A Framework for Providing Quality of Service in Chip Multi-Processors *

Fei Guo, Yan Solihin
Dept. of Electrical and Computer Engineering
North Carolina State University
{fguo, solihin}@ncsu.edu

Li Zhao, Ravishankar Iyer
System Technology Lab
Intel Corporation
{li.zhao, ravishankar.iyer}@intel.com

Abstract

The trends in enterprise IT toward service-oriented computing, server consolidation, and virtual computing point to a future in which workloads are becoming increasingly diverse in terms of performance, reliability, and availability requirements. It can be expected that more and more applications with diverse requirements will run on a CMP and share platform resources such as the lowest level cache and off-chip bandwidth. In this environment, it is desirable to have microarchitecture and software support that can provide a guarantee of a certain level of performance, which we refer to as performance Quality of Service.

In this paper, we investigated a framework that would be needed for a CMP to fully provide QoS. We found that the ability of a CMP to partition platform resources alone is not sufficient for fully providing QoS. We also need an appropriate way to specify a QoS target, and an admission control policy that accepts jobs only when their QoS targets can be satisfied. We also found that providing strict QoS often leads to a significant reduction in throughput due to resource fragmentation. We proposed throughput optimization techniques that include: (1) exploiting various QoS execution modes, and (2) a microarchitecture technique that steals excess resources from a job while still meeting its QoS target. We evaluated our QoS framework with a full system simulation of a 4-core CMP and a recent version of the Linux Operating System. We found that compared to an unoptimized scheme, the throughput can be improved by up to 47%, making the throughput significantly closer to a non-QoS CMP.

1 Introduction

Recently, Chip Multi-Processor (CMP) or multicore design has become the mainstream architecture of choice for major microprocessor makers. Compared to single core design, CMPs provide throughput improvement for multithreaded and multiprogrammed workloads. However, since many on-chip platform resources, such as the lowest level on-chip cache and off-chip bandwidth, are shared by all the processor cores, the performance and throughput of programs running on a CMP depend heavily on how these resources are allocated to them. The absence of resource allocation policies and the inability to partition these resources can lead to severe resource contention and result in a large performance variation for many applications [4, 9, 10, 11, 23, 26]. As the number of cores in a CMP increases, the degree of sharing of platform resources can be expected to increase and will only exacerbate the performance variability problem.

A large performance variation suffered by applications in current CMPs is ill-suited for future uses of CMPs considering recent trends in enterprise IT toward service-oriented computing, server consolidation, and virtualization. These trends point to a future in which CMPs will run a diverse set of applications that have diverse computing requirements [22]. In these environments, many applications will require a guarantee of a certain level of performance, and we refer to such a guarantee as performance Quality of Service (QoS). For example, in a service-oriented or utility computing environment, the utility computing provider may set up different service-level agreements (SLAs) to different clients that encapsulate guarantees in performance, reliability, manageability, and other metrics [16, 22]. Such an environment would require the server to be able to allocate platform resources proportionally to the level of the performance guarantee for each workload. For example, a job from a client with a “gold” SLA may be allocated more resources (e.g., larger cache sizes, higher processor count, and off-chip bandwidth) compared to jobs with standard SLAs. In virtualization, a virtual machine manager (VMM) hosts multiple virtual machines (VMs), where each VM runs a guest Operating System for various purposes ranging from regular to critical computations. It is beneficial if the VMM can allocate more platform resources to critical VMs and fewer resources to regular VMs. Finally, many transaction processing applications in a service-oriented computing domain would require a minimum level of real-time performance to be guaranteed. Overall, we believe that it is critical to support QoS in CMPs.

In recent studies, researchers have introduced frameworks in which applications specify their QoS targets, expressed in instructions per cycle (IPC) or resource performance (e.g. cache miss rates), while a resource manager dynamically partitions shared resources in order to meet

*This work is supported in part by the National Science Foundation through grant CNS-0406306 and CCF-0347425, and by gifts from Intel.
each application’s QoS target [9, 10, 11, 15, 19]. Unfortunately, these frameworks are insufficient if one wants to fully provide QoS in CMPs. Figure 1 shows an example in which multiple jobs (each job runs the SPEC2006 benchmark bzip2) run on a 4-core CMP with private L1 caches but a shared L2 cache. Let us assume that each job’s QoS target is to reach an IPC of at least 0.25, which is \( \frac{2}{3} \) of its IPC when it runs alone. If the resource manager tries to satisfy the QoS targets of all jobs, it will equally divide the L2 cache among all instances of \texttt{bzip2}. However, from this figure, we can see that while the jobs’ QoS targets are met when only two jobs run simultaneously, they are not met when three or four jobs run in the CMP. There are two major reasons why such frameworks fail in meeting the QoS targets of the jobs. First, the CMP does not have the ability to check whether its available resources are sufficient to satisfy a job’s IPC target or the amount of resources needed by the job to meet its IPC target. Secondly, the lack of an admission control policy means that jobs are always accepted and run even when their QoS targets cannot be met.

![Figure 1. The IPC of different numbers of instances of bzip2 running on a 4-core CMP with a 32-KB L1 cache per core, and a 2MB L2 cache shared by all cores. The full evaluation setup can be found in Section 6.](image)

**Contributions.** In this paper, we investigate a framework that is needed to fully provide QoS in a CMP system. First, we study how a QoS target should be specified in order to enable the system to compare its available computation capacity against the requested computation capacity. We found that popular metrics (IPC and miss rate) are not suitable for this purpose, while capacity specification (e.g., cache size) naturally makes it easier to compare available computation capacity with demanded capacity. We then add an admission control policy to ensure that jobs are accepted only when their QoS targets can be satisfied.

Secondly, we introduce three QoS execution modes, as a way for jobs to specify how flexible they are with regard to their QoS targets. These execution modes are needed to match users’ diverse requirements for their workloads as well as to provide a means for the system to boost throughput.

We use our framework to enable QoS in a CMP and apply it for managing processor cores and the shared L2 cache resources. We found that providing a strict QoS guarantee usually comes at a cost of significant reduction in system throughput due to jobs overspecifying their QoS targets, resulting in fragmentation of various resources in the system. We propose and investigate two techniques to recover the lost throughput. The first is a microarchitecture technique that we refer to as resource stealing, which steals cache capacity from a job which may have excess resources while still meeting the job’s QoS target. The second technique speculatively downgrades a job’s QoS execution mode in order to boost the overall throughput.

We evaluate the proposed schemes and mechanisms on a 4-core CMP machine model based on Simics, with a recent version of the Linux Operating System, and workloads constructed by using SPEC2006 benchmarks. We found that through a combination of appropriate QoS target specification, admission control, and execution modes, a CMP can ensure that all accepted jobs have their QoS targets satisfied. In addition, we found that the proposed QoS execution modes and resource stealing mechanism are effective in improving throughput without violating jobs’ QoS targets. They achieve throughput improvement between 13% and 47%, making the throughput significantly closer to a non-QoS CMP.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 presents QoS target specification and execution modes. Section 4 presents our microarchitecture technique for improving throughput. Section 5 discusses the implementation of an admission control policy. Section 6 presents the evaluation setup, while Section 7 presents and discusses the evaluation results. Finally, Section 8 summarizes the findings.

## 2 Related Work

In a CMP system, some platform resources, such as the off-chip bandwidth and the lowest level on-chip cache, are typically shared among cores. With the increasing number of cores on a chip (possibly to more than one hundred cores by 2015 [1]), the contention for these critical shared resources suffered by applications running simultaneously in different cores will increase significantly and needs to be carefully managed.

Some studies that address the management of shared resources have focused on improving the overall throughput or fairness of the CMP. Suh et al. proposed a cache partitioning policy that minimizes the total number of cache misses [26]. Kim et al. proposed a cache partitioning policy that optimizes for uniform slowdown (fairness) to applications that share the cache [23], while Qureshi et al. proposed utility-based cache partitioning [18]. Hsu et al. studied the impact of various optimization goals in guiding how cache partitions are allocated in a CMP architecture [9]. The optimization goals include maximizing the overall performance metric (e.g., IPC or miss rate) and the overall fairness. Cho and Jin proposed an OS-level page allocation algorithm in a shared L2 non-uniform cache architecture for future many-core processors to reduce the cache...
access latency, on-chip network traffic and power consumption [6]. Chang and Sohi proposed a cooperative cache partitioning algorithm that optimizes for several metrics, such as the sum of per-thread slowdowns as well as the harmonic mean of per-thread speedups over an equal-partition cache baseline [5]. While these studies seek to mitigate the impact of contention and optimize for an overall goal, they do not provide QoS to individual applications.

Some recent studies have recognized the need for CMPs to have QoS-enabling features to provide differentiated services to various applications. For example, Iyer described the need for a priority-based QoS framework in SMP architectures in which a job can specify whether it should be run with a high or low priority, and resource allocation is guided by job priorities [10]. Rafique et al. proposed an architecture support and Operating System (OS) interface that allows OS-level cache partitioning, in which an application is prevented from occupying more than a certain fraction (quota) of the cache [19]. Nesbit et al. proposed a Virtual Private Cache (VPC) that combines the resource allocation policies for caches and the memory controller using a fair queuing algorithm [15]. VPC provides an abstraction of private caches through partitioning the shared cache into per-core partitions, while resource allocation policies ensure that the IPC achieved is equal to that of real private caches. Iyer et al. [11] proposed several priority-based resource management policies as well as a QoS-aware cache and memory architecture. Individual applications can specify their own QoS target (e.g. IPC, miss rate, cache space) and the hardware dynamically adjusts cache partition sizes to meet their QoS targets. We note that while all above studies mention QoS, they do not associate QoS with the notion of performance guarantee. While both priorities and cache quota correlate with performance and help the system to favor one application over another in allocating resources, they do not automatically provide QoS guarantees to the applications.

In some QoS models, individual applications can specify an acceptable performance level expressed in IPC. The system then translates the IPC into the minimum resources required to achieve it, through profiling a thread at run-time and recording how the thread’s IPC changes as the amount of resources are varied [12, 29]. The fact that the studies above require a greedy search in the resource allocation space illustrates the difficulty of using IPC as a QoS target. We believe that to really provide QoS, the CMP must have an ability to easily compare available and demanded computation capacity, and such an ability is a critical component that enables the construction of an admission control policy.

Finally, some of the concepts in our framework, such as the resource reservation and admission control, are borrowed from the real-time system domain [2, 3, 20]. However, while in traditional real-time systems the operating system and processor architecture are structured to suit the needs of real-time constraints, we seek to provide QoS in CMP-based general purpose servers with a largely unmodified OS, processor architecture, and memory hierarchy.

3 QoS Target Specification and Execution Modes

This section presents the assumptions of our working environment (Section 3.1), QoS goals and metrics for specifying a QoS target (Section 3.2) and the proposed QoS execution modes (Section 3.3).

3.1 Definitions and Assumptions

We refer to a job as the unit of aperiodic computation task that has its own QoS target. A job may consist of a thread, an application, or a group of applications. In this paper, we limit our study by associating an instance of a single-threaded application as a job. We define computation capacity as the resources that can be used for providing performance. Basically, a job’s QoS target is computation capacity demand, while available resources in the server are computation capacity supply.

![Figure 2. The assumed working environment.](image)

We assume a server platform consisting of CMP nodes as shown in Figure 2. The server has a global admission controller (GAC) which decides whether to accept or reject a newly arriving job submitted by user. To achieve this, the global admission controller probes each CMP node’s local/per-CMP admission controller (LAC) to find which CMP node can accept the job and satisfy its QoS target. When the GAC cannot find any CMP node that can accept the job, it rejects this job or negotiates with the user for another acceptable QoS target. A comprehensive discussion of the GAC is beyond the scope of this paper. In this paper, only the LAC is considered as a component of our QoS framework because it has a direct interaction with microarchitecture resources.

3.2 Specifying QoS Target

Earlier we have argued that for a CMP to fully provide QoS to an incoming job, it must have available computation capacity in excess of the capacity demanded by the job. We refer to the units of target as the units in which a QoS target is specified, and units of capacity as the units in which computation capacity in the CMP is expressed. Before continuing our discussion, it is helpful to define one term.

**Definition 1** A QoS target is convertible if its units of target can be converted into units of computation capacity. In order for a CMP to really provide QoS, two conditions must be met. The first condition is that a QoS target must be convertible, which allows the system to easily compare the available computation capacity with demanded capacity.
The second condition is that a job should be accepted only if its QoS target can be satisfied. Convertibility and satisfiability checking are the basis for constructing an admission control policy to ensure that the QoS targets of all accepted jobs can be met.

One way to provide QoS in a CMP system is to model it after a traditional real-time system in which the QoS target of a job is specified by its deadline. In a real-time system, deadline convertibility can be achieved through Worst-Case Execution Time (WCET) analysis which determines the maximum execution time of a job by taking into account the maximum path length of the code and maximum latencies that can occur in the architecture. Unfortunately, in traditional real-time systems the operating system and processor architecture are often structured to suit the needs of real-time constraints, such as by restricting out-of-order execution, dynamic branch prediction, and a complex memory hierarchy. Our goal is to provide QoS in general purpose servers with a largely unmodified OS, processor architecture, and memory hierarchy. Furthermore, in a server environment, jobs often have unpredictable arrivals, dynamic and input-dependent behavior, and may not have meaningful deadlines. Hence, unlike in traditional real-time systems, we cannot use deadlines as the primary QoS target.

In [11], Iyer et al. proposed three types of QoS targets. The first is Resource Usage Metrics (RUM), which specify the amount of resources needed by the application, such as the processor count, cache size, and bandwidth rate. The second is Resource Performance Metrics (RPM), which specify the performance of specific resources being used, for example cache miss rates. The final is Overall Performance Metrics (OPM), which specify the overall throughput of the program, expressed in IPC. Most prior studies in architecture support for QoS assume that the QoS target is expressed in IPC. However, we believe that IPC is not suitable to specify a QoS target because IPC is not easily convertible. A CMP system cannot easily determine how much IPC it can provide for a particular job (unless it uses an elaborate performance model). Furthermore, it also cannot easily determine the amount of platform resources that are needed to achieve a target IPC. Similarly, a CMP cannot easily determine what miss rate it can provide to a particular job or the amount of resources needed in order to provide a given miss rate. In fact, in addition to being non-convertible, it may be hard to check whether particular RPM and OPM values are not ill-defined, i.e. they cannot be satisfied no matter how many resources are allocated. As a result, we believe that OPM and RPM are not suitable for a CMP system to fully provide QoS.

In contrast to RPM and OPM, RUM are easily convertible if a CMP is equipped with relatively simple hardware that tracks the current allocation of platform resources for different cores. For example, with RUM, an incoming job’s request in terms of the amount of cache capacity it needs (demand) can be compared trivially against the amount of cache capacity that has not been allocated yet (supply). This leads to the ease of constructing an admission control policy. An additional benefit of using RUM is that such metrics are already used in batch job systems. For example, in the Lsbatch batch job system [28], a job can specify its requirements in terms of the number of processors, memory size, disk space size, and the maximum wall clock time it would run. Hence, RUM has an advantage of being time-tested and has a high familiarity with users.

In the context of CMPs, RUM must extend beyond resources that are specified in traditional batch job systems, for example by including the shared cache capacity and off-chip bandwidth rate. In this work, we focus on the shared L2 cache capacity and processor core resources in the QoS target specification. We acknowledge that a complete QoS target would include off-chip bandwidth rate, main memory size, network bandwidth, disk size, and other resources. However, we note that those resources are not specific to CMP design or do not contribute as strongly to the performance variation of a job compared to the L2 cache capacity and processor core. Hence, we leave them for future work.

Optionally, a QoS target may include a timeslot resource, which can be specified through a maximum wall-clock time which indicates the size of the timeslot, and a deadline which indicates the latest expected completion of the timeslot. Maximum wall-clock time is a concept borrowed from batch job systems [28]. It specifies the maximum amount of time that a job should be allowed to run assuming it gets all its requested resources. The maximum wall-clock time is different from WCET in a real-time system in that it does not need to be a safe execution time upper bound. Embedded in it is the users’ expectation that a job may be terminated if it runs longer than its maximum wall-clock time. Another reason why timeslot resource specification is optional is that jobs do not necessarily have a maximum wall-clock time or a deadline. Long-running applications, OS daemons, or other legacy applications may not specify timeslot resources, and in this case resources will be allocated to them for their entire lifetime.

In order to make it easier for users to determine an appropriate QoS target, the system may provide several preset RUM targets that users can choose from. Similar preset targets have been employed in many batch job systems. For example, a job can choose one of large, medium, or small configurations. Each configuration comes with preset memory size, maximum processor count, and maximum wall-clock time. However, while preset QoS targets could greatly simplify QoS target selection for users, they may also exacerbate QoS overspecification, a situation in which a job needs less resources than what it specifies. This leads to resource fragmentation that will reduce overall throughput. We will address how to recover from resource fragmentation in the following sections.

3.3 QoS Execution Modes

Besides QoS target specification, another important component of our QoS framework is how strictly the QoS target must be followed. Similar to the postal delivery sys-
tem that allows various strictness levels in delivery times, it may be desirable for a utility computing server to provide various strictness levels in meeting the QoS target. To achieve this variety, we propose the following execution modes:

1. **Strict**: The Strict execution mode may be used by jobs that have rigid requirements for a minimum throughput (implied by the RUM) and deadline. To meet a Strict job’s QoS target, the requested resources and timeslot must be strictly reserved.

2. **Elastic(X)**: The Elastic(X) execution mode can be used by jobs that have a rigid deadline requirement but can tolerate some deviation of throughput compared to that implied by the amount of resources requested in RUM. The deviation is such that the reduction of throughput (slowdown) is not more than X% compared to the case in which the resources are reserved (i.e., in the Strict mode).

3. **Opportunistic**: The Opportunistic mode may be used by jobs that do not have rigid throughput and deadline requirements. For example, users may use the Opportunistic mode for jobs whose deadlines are still far away.

Mode downgrade can also be performed transparently by the system as long as the old and new modes are interchangeable. We define two modes as interchangeable if they can be used to guarantee completion of a job by the same deadline, and throughput variation can be tolerated by the job. Suppose we have a Strict job with deadline of $td$ and maximum wall-clock time of $tw$, arriving at time $ta$. We note that there is a time slack of $(td - ta) - tw$. This amount of slack means that the job can be downgraded as an Elastic($\frac{(td - ta) - tw}{tw}$) job while still meeting its deadline. Additionally, it can also be downgraded to the Opportunistic mode for $(td - ta) - tw$ amount of time, but if it has not completed by time $td - tw$, it needs to be switched back to the Strict mode. We refer to these as automatic mode downgrade.

### 3.4 The Impact of Execution Mode Downgrade

In this section, we examine the impact of manual and automatic mode downgrade. Before we continue the discussion, it is helpful to distinguish two factors that contribute to sub-optimal throughput when there are only Strict jobs. The first is external resource fragmentation which refers to idle resources that are not allocated to any jobs. Typically, external resource fragmentation occurs when there are not sufficient remaining available resources to accept a new job. The second factor is internal resource fragmentation which is caused by a job not using all resources that are allocated to it.

First, let us consider the impact of manual mode downgrade as illustrated in Figure 3. In the figure, each bar represents the time each job takes to complete its computation, the x-axis represents the time duration and the y-axis represents different accepted jobs. In this example, we assume that jobs are sequentially submitted to the system, and the first six accepted jobs are shown in the figure. Assume that each job requires 40% of the shared cache size in order to complete in $T$ time. The deadline of each job is assumed to be $1.5T$ from the time when the job is accepted. If there are six Strict jobs (Figure 3(a)), at most two jobs can be executed simultaneously since there is not enough cache space to run more than two jobs simultaneously. The external resource fragmentation includes two idle cores in a 4-core CMP and 20% of the L2 cache capacity. The CMP takes $3T$ to finish all six jobs and all of them meet their deadlines. However, if users manually downgrade the third and sixth jobs to Opportunistic jobs (Figure 3(b)), the system can accept and run more jobs simultaneously and reduce external resource fragmentation. Although the third and the sixth jobs run slower, the overall throughput is improved as it only takes slightly more than $2.5T$ to complete all six jobs. If users also manually downgrade the second and fifth jobs to Elastic(X) jobs, the system can employ the resource stealing technique (Section 4) to discover unused cache capacity allocated to Elastic(X) jobs and reallocate them to Opportunistic jobs. This results in the third and sixth jobs

\[1\text{The concept of Elastic mode is similar to the one in Buttazzo et al. [3]. However, while in [3] it only applies to periodic jobs, we define the elasticity in terms of slowdown, hence it is applicable to aperiodic jobs as well.} \]
With manual mode downgrade, when a Strict job is downgraded to Elastic(X), its unused resources are re-allocated to Opportunistic jobs. However, since the job may be slowed down by up to X% in order to guarantee meeting its deadline, the job needs to reserve resources for a longer time duration of $tw \times (1 + X)$ (versus $tw$ if it remains a Strict job). Since the same amount of resources are reserved for a longer time, the ability of the CMP to accept future jobs may be reduced, which in turn may reduce future throughput. Consequently, manual downgrade of a Strict job to Elastic(X) may reduce throughput if it is not accompanied by a throughput increase due to Opportunistic jobs benefiting from reallocation of excess resources. Overall, we can expect throughput improvement to be higher when there are both Elastic(X) and Opportunistic jobs to complement Strict jobs, compared to when there are only Opportunistic jobs to complement Strict jobs.

With automatic mode downgrade, an additional impact occurs when a Strict job is downgraded to Opportunistic mode. In contrast to the manual mode downgrade in which an Opportunistic job does not reserve any resources, with automatic mode downgrade the resources requested by the job still need to be reserved for the length of its maximum wall-clock time. The job can only be run in Opportunistic mode before it meets its reserved timeslot, by which time it has to switch back to Strict mode in order to ensure that its deadline is met. When a job completes before it meets its reserved timeslot, the reserved resources can be reclaimed to allow new jobs to be accepted and future throughput to be improved. As a result, the reserved timeslot needs to be placed as far away as possible in order to increase the probability that the job completes before the reserved timeslot is encountered. Finally, we do not consider automatically downgrading a Strict job to Elastic(X) job because the Elastic(X) mode reserves resources longer than the original Strict mode, which is likely to be detrimental to throughput.

Comparing manual and automatic mode downgrade, if the manually downgraded Opportunistic jobs do not suffer from too much slowdown due to lack of resources, it can be expected that manual downgrade would achieve a higher throughput than automatic downgrade since the latter still relies on resource reservation, which may reduce job admission rate. However, the automatic mode downgrade is still useful because it does not rely on users’ willingness to downgrade their jobs to weaker modes.

4 Resource Stealing

In this section, we will discuss resource stealing, a key technique that supports the proposed Elastic(X) execution mode.

4.1 Managing Cache Capacity Partitions

In order to track and control the shared cache allocation across cores, we need to employ a cache partitioning scheme. Cache partitioning can be achieved through a global approach or per-set approach. In the global approach, a modified LRU policy [27] keeps a global counter that tracks the number of cache blocks currently allocated to each core, and another counter that records the target number of cache blocks that should be allocated to each core. On a cache miss, the victim block is chosen from the blocks that belong to the core which has more allocated blocks than its target number of cache blocks. This process is repeated until each core reaches its target cache allocation. The number of blocks allocated to a core in different sets vary, but the sum of them over all sets would match the target allocation.

We note that while this global approach is relatively simple to implement, it has a drawback that the distribution of blocks allocated to an application in different cache sets varies across different runs as it is affected by other applications that run simultaneously. This variation of allocation, especially when it occurs in heavily-used sets sometimes introduces a large variation in miss rates and performance for
the same application across different runs. As a result, we do not use it in our QoS framework.

The per-set cache partitioning algorithm in [10, 15] is a finer-grain version of the modified LRU replacement policy proposed by Suh et al. [27]. In [10, 15], each core is assigned a target allocation counter that records the number of cache ways that should be assigned to it. Each set in the cache also has per-set counter to track the number of blocks in the set that are currently allocated to the core. When a core suffers a cache miss on a set, the core’s per-set counter is compared against its target allocation counter. If the per-set counter has a lower value, a block that belongs to one of the over-allocated cores is selected as the victim. Otherwise, a block that belongs to the core itself is selected as the victim. Over time, the number of blocks allocated to a core will be the same over all sets. Consequently, over different runs, the same job with the same cache space allocation will perform more uniformly. This is a desirable factor in a system that tries to provide QoS.

Our cache partitioning scheme is based on the fine-grain approach in [10, 15], but we adapt it to our QoS framework. In our modified version, the selection of the victim block for a cache miss also depends on the execution mode of the job that the victim belongs to. On each cache miss, if there is more than one over-allocated core, the victim is first selected from an over-allocated Strict or Elastic(X) job (if there is any). Otherwise, the LRU block among the blocks from Opportunistic jobs is selected as the victim. The reason why over-allocated Strict/Elastic(X) jobs are given a higher priority for victim selection is that we would like to accelerate the cores running these jobs in converging to their target allocations, and reallocate the excess cache capacity from Elastic(X) jobs to Opportunistic jobs as fast as possible.

### 4.2 Criteria for Resource Stealing

In Section 3.3, the Elastic(X) execution mode was proposed in order to enable the system to remove excess cache capacity allocated to a job due to QoS overspecification, and X specifies the maximum slowdown (slack) that is acceptable to the user. When applying resource stealing, it is difficult to accurately measure how much CPI increases when we employ resource stealing versus when we do not. Hence, we need a more measurable metric.

First, we note that components of CPI are additive, i.e. the overall CPI is the sum of CPI assuming an infinite cache and additional CPI when cache misses are considered [7, 13]. Specifically, assuming a system with two levels of caches, we use Luo’s model [13] that expresses CPI as:

\[
\text{CPI} = \text{CPI}_{L1,\infty} + h_2 \times t_2 + h_m \times t_m
\]

where \(\text{CPI}_{L1,\infty}\) indicates the CPI of the program when the L1 cache has infinite size, \(h_2\) and \(h_m\) indicate the number of L2 accesses per instruction and the number of L2 misses per instruction respectively, and \(t_2\) and \(t_m\) indicate the penalty (in number of cycles) of an L2 access and an L2 miss respectively. With resource stealing, we attempt to steal L2 cache capacity from an Elastic(X) job and reallocate it to Opportunistic jobs. The effect of reduced L2 cache size for the Elastic(X) job is a higher L2 cache miss rate (hence higher \(h_m\)), but other variables would remain roughly unchanged \(^2\). Since \(h_m \times t_m\) is only one component of the CPI, and all other components have positive (at least non-negative) values, an increase of X% in \(h_m\) would result in a less than X% increase in CPI. We exploit this observation to guide our resource stealing algorithm such that it removes cache capacity from an Elastic(X) job but without increasing the job’s L2 cache miss rate by more than X%.

Note that our criteria of allowing the L2 cache miss rate to increase by not more than X% is likely to be conservative, i.e. the increase in CPI is smaller than X%. However, we note that even a conservative criteria and a small X value can be sufficient to recover unused cache capacity from a job. For example, if a job has a working set which is much smaller than its L2 cache partition size, much of its excess L2 cache capacity can be stolen before the job’s L2 cache miss rate increases by X%. Finally, monitoring the L2 miss rate increase is achievable with relatively simple hardware modification which will be described in the next section. Therefore, in this paper, we choose to use L2 miss rate as the metric to guide the resource stealing mechanism.

### 4.3 Microarchitecture Support

In order to monitor the miss rate increase due to partition changes, we need a mechanism to dynamically obtain the miss rates for both the reduced and original partition cases. To achieve this, we use a straightforward method that utilizes an additional duplicate cache tag array [17, 25] that keeps track of what blocks the cache would have if resource stealing had not been applied, while the main cache tag array keeps track of the actual cache content. To reduce the storage overhead of keeping the duplicate tag array, we employ set sampling [17, 18], in which only a few sets are augmented with duplicate tags and profiled to infer the global cache behavior. We let the stream of all L2 cache accesses be visible to both tag arrays so that only their numbers of misses differ.

When an Elastic(X) job runs, the resource stealing algorithm is activated. The algorithm reduces the Elastic(X) job’s partition size by one way. This “stolen” way is reallocated to one of the Opportunistic jobs. The target allocation counters for the Elastic(X) job and the Opportunistic job are updated, which allows the cache to converge to the

\(^2\)The number of L2 accesses per instruction (\(h_2\)), the latency of an L2 cache access (\(t_2\)), and the CPI of infinite L1 cache (\(\text{CPI}_{L1,\infty}\)) are mostly affected by physical L1 and L2 cache organization, and not affected much by L2 cache partition sizes. The latency of an L2 miss (\(t_m\)) may be affected as memory bus contention increases due to the increase in the number of L2 cache misses. However, this can be mitigated by prioritizing memory requests from Elastic(X) jobs over those from Opportunistic jobs. In addition, we can monitor bus utilization and disable resource stealing when bus saturation is reached. According to Little’s Law [8], prior to saturation, queueing delay is roughly constant.
new partitions over time. In the meantime, the duplicate tags keep track of the total number of misses for the Elastic(X) job without resource stealing. If the extra number of misses in the main tags reaches or exceeds X% compared to that in the duplicate tags, then the resource stealing has potentially caused the Elastic(X) job to slowdown by more than X%, so the resource stealing is canceled and all the stolen ways are returned to the Elastic(X) job. Otherwise, in the next time interval we steal another way from the Elastic(X) job. Note that while repartitioning occurs at periodic intervals, the number of misses in the duplicate tag array and main tag array are not reset at each interval. This ensures that the total number of misses since the start of the Elastic(X) job does not increase by more than X% due to resource stealing.

5 Local Admission Controller Implementation

For the Local Admission Controller (LAC), we implement a First Come, First Served (FCFS) scheduling algorithm using a basic resource allocation model from [21]. The LAC maintains a list of vectors that encode processor core and cache capacity resources and the timeslots in which they are available. A job specifies its QoS target through a resource request vector which encodes the amount of resources needed for each resource type, and for how long (based on the maximum wall-clock time) they are needed. For a Strict or Elastic(X) job, the LAC tries to find the earliest timeslot in which this vector can be fit before the job’s deadline. If such a timeslot is found, the job is accepted and the resources are reserved in that timeslot. An Opportunistic job is always accepted if there are spare resources not already taken up by Strict or Elastic(X) jobs.

To keep the OS thread scheduler unchanged, we implement the LAC as a user-level program. The LAC has a scheduler queue to store accepted jobs and manage their resource and timeslot reservations. Once it is time to start a job, the job is submitted to the OS. If the job is an Elastic(X) job, resource stealing is also activated. To avoid timesharing from violating a Strict or Elastic(X) job’s deadline, the LAC pins only one such job to one processor core. However, the LAC may pin multiple Opportunistic jobs on a core that is not already assigned to Strict or Elastic(X) jobs.

6 Evaluation Methodology

Simulation Environment. To evaluate the proposed QoS framework, we use a full-system simulator based on Simics [14] to model a 4-core CMP node with Fedora Core 4 Linux as the operating system. Each core is an in-order processor with a 2GHz clock frequency. Each core has a private L1 instruction and a private L1 data cache with a 32KB size, 4-way associativity, 64-byte block size, LRU replacement policy, write back policy, and 2-cycle access time. The L2 cache is shared by all four cores, and is a unified cache with a 2MB size, 16-way associativity, 64-byte block size, modified LRU replacement policy (Section 4.1), write-back policy, and 10-cycle access time. The main memory is 4GB in size with a 300-cycle access time. The peak bandwidth to the main memory is 6.4GB/s. The simulator ignores the impact of page mapping by assuming each job is allocated contiguous physical memory.

Resource Stealing. We employ set sampling to implement the duplicate tags used in resource stealing. The duplicate tags only cover \(\frac{1}{3}\) of the total number of sets, which is conservative compared to 32 sets used in [18]. Every 8th set is sampled. The interval for triggering cache repartitioning is 2 million instructions of the Elastic(X) job.

Individual Job. To evaluate our proposed QoS execution modes and resource stealing technique, we choose fifteen C/C++ benchmarks from SPEC2006 benchmark suite [24]: gcc, bzip2, perl, gobmk, mcf, hmmer, sjeng, libquantum, h264ref, milc, astar, namd, soplex, povray, sphinx. We use the ref input sets for all benchmarks except for milc and soplex which use the train input sets because their ref input sets require too much memory. For each benchmark, we first inspect its source code and identify its initialization routines. Then we skip the initialization routines and simulate the next 200 million instructions.

To reduce the number of benchmarks that we need to evaluate, we classify the benchmarks according to their cache space sensitivity. For each benchmark, we calculate the CPI increase when we reduce its L2 cache allocation from 7 ways to 1 way, and from 7 ways to 4 ways. Then we plot them in a two-dimensional space (Figure 4).

![Figure 4. The sensitivity of each benchmark to cache capacity.](image)

From the figure, we can roughly classify the fifteen benchmarks into three groups based on how sensitive they are to the allocated cache space: highly sensitive (Group 1), moderately sensitive (Group 2), and insensitive (Group 3). Similar classification methods can be found in [5, 18]. Highly sensitive benchmarks are ideal beneficiaries of resource stealing, whereas insensitive benchmarks are ideal donors for resource stealing. From each group, we choose one representative benchmark: bzip2 from Group 1, hmmer from Group 2 and gobmk from Group 3. The L2 miss rates
and L2 misses per instruction of these benchmarks when they are allocated 7 cache ways are listed in Table 1.

**Table 1. The benchmarks used as individual jobs in the evaluation.**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Input Set</th>
<th>L2 Miss Rate</th>
<th>L2 Misses Per Instruction</th>
<th>Number of Skipped Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2</td>
<td>ref.chicken</td>
<td>20%</td>
<td>0.0055</td>
<td>315M</td>
</tr>
<tr>
<td>hmmer</td>
<td>ref.retro</td>
<td>17%</td>
<td>0.001</td>
<td>0.3M</td>
</tr>
<tr>
<td>gobmk</td>
<td>ref.angs</td>
<td>24%</td>
<td>0.004</td>
<td>267M</td>
</tr>
</tbody>
</table>

**Workload Composition.** We construct 10-job workloads and measure the wall clock time to complete all ten jobs. Each job requests a processor core and L2 cache capacity of 896KB (7 ways in the 16-way L2 cache). We assume incoming jobs with Poisson arrival, with an inter-arrival time that assumes full computation capacity utilization of a 128-CMP server. Specifically, on a 4-core CMP, in one job’s wall-clock time, there are on average 4 × 128 new jobs that arrive and probe the CMP’s Local Admission Controller. Job deadlines are assigned as follows. We pseudo-randomly set 50% of them with a tight deadline \((td − ta = 1.05 × tw)\), 30% with a moderate deadline \((td − ta = 2 × tw)\) and 20% with a relaxed deadline \((td − ta = 3 × tw)\).

To evaluate the various execution modes, we use five configurations shown in Table 2. The base configuration is **All-Strict**. The **EqualPart** configuration mimics the Virtual Private Cache [15] by equally partitioning the cache capacity among all cores, but without an admission controller and bandwidth partitioning.

**Table 2. Execution modes configurations.**

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Percentage of Jobs in Various Execution Modes</th>
</tr>
</thead>
<tbody>
<tr>
<td>All-Strict</td>
<td>100% Strict</td>
</tr>
<tr>
<td>Hybrid-1</td>
<td>70% Strict + 30% Opportunistic</td>
</tr>
<tr>
<td>Hybrid-2</td>
<td>40% Strict + 30% Elastic(5%) + 30% Opportunistic</td>
</tr>
<tr>
<td>All-Strict+AutoDown</td>
<td>100% Strict, jobs with moderate or relaxed deadlines are automatically downgraded.</td>
</tr>
<tr>
<td>EqualPart</td>
<td>No admission control, default Linux job scheduling. L2 cache is equally partitioned among cores.</td>
</tr>
</tbody>
</table>

The 10-job workloads are constructed in two ways. First, we use instances of the same benchmark, i.e. 10 instances of *bzip2*, 10 instances of *hmmer*, and 10 instances of *gobmk*. In addition, we also construct two mixed-benchmark workloads shown in Table 3. Mix-1 is a favorable workload for our framework since cache-sensitive *bzip2* is the recipient of resource stealing, while cache-insensitive *gobmk* is the donor of resource stealing. In contrast, Mix-2 is not favorable to our resource stealing technique.

**Table 3. Mixed-Benchmark Workloads.**

<table>
<thead>
<tr>
<th>Type</th>
<th>Workload Composition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mix-1</td>
<td><em>hmmer</em> (Strict), <em>gobmk</em> (Elastic(5%)) and <em>bzip2</em> (Opportunistic)</td>
</tr>
<tr>
<td>Mix-2</td>
<td><em>hmmer</em> (Strict), <em>bzip2</em> (Elastic(5%)) and <em>gobmk</em> (Opportunistic)</td>
</tr>
</tbody>
</table>

**7 Evaluation Results**

In this section, we will present our experiment results, including the impact of various execution modes (Section 7.1), the impact of automatic mode downgrade (Section 7.2), evaluation of the resource stealing technique (Section 7.3), evaluation of mixed workloads (Section 7.4) and characterization of the Local Admission Controller (Section 7.5).

**7.1 Impact of Various Execution Modes**

Let deadline hit rate refer to the fraction of jobs that meet their deadlines. Figure 5(a) shows the deadline hit rates for various configurations (Table 2) with workloads consisting of ten identical instances of a single benchmark. For our QoS framework, the deadline hit rate is only computed for Strict and Elastic(X) jobs. The figure shows a consistent result of 100% deadline hit rate in our QoS framework. In contrast, the deadline hit rates are only 50%, 10% and 20% in EqualPart for *gobmk*, *hmmer* and *bzip2*, respectively. This is because in EqualPart, the lack of an admission controller causes jobs to be accepted continuously despite the fact that the CMP no longer has sufficient computation capacity to meet the jobs’ deadlines. This observation reinforces the argument that partitioning the cache capacity among cores alone cannot fully provide QoS. Only after incorporating an admission control policy and using appropriate QoS targets can jobs reliably meet their deadlines.

**Figure 5. Comparing QoS and throughput of different configurations.**

Figure 5(b) compares the job throughput of the single-benchmark workloads on various configurations, measured as the total wall-clock time to complete the first ten accepted jobs. The throughput is normalized to the All-Strict.
case. Comparing All-Strict and EqualPart, the figure clearly shows that in all workloads, providing strict QoS comes at a cost of significantly lower job throughput, e.g. the throughput in EqualPart is higher by 64%, 54% and 25% for gobmk, hmmer and bzip2, respectively. There are several factors that cause this result. The first is various external resource fragmentation such as the processor cores (only two jobs run simultaneously, leaving two idle cores) and cache capacity (only 14 ways in the 16-way L2 cache are allocated). The second is the internal cache capacity fragmentation if the jobs do not fully utilize their allocated cache partitions, which is especially the case for cache-insensitive benchmarks such as gobmk. In contrast, the EqualPart configuration does not suffer from any external resource fragmentation and suffers little from internal resource fragmentation. As a result, the more cache sensitive the jobs in a workload are, the smaller throughput reduction the workload suffers from when QoS is provided.

In Hybrid-1, the existence of Opportunistic jobs effectively removes external core and cache capacity fragmentation, resulting in a significant 25% improvement in throughput for all workloads. However, since internal cache capacity fragmentation still exists (especially in gobmk and hmmer), the throughput is still lower than that in EqualPart. In Hybrid-2, the existence of Elastic(X) jobs conceptually allows some cache capacity to be stolen and reallocated to Opportunistic jobs, reducing some internal cache capacity fragmentation. However, the figure shows that the throughputs of all the workloads in Hybrid-2 are almost the same as those in Hybrid-1. Upon closer analysis, we find that indeed Opportunistic jobs complete sooner but the overall job throughput is largely unaffected because the tenth accepted job is a Strict job and it completes at almost the same time in Hybrid-1 and Hybrid-2 for all workloads. Note that internal cache capacity fragmentation still exists in Strict jobs (70% in Hybrid-1 and 40% in Hybrid-2) and it cannot be removed without the risk of reducing throughput and missing deadlines. This prevents Hybrid-1 and Hybrid-2 from matching the throughput of EqualPart.

Finally, in All-Strict+AutoDown, automatic mode downgrade is applied to Strict jobs to remove some of the resource fragmentation, resulting in throughput improvements of 39%, 20% and 13% for gobmk, hmmer and bzip2 respectively. Again, due to their higher internal cache capacity fragmentation, the throughput improvement for gobmk and hmmer are higher than that of bzip2. The throughput improvement over All-Strict is substantial, especially considering that the optimization does not require users to downgrade the modes of their jobs. However, we note that since we apply automatic mode downgrade only to jobs whose deadlines are relaxed or moderate (Table 2), only half of the jobs benefit from automatic mode downgrade. We can expect that if more jobs have moderate or relaxed deadlines, the throughput will be closer to that of EqualPart.

Overall, we conclude that while providing QoS imposes a severe penalty on throughput, a significant part of the throughput can be recovered through several schemes: providing various execution modes to users, and/or employing automatic mode downgrade transparently.

To further analyze the impact of various execution modes on wall-clock time, Figure 6 shows the average wall-clock time of jobs in different configurations for the single-benchmark workload of bzip2. The candle on each bar shows the range between the minimum and the maximum wall-clock time. The figure shows that with our QoS framework, Strict jobs in all configurations except All Strict+AutoDown have short and almost-constant wall-clock times. The Elastic(X) jobs in Hybrid-2 run slightly longer than Strict jobs because of resource stealing. However, their wall-clock time is still absent of much variation. As expected, because resources are not reserved, Opportunistic jobs in Hybrid-1 and Hybrid-2 have a higher average and variation of wall-clock time compared to Strict jobs. Opportunistic jobs in Hybrid-2 have lower wall-clock time compared to those in Hybrid-1, thanks to the extra cache capacity stolen from Elastic(X) jobs. We can see the impact of automatic mode downgrade on Strict jobs’ wall clock time in the All-Strict+AutoDown configuration. Both the wall-clock time and variation in wall-clock time increase significantly compared to the Strict jobs in the All-Strict configuration. However, as long as the wall-clock time variation can be tolerated by the job, this is a good trade-off because all the jobs in the All-Strict+AutoDown still meet their deadlines and overall job throughput is significantly improved compared to the All-Strict case. Finally, the EqualPart configuration suffers from a high average and variation of wall-clock time, which is caused by the lack of admission control and resource reservation, as well as time-sharing.

### 7.2 Impact of Automatic Mode Downgrade

To better understand how automatic mode downgrade improves throughput, we show the detailed execution of each job in the All-Strict and All-Strict+AutoDown for the single-benchmark workload of bzip2 in Figure 7. The x-axes shows time in the number of millions of cycles, while the y-axes shows the first ten accepted jobs. Boxes with
solid lines represent the time from a job starting its execution until the time it completes, while boxes with dashed lines represent the amount of time between the job’s completion and its deadline. In addition, automatically downgraded jobs are shown in a darker color and the arrows point to the time when they are to be switched back to the Strict execution mode.

Figure 7. Execution trace of ten accepted jobs in the All-Strict case (a) versus in the All-Strict+AutoDown case (b). The jobs are instances of bzip2.

From the figure, we can see that in the All-Strict case, only two jobs can be accepted and run simultaneously, leading to a low admission rate and throughput. It takes 3,883M cycles to complete all ten jobs. In the All-Strict+AutoDown case, it only takes 3,451M cycles to complete ten jobs for several reasons. The first reason is that because Strict jobs running in the Opportunistic mode do not reserve resources, more jobs can be accepted and started earlier. For example, the third job is executed earlier because the first job runs in the Opportunistic mode rather than the Strict mode. The fifth, sixth, and seventh jobs are also admitted sooner and executed earlier. While automatically downgraded jobs run significantly slower, they utilize fragmented resources which otherwise would not have been utilized. The second reason is that when automatically downgraded jobs complete execution, the LAC reclaims their resources, allowing other jobs to be accepted earlier. For example, the completion of the fifth job allows the eighth job to be accepted and executed earlier, while the completion of the sixth job allows the tenth job to be accepted and executed earlier. Another observation is that out of the five automatically downgraded jobs, four of them (the first, fifth, sixth, and seventh) need to be switched back to the Strict mode because they do not complete early enough, while only one (the ninth job) is able to complete before it needs to be switched. Furthermore, the first and seventh jobs likely would not have met their deadlines had resources and timeslots not been reserved. This emphasizes the importance of reserving resources and timeslots for automatically downgraded jobs in order to guarantee meeting their deadlines.

7.3 Resource Stealing Evaluation

The amount of slack available in Elastic(X) jobs determines the amount of cache capacity that can be stolen and reallocated to Opportunistic jobs. Figure 8 shows the impact of varying the amount of slack (i.e., X) for Elastic(X) jobs in the Hybrid-2 case for the single-benchmark workload of bzip2. Recall that resource stealing attempts to steal cache capacity from Elastic(X) jobs such that their L2 cache miss rates may be increased by up to X%. Figure 8(a) shows that our cache miss tracking technique using duplicate tags is effective: the increase in miss rate in Elastic(X) jobs closely tracks the slack shown with the dashed line. The figure also shows that the CPI of Elastic(X) jobs increases at a slower rate compared to the increase in the L2 cache miss rate, roughly between one third to one half of the rate. This demonstrates that bounding the increase in L2 miss rate is a safe proxy for bounding the increase in CPI (Section 4.2).

Figure 8. The impact of the amount of performance slack for Elastic(X) jobs in the Hybrid-2 configuration on the increase in miss rate and CPI (a) and on throughput of Opportunistic jobs (b). The jobs are instances of bzip2.

Figure 8(b) shows the average wall-clock time of Opportunistic jobs for various slack values in the Hybrid-2 configuration. While the figure expectedly shows that increasing the slack of Elastic(X) jobs leads to the decrease in wall-clock time of Opportunistic jobs, the benefit gradually shrinks with higher slack values. For example, when X=5%, Elastic(X) jobs may be slowed down by up to 5%, but Opportunistic jobs are sped up by almost 10%. However, when X=20%, Elastic(X) jobs may be slowed down by up to 20% but Opportunistic jobs are sped up by only 15%. This points to an observation that even a small slack amount
can recover much of the excess cache capacity, while a large slack amount has a diminishing role in recovering excess cache capacity.

### 7.4 Evaluating Mixed-Benchmark Workloads

So far we have only used single-benchmark workloads to evaluate our QoS framework. In this section, we evaluate two mixed-benchmark workloads described in Table 3. Recall that Mix-1 represents an ideal workload for resource stealing: the cache-sensitive benchmark (bzip2) forms Opportunistic jobs while the cache-insensitive benchmark (gobmk) forms Elastic(5%) jobs. Mix-2, on the other hand swaps the execution modes of bzip2 and gobmk, so it is not an ideal workload for resource stealing.

Figure 9(a) shows the deadline hit rates for different configurations. While our QoS framework achieves 100% deadline hit rates for all Strict and Elastic(X) jobs in mixed-benchmark workloads, EqualPart has low deadline hit rates (30% for Mix-1 and 40% for Mix-2). Figure 9(b) shows the job throughput for Mix-1 and Mix-2 normalized to the respective All-Strict cases. Overall, all of Hybrid-1, Hybrid-2, and All-Strict+AutoDown achieve a significant improvement in throughput compared to All-Strict. The throughput improvements achieved in Hybrid-1 and Hybrid-2 sometimes even exceed that of EqualPart. This is a significant result considering that while a majority of jobs miss their deadlines in EqualPart, they meet their deadlines in our QoS framework while simultaneously achieving higher throughputs.

![Figure 9. Comparing two mixed-benchmark workloads.](image)

Comparing the throughput improvement between the two workloads, Mix-1 achieves lower throughput improvement than Mix-2 in Hybrid-1 (35% vs. 42%) but higher than Mix-2 in Hybrid-2 (47% vs. 39%). This is because in Hybrid-1, the Opportunistic jobs can only utilize unallocated cache capacity, and because bzip2 is more cache sensitive than gobmk, instances of bzip2 have more restricted throughput compared to instances of gobmk. However, in Hybrid-2, Opportunistic jobs benefit from extra cache capacity stolen from Elastic(X) jobs. In this case, Opportunistic bzip2 jobs benefit more from resource stealing than Opportunistic gobmk jobs, while at the same time Elastic(X) gobmk jobs can give up more of their cache space and are slowed down less than Elastic(X) bzip2 jobs. The outcome of this combination is that resource stealing is more effective for Mix-1 than for Mix-2. The consequence of this observation is that resource stealing should be applied selectively if our goal is to maximize throughput.

### 7.5 Characterization of the Local Admission Controller

In our framework, the LAC is implemented as a user level program and is fully simulated in our evaluation. The LAC incurs performance overheads when it performs admission tests and scheduling. However, since the LAC only performs a simple admission control policy and implements a simple scheduling algorithm, the occupancy of the LAC is less than 1% of each workload’s wall-clock time. If the number of jobs submitted to the CMP increases, or if the number of cores in the CMP increases, the LAC’s overheads will increase proportionally although they likely remain low.

### 8 Conclusions

This paper has presented a novel QoS framework that provides performance QoS through the use of appropriate QoS target specification and execution modes. It has also presented how throughput can be improved while still preserving QoS, by employing several techniques such as manual and automatic mode downgrade and resource stealing. Through this study, we discover several findings. First, QoS targets should be specified with Resource Usage Metrics (RUM) in order to fully provide QoS and to build an admission control policy. Secondly, QoS-enabling features such as the ability to dynamically partition caches and a resource manager which tries to meet the IPC target of all jobs, are by themselves insufficient for fully providing QoS. Thirdly, substantial throughput is lost when we provide strict QoS with our framework due to external processor core and cache fragmentation, and internal cache fragmentation. Fourthly, the two alternative QoS execution modes (Elastic(X) and Opportunistic) enable the system to recover much of the lost throughput by reducing resource fragmentation. Manual mode downgrade in general is more effective than automatic mode downgrade if there is an appropriate mixture of Elastic(X) and Opportunistic jobs. However, even when there are only Strict jobs, the overall throughput of the system can still be boosted by using automatic mode downgrade. Finally, resource stealing is an effective microarchitecture technique for improving throughput by...
reallocating excess cache capacity from Elastic(X) jobs to Opportunistic jobs while still meeting the Elastic(X) jobs’ QoS targets.

References


