processors. In *Proc, of the 11th International Symposium on High Performance Computer Architecture(HPCA)*, Feb 2005