ReQoS: Reactive Static/Dynamic Compilation for QoS in Warehouse Scale Computers

Lingjia Tang   Jason Mars
University of California, San Diego
{lingjia, mars}@cs.ucsd.edu

Wei Wang   Tanima Dey   Mary Lou Soffa
University of Virginia
{wwang,td8h,soffa}@virginia.edu

Abstract

As multicore processors with expanding core counts continue to dominate the server market, the overall utilization of the class of datacenters known as *warehouse scale computers* (WSCs) depends heavily on colocation of multiple workloads on each server to take advantage of the computational power provided by modern processors. However, many of the applications running in WSCs, such as websearch, are user-facing and have quality of service (QoS) requirements. When multiple applications are co-located on a multicore machine, contention for shared memory resources threatens application QoS as severe cross-core performance interference may occur. WSC operators are left with two options: either disregard QoS to maximize WSC utilization, or disallow the co-location of high-priority user-facing applications with other applications, resulting in low machine utilization and millions of dollars wasted.

This paper presents **ReQoS**, a static/dynamic compilation approach that enables low-priority applications to adaptively manipulate their own contentiousness to ensure the QoS of high-priority co-runners. ReQoS is composed of a profile guided compilation technique that identifies and inserts markers in contentious code regions in low-priority applications, and a lightweight runtime that monitors the QoS of high-priority applications and reactively reduces the pressure low-priority applications generate to the memory subsystem when cross-core interference is detected. In this work, we show that ReQoS can accurately diagnose contention and significantly reduce performance interference to ensure application QoS. Applying ReQoS to SPEC2006 and SmashBench workloads on real multicore machines, we are able to improve machine utilization by more than 70% in many cases, and more than 50% on average, while enforcing a 90% QoS threshold. We are also able to improve the energy efficiency of modern multicore machines by 47% on average over a policy of disallowing co-locations.

Categories and Subject Descriptors B.3.3 [*Hardware*]: Memory Structures—Performance Analysis and Design Aids; D.3.4 [*Programming Languages*]: Processors—code generation, run-time environments, compilers, optimization; D.4.8 [*Operating Systems*]: Performance—measurements, monitors

General Terms Performance, Algorithms, Measurement

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASPLOS'13, March 16–20, 2013, Houston, Texas, USA.
Copyright © 2013 ACM 978-1-4503-1870-9/13/03...$15.00

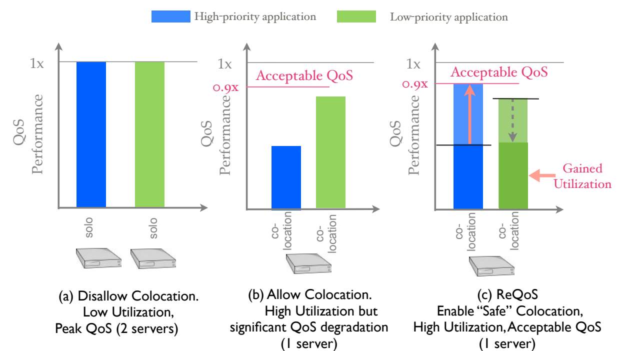

Figure 1. Goal of **ReQoS**

Keywords datacenter; warehouse scale computers; quality of service; contention; cross-core interference; compiler; dynamic techniques; online adaptation; runtime systems; multicore

1. Introduction

Web-service companies such as Google, Yahoo, Amazon, and Microsoft host large-scale data intensive applications that require nothing less than a *Warehouse Scale Computer* (WSC) [3] to run. These WSCs house hundreds to thousands of machines to provide the computing resources needed to serve millions of users. To limit the cost of ownership of WSCs, these machines are composed of commodity components that are cheap and easily replaceable, often 2 to 4 server grade processors per machine, and 4 to 12 cores per chip. The cores of a single chip share memory subsystem resources such as caches and bandwidth to the main memory on the machine. When multiple applications are running simultaneously on a multicore machine, resources sharing and contention among cores can result in a significant amount of performance interference. This interference leads to a significant problem for the service level requirements of user facing web-service applications [19, 33].

User facing *latency-sensitive* applications must provide a predictable and sometimes strictly defined *quality of service* (QoS). To avoid the constant unpredictable threat that shared resource contention poses to application QoS, datacenter operators and system designers typically disallow co-locations of latency-sensitive jobs with other jobs. This unnecessary *over-provisioning* of compute resources reduces the overall utilization of WSCs, recently reported to be below 30% on average [22], and results in an unnecessarily high cost and a large environmental footprint for a given set of web-service workloads.

Improving the utilization of multicore processors in light of QoS constraints has been identified as an important goal by prior

work. However, the problem is far from solved in real-world deployments. A number of novel hardware solutions have been proposed [12, 13, 27, 28, 30, 36] to address contention and performance fairness. However, these solutions are not readily deployable and high cost may hinder their adoption in production. Contention aware scheduling has also been proposed in prior work [2, 4, 7, 11, 15, 23, 34, 37]. However, these techniques assume that a balanced set of contentious and non-contentious jobs are available to maximize utilization and can only select co-run schedules of these jobs. When contentious workloads must co-run, one approach that has largely been unexplored in prior work is to dynamically *change* the contentious nature of applications.

The goal of this work is to manage contention directly by manipulating the contentiousness of co-running applications to ensure high quality of service and high utilization, *irrespective* to the co-run schedule. This goal is summarized in Figure 1. As opposed to disallowing the co-location of high-priority and low-priority applications to guarantee QoS, as shown in 1(a), or simply allowing co-locations and suffering performance interference, as shown in 1(b), we aim to enable "safe" co-locations between potentially contentious low-priority applications with high-priority applications, shown in 1(c), by providing a software mechanism to manipulate contentiousness directly.

Recently, compilation has been proposed to provide a mechanism to indeed modify the contention characteristics of applications to attenuate the interference caused to co-running applications [32] by applying transformations to stagger the memory request rate of contentious code regions. This approach effectively trades the performance of low-priority applications for the QoS of high-priority applications to facilitate co-location and has shown to be particularly useful within the ecosystem of applications in WSCs as the workloads and infrastructure in a WSC are typically owned by the same web-service company, such as Google, Microsoft, Yahoo and Apple. However, there are three critical limitations of this approach, all of which stem from the fact that the code transformations are decided before runtime and applied statically:

1. The performance sacrificed to ensure the "niceness" of the low-priority application is overly conservative as it persists regardless of whether it is in fact co-running with a high-priority application that is indeed sensitive to contention.
2. Specific QoS goals cannot be targeted and the self-adaptation of the low-priority application to ensure a QoS goal is not possible. With a static approach, the resulting QoS of the high priority application when co-located is variable and unpredictable.
3. A significant tuning effort is required for each low-priority application as it is based entirely on the particular contentious characteristics of the application.

To address these three challenges and best accomplish the goal outlined in Figure 1, a *reactive* dynamic approach is needed to effectively detect contention at runtime and adaptively adjust the contentious nature of an application based on the amount of contention that is actually occurring. Such an approach addresses each of the three limitations noted. In designing this reactive approach, we aim to ensure its deployability in WSCs by providing a lightweight software solution that does not incur high performance overheads and is unobtrusive in that it does not require changes to the current production system software stack in deployment. In this paper, we present **ReQoS**, a static/dynamic compilation approach for adaptively manipulating application contentiousness online to increase WSC utilization while enforcing application QoS.

ReQoS is composed of two co-designed components: the **RQ-Compiler** and **RQ-Runtime**. First, our RQ-Compiler uses a profiling approach similar to that proposed in prior work [32] to identify the code regions in low-priority applications that aggressively demand memory resources and may cause resource contention. Then, as opposed to statically pessimizing the contentious region, our RQ-Compiler applies a code marking technique on those regions. The marking technique has been co-designed with the RQ-Runtime to enable the flexible and reactive manipulation of the code regions' contentiousness. Dynamically, the RQ-Runtime uses a novel software-only technique to detect when contention has caused QoS degradations and adaptively throttles down the memory request rate by injecting short naps in the contentious code regions of the low-priority application. The degree of nap insertion applied to the contentious code regions of the low-priority application is dynamically determined based on the severity of observed QoS degradation of the high-priority application, resulting in more drastic responses to higher levels of contention. As contention lessens dynamically, naps are reduced, increasing the execution rate of the low-priority application to maximize machine utilization. Keep in mind that when the application executes code not identified as contentious, these naps do not occur. In performing this execution rate manipulation, cores that would otherwise remain idle are utilized.

To the best of our knowledge, this paper is the first to enable the direct modification of the contentiousness of low-priority applications dynamically to ensure the QoS of high priority applications. Specifically, this paper makes the following contributions:

- **The RQ-Compiler** - We present a compilation approach that enables the adaptive manipulation of contentiousness of the low-priority application. The RQ-Compiler identifies the contentious code regions of an application and inserts hooks in these regions that are used to invoke runtime manipulation.
- **The RQ-Runtime** - We present a runtime system that continuously monitors the QoS of high-priority applications, detects when contention is occurring dynamically, and directs the manipulation of the contentiousness of low-priority applications based on an adaptation policy.
- `Simple` **and** `Targeted` **Adaptation Policies** - We present two adaptation policies, `simple`, which provides a "knob" that controls whether the online response to contention emphasizes higher utilization or higher QoS, and for when a specific QoS threshold is available, `targeted`, that dynamically self tunes to the specified QoS target while maximizing utilization.
- We present a thorough evaluation of ReQoS and each of its adaptation policies including a phase level analysis demonstrating how ReQoS adapts during execution to ensure application QoS.

Using ReQoS on SPEC2006 and SmashBench workloads on a Quad Core Intel Nehalem machine, we are able to improve utilization by more than 70% in many cases, and more than 50% on average, while enforcing a 90% QoS threshold. We are also able to improve the energy efficiency of modern multicore machines by 47% on average over a policy of disallowing co-locations. Comparing to prior work [32], ReQoS improves utilization by 51% on average while providing similar QoS improvement.

The rest of the paper is organized as follows: Section 2 presents the overview of ReQoS. Section 3 presents our compilation technique to instrument markers to identified contentious code regions to invoke dynamic adaptation. Section 4 presents the runtime and policies for dynamic contention detection and reaction. Section 5 presents the evaluation. Section 6 presents prior work and Section 7 concludes.

2. ReQoS Overview

ReQoS provides a software mechanism that automatically and adaptively regulates the pressure that a low-priority *batch* appli-

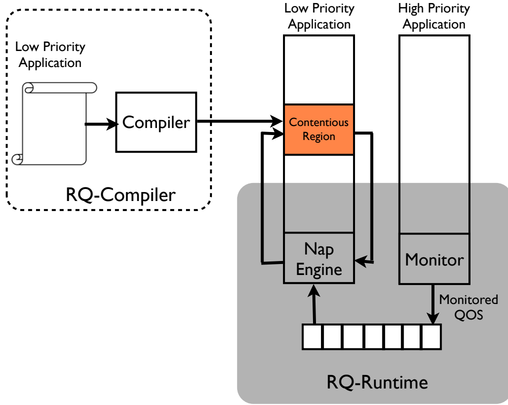

Figure 2. ReQoS Overview

cation applies to the shared memory subsystem resources to ensure the QoS of high-priority *latency-sensitive* applications. As shown in Figure 2, ReQoS consists of two components: The RQ-Compiler, and RQ-Runtime.

2.1 RQ-Compiler

The RQ-Compiler is a static profile-driven compiler approach that uses a performance counter based profiling analysis to identify contentious code regions and insert markers on those regions to steer the runtime adaptation. This profiling allows us to pinpoint the potentially problematic code regions for accurate dynamic contention detection with a small runtime overhead. As shown in Figure 2, the inserted markers trigger the RQ-Runtime, via the *Nap Engine*, when contentious code regions are executed. These triggers call upon the runtime to directly manipulate the rate of memory accesses generated by the low-priority application through the Nap Engine interface. The binaries produced by the RQ-Compiler can also be run without the RQ-Runtime. In this case, the inserted markers are benign, and the application runs as normal. The overhead of having these markers present in the binary is minimal. A full evaluation of the overhead is presented in Section 5. The RQ-Compiler and the profiling analysis used to identify contentious code regions are described in detail in Section 3.

2.2 RQ-Runtime

The RQ-Runtime is responsible for monitoring the QoS of high-priority applications, detecting when a low-priority application interferes with the performance of the high-priority application, and dynamically deciding the degree of memory access rate reduction to apply to alleviate the performance interference. As shown in Figure 2, a lightweight dynamic runtime that monitors application QoS is attached to the high-priority application. This runtime periodically reports an application's QoS through a shared memory buffer. The Nap Engine that is attached to the application binary of the low-priority application reads the most recent QoS reports from this buffer to steer the online contention response. The RQ-Runtime and the adaptive policies are described in detail in Section 4.

Figure 2 also illustrates how ReQoS is used in the context of a WSC. All low-priority applications in the WSC are compiled with a flag denoting that it is a low-priority application. The applications are then compatible for execution with ReQoS enabled. When these applications are scheduled to co-run with a high-priority application, QoS monitoring is turned on, and the Nap Engine enacts the adaptation policy.

2.3 ReQoS in the OS

ReQoS can be integrated into the OS by implementing the RQ-Runtime as a module that is compiled into the OS and allowing the application markers to trigger the runtime via system calls for ReQoS. However, while contention aware scheduling approaches can naturally be realized as OS techniques, the dynamic manipulation of the contentiousness of specific code regions can best be realized using user-mode runtime systems as the cost to invoke the runtime in user-mode is an order of magnitude more efficient than suffering an OS context switch for each runtime invocation. These context switches can become overbearing especially when considering the frequency of invocations during detection and the fine grain feedback control mechanisms discussed in Section 4.

3. Compiling for ReQoS

In this section we present RQ-Compile, our static compilation to enable dynamic contention mitigation and QoS improvement at runtime. The RQ-Compile process is illustrated in Figure 3. To compile a low-priority application, we first identify its contentious code regions using a profiler that scores the contentious nature of code regions as they execute. We then insert markers in those regions that periodically invoke the RQ-Runtime. Because markers target the problematic regions, the runtime engine is only triggered when the contentious regions are executing, minimizing the runtime overhead.

3.1 Profiling to Identify Contentious Code

Our profiling analysis provides dynamic scoring of sequences of executed code for its contentious nature using a prediction model based on prior work [32]. The model uses the memory resource usage of a code region to predict the severity of the performance interference the code may cause when co-running with other applications. The more aggressively a code region consumes the shared memory resources such as caches and memory bandwidth when executing, the more likely it is to be contentious for the resource when co-running. To collect the memory resource usage information of a code region, the profiler samples performance monitoring units (PMUs) during runtime. Using the prediction model based on the performance counter sampling, the profiler dynamically calculates the contention score of a code region, identifies contentious code regions that aggressively demand shared memory resources and thus most likely to cause contention and performance degradation when co-running with other applications.

The prediction model takes several important shared resources including shared caches, memory bandwidth and prefetchers into consideration. Our general linear model is as follows:

$$C = a_1 \times LLC_usage + b_1 \times BW_usage + c_1 \times Pref_usage, \quad (1)$$

where C is the contention score indicating the amount of the potential degradation a code region may cause to its co-runners. LLC is the shared last level cache; BW is memory bandwidth and $Pref$ is the prefetchers.

After identifying the appropriate performance counters, we apply multiple regression to determine the appropriate model coefficients for our experimental platform (Section 5.1) using **SmashBench** [19], a suite of contentious kernels developed within Google to span a spectrum of contentious memory access patterns and working set sizes. The established model is as follows:

$$\begin{aligned} C = {} & 1.663 \times (L2LinesIn/ns - L3LinesIn/ns) \\ & + 8.890 \times L3LinesIn/ns + 0.044, \end{aligned} \quad (2)$$

where $L2LinesIn$ is the number of cache lines brought in for the last level private cache (L2) and $L3LinesIn$ for the shared last level cache (L3). The difference between the two measures the

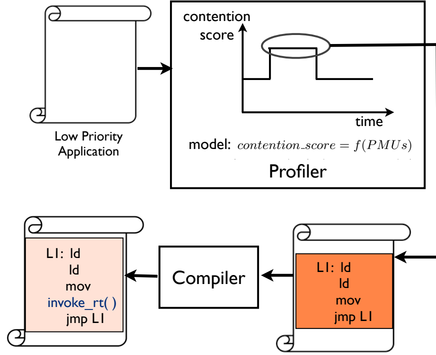

Figure 3. ReQoS Compilation

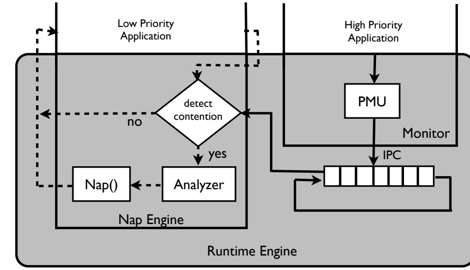

Figure 4. ReQoS Runtime Architecture

amount of data coming from the shared cache instead of the main memory, including the traffic generated by L2 prefetchers. It is used as a proxy to measure the shared cache usage. $L3LinesIn$ is used to measure the memory bandwidth consumption. The coefficients of each term in Equation 2 demonstrate the relative impact of bandwidth contention and LLC contention, implying that memory bandwidth contention may have a more dominating impact on performance degradation on this platform.

3.2 Compiling Contentious Code

Our profiler identifies the contentious code regions based on the PMU model and applies instrumentation to these regions. The instrumentation facilitates the dynamic manipulation and adaptation of a code region's contentiousness. To identify these contentious regions, during profiling, performance counters (L2 and L3 cache lines in rate) are sampled every 1 `ms` and the contention score is calculated using Equation 2. To correlate the contention score to the corresponding static code regions, the number of instructions retired in each 1 `ms` execution interval is also sampled and recorded. After the profiling run, a PIN [18] tool is used to replay the execution. Based on the recorded instruction profile, our PIN tool identifies the hottest basic blocks that are executed during each 1 `ms` execution interval and assigns the corresponding contention score to these basic blocks. The PIN tool then selects the basic blocks with a high contention score.

After these highly contentious basic blocks are identified, we instrument markers, *invoke_rt( )*, to the contentious code, shown in Figure 3. At runtime, these markers invoke the RQ-Runtime to dynamically decide the throttling policy. To minimize the potential overhead of frequent calls to the runtime, we have implemented a number of optimizations. Most notably, we use a self-tuning global checker that allows the call to the runtime to be executed only after sufficient execution iterations of the same basic block. This is especially helpful when a large number of markers are inserted in the critical path of execution. Instead of executing the function call every time, an increment and compare is executed in the average case.

4. The ReQoS Runtime

ReQoS combines both static compilation and dynamic adaptation. The profiling and static compilation enable the dynamic engine to manipulate the execution of the low-priority application. They also assist the diagnosis of contention and trigger the runtime only when problematic code regions are executing. As neither the particular co-locations of high-priority applications and low-priority applications nor how sensitive the high-priority application may be to resource contention are known statically, it is desirable to have a runtime approach that can adjust the amount of throttling dynamically. In this section, we present RQ-Runtime, our runtime engine that detects the QoS degradation of high-priority applications due to resource contention online, and adaptively manipulates the contentiousness of the low-priority applications to mitigate the degradation.

4.1 Runtime

Figure 4 illustrates the design for RQ-Runtime. The runtime engine is composed of two main components: *Monitor* and *Nap Engine*. The Nap Engine is linked into the low-priority application and the Monitor is either linked into, or attached to the PID of, the high-priority application. The Nap Engine and the Monitor communicate through a shared memory buffer.

[Monitor] The Monitor is responsible for monitoring the QoS of high-priority applications. The monitor is capable of using various performance metrics. In our implementation we use instruction-per-cycle (IPC) as a proxy for QoS. The IPC is often used in production datacenters as a QoS proxy because it is readily available using hardware performance counters and can be sampled with little overhead. For example, Google Wide Profiling (GWP) is currently deployed in Google's fleet to collect IPC and other counters for performance monitoring and debugging [26]. The Monitor uses a *periodic probing* technique, leveraging a timer interrupt to sample the hardware performance counters every 1 `ms`, and storing the recent sequence of IPC samples in a circular buffer in the shared memory. As we show in Section 5, this period probing technique incurs a minimal overhead (often less than 1%).

[Nap Engine] Based on the monitored QoS, the Nap Engine detects resource contention and QoS degradation, and accordingly reacts by deciding the appropriate execution rate reduction for the low-priority application. The Nap Engine is only invoked by the instrumented markers when the low-priority application is executing the contentious regions. Instead of invoking the Nap Engine every time an instrumented contentious basic block is executing, a timer based on the time stamp register, read using the RDTSC instruction [1], is used in the instrumentation to only yield control from the low-priority application to the Nap Engine periodically (2 `ms` in our experiments). To further reduce the overhead of timer checking, we also use this timer to adapt the global checker mentioned in Section 3 by approximating the amount of runtime invocations to skip before reading the timestamp counter again. This approximation requires a simple calculation based on the time past since the prior invocations and is adaptively adjusted upon every timestamp read. Due in part to these optimizations, the overhead of invoking the Nap Engine is low, never exceeding 5%. The evaluation is presented in Section 5.

When invoked, the Nap Engine's main tasks are to firstly detect contention and QoS degradation based on the information provided by the Monitor, and secondly if contention is detected, analyzes and decides how to appropriately throttle down the low-priority appli-

cation to mitigate the degradation. The Nap Engine controls the execution rate of a low-priority application by putting the execution of a contentious code region to epochal intermittent short "nap" mode.

Naps reduce the memory request rate and execution rate of the low-priority application and the pressure it puts on the shared memory subsystem. This in turn prioritizes the memory requests of the co-running high-priority application, and as a result, the QoS degradation it suffers due to the resource contention is greatly reduced or eliminated. The nap is implemented using nanosleep( ). Two main parameters that affect the behavior and the effectiveness of napping include the *frequency* and the *duration* of naps. The Nap Engine controls these parameters and decides whether and when a nap should occur (essentially how long the low-priority application should execute at a normal rate) and how long of a nap it should take to effectively improve the QoS of the co-running high-priority application. Because the Nap Engine can directly control these two parameters, it has a fairly accurate and predictable rate reduction control. Flexible policies and heuristics for contention detection and reaction can be implemented in RQ-Runtime, which are further discussed in the next section.

4.2 Detection and Reaction

In this section, we present two adaptation policies used in RQ-Runtime to detect resource contention and QoS degradation, and to reactively control the execution rate of the low-priority application to mitigate contention if necessary. It is challenging to design a software approach to detecting contention as it occurs. As we mentioned earlier, this is mostly due to the fact that contention in various hardware components such as shared caches and memory controllers is not exposed to the software. We design probabilistic empirical approaches to tackling the challenge of dynamic contention detection based on the online monitoring and feedback control. Once contention and QoS degradation are detected, the Nap Engine is also tasked to decide the appropriate rate reduction to apply to the low-priority application to reduce the QoS degradation.

In this work, we design two heuristics for the Nap Engine: `simple` and `targeted`. The `simple` heuristic directly relies on QoS monitoring information of the high-priority application and is designed to provide users with a flexible, tunable "knob" to manage the tradeoffs between QoS and utilization. For example, the heuristic can be configured to prioritize QoS and conservatively reduce utilization or prioritize utilization and risk the QoS. However, the `simple` heuristic does not strive for a strict QoS goal, such as improving the QoS of high-priority application to above 90% of its normal QoS when running alone. The `targeted` heuristic on the other hand, is designed to accommodate a pre-specified QoS target. `Targeted` makes the detection based on closely monitoring the impact of throttling of the low-priority application on the QoS and uses an analytical model to adjust the appropriate nap duration adaptively.

Algorithm 1: Nap_Engine (Heuristic 1: Simple)

Input : $threshold_low, threshold_high, nap_ratio_low, nap_ratio_mid, nap_ratio_high$

ipc = latest IPC sample from the shared IPC buffer;
if $(ipc < threshold_low)$ **then**
  $nap_duration \leftarrow nap_ratio_low \times exec_duration$;
else if $(ipc < threshold_high)$ **then**
  $nap_duration \leftarrow nap_ratio_mid \times exec_duration$;
else
  $nap_duration \leftarrow nap_ratio_high \times exec_duration$;
end
$nap(nap_duration)$;

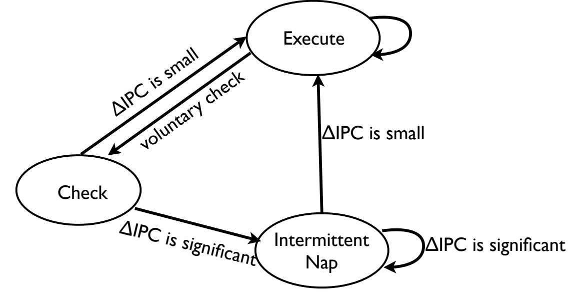

Figure 5. DFA for `Targeted` Heuristic

[Heuristic 1: Simple] The basic idea of `simple` is to detect and react purely based on the monitored QoS of the high-priority application. In our runtime implementation, we use instruction-per-cycle (IPC) as a proxy for QoS and `simple` adjusts the nap duration based on the dynamically monitored IPC. The details of our algorithm are described in Algorithm 1. When the contentious code region of a low-priority application (LP) is executing, the Nap Engine is invoked periodically (every `execution duration`). The Nap Engine then reads the latest IPC sample of the high-priority application (HP). Two thresholds (`threshold_low`, `threshold_high`) are used to bucket the monitored IPC into low, medium and high. The nap duration is decided based on which bucket the IPC is in. The lower the IPC, the longer the nap duration. In addition to IPC, application specific performance metrics such as query latency can also be monitored and bucketed. The rationale of this heuristic is that although many factors other than contention may cause QoS degradation (such as load fluctuations), we will conservatively throttle down the low-priority application once the QoS degradation of the high-priority application is observed. The parameter configurations (IPC thresholds and nap ratio) decide how QoS-biased (`conservative`) or utilization-biased (`optimistic`) the `simple` heuristic is. The sensitivity of those parameters is discussed in Section 5.

[Heuristic 2: Targeted] The primary design goal of `targeted` is to adaptively adjust the nap duration to improve the QoS to a user-specified goal (such as a minimum QoS of normalized 90% of a specific QoS target). The basic idea of `targeted` is to detect and react based on measuring how the QoS of high-priority application is affected by naps of low-priority applications. Figure 5 illustrates the logic of `targeted`. There are three basic states for a low-priority application (LP), which the Nap Engine tracks and controls. Periodically, the nap engine is invoked to analyze the QoS samples of the high-priority application (HP) and the analysis result triggers potential state transitions.

- `Intermittent nap state`. In this state, naps are inserted to throttle LP's execution rate. The QoS of the HP is sampled both when the LP is napping and when the LP wakes up from the nap and resumes executing. The difference between the two samples, *delta_IPC* is used to adjust the next nap duration. The bigger the difference, the more significant the impact of contention is, and the longer the nap duration should be. The detailed model of adapting the nap duration is shown later this section. When the IPC delta is smaller than the pre-specified QoS degradation threshold, it indicates that napping does not have a significant impact, and LP transition to the `execution state`.
- `Execution state`. In `execution state`, LP executes at the full rate with no naps inserted. However, LP does not stay in `execution state` indefinitely. A countdown is set to trigger

the transition to voluntary `check state` after a pre-specified execution period.

- `Check state`. The purpose of the `check state` is to periodically detect if contention occurs after a period of execution. The detection is similar to `intermittent nap state`. LP is put to nap for a short interval and then is run for a short interval. The difference of the IPC samples of HP during these two intervals is used to decide if contention is occurring. If so, LP transitions to the `intermittent nap state`; if not, the `execution state`.

Algorithm 2: Nap_Engine (Heuristic 2: Targeted)

Input : $QoS_goal, conservative_factor, execute_period$

if $(LP_state == execution_state \&\& execute_countdown > 0)$ **then**
  $execute_countdown - -$;
  return; `/* no napping, running ahead */` ;
else if $(LP_state == execution_state \&\& execute_countdown = 0)$ **then**
  $nap(check_interval)$;
  $ipc_nap \leftarrow$ read the average IPC of HP when LP is in the last napping duration from the shared IPC buffer ;
  $LP_state \leftarrow check_state$; `/* voluntarily nap for a short period to test if contention is back by checking delta_IPC */` ;
else if $(LP_state == nap_state \,||\, LP_state == check_state)$ **then**
  $ipc_exec \leftarrow$ read the average IPC of HP when LP is the execution interval from the shared IPC buffer ;
  $delta_ipc \leftarrow (ipc_nap - ipc_exec)/ipc_nap$;
  if $(delta_ipc < conservative_factor * (1 - QoS_goal))$ **then**
    $execute_countdown \leftarrow execute_period$;
    $LP_state \leftarrow execution_state$;
    return; `/* significant contention is not detected nap does not seem to have a big enough effect on IPC */` ;
  else
    $nap_duration \leftarrow calculate_duration(delta_ipc, QoS_goal)$;
    $nap(nap_duration)$;
    $ipc_nap \leftarrow$ read the average IPC of HP when LP is the last napping duration from the shared IPC buffer ;
    $LP_state \leftarrow nap_state$;
  end
end

The algorithm for `targeted` heuristics is described in Algorithm 2. A parameter `conservative factor` is used to guard how close the monitored QoS degradation is to the pre-specified threshold (*delta_IPC*, for example) before the napping is used to throttle down the LP. Also in Algorithm 2, we estimate the appropriate nap duration based on the QoS goal (the degradation threshold QoS_{thresh}, such as 90% of the optimal QoS), the given $exec_duration$ (how long LP executes between consecutive nap engine invocations) and the observed difference between IPC of HP when LP is napping (IPC_{nap}) and when LP is executing (IPC_{exec}). To estimate the appropriate nap duration we solve the following equation:

$$QoS_{thresh} = \frac{IPC_{nap} \times nap_duration + IPC_{exec} \times exec_duration}{IPC_{nap} \times (nap_duration + exec_duration)}, \quad (3)$$

where $exec_duration$ is the duration of the execution interval between inserted intermittent naps.

5. Evaluation

In this section, we first valuate the effectiveness of our two heuristics in mitigating the QoS degradation due to resource contention and in improving machine utilization. We also take a deeper look into the dynamic behavior of ReQoS and its reaction to contentious phases throughout execution. We then compare ReQoS against the technique presented in prior work. Lastly, we evaluate the overhead and power efficiency of ReQoS.

Configurations	thresh_low	thresh_high	nap_ratio
util_biased	0.5	1.0	{0, 1, 2}
balanced	0.8	1.5	{0, 1, 2}
QoS_biased	0.5	1.0	{1, 2, 3}

Table 1. Three configurations for `simple` heuristic

5.1 Setup and Methodology

Our evaluation is conducted on a 2.67GHZ Quad Core *Intel Nehalem* processor with private L1/L2 caches, an 8MB last level cache (L3) shared by four cores and 4GB main memory. This platform runs Linux 2.6.29.6 and a customized GCC 4.4.6.

The workloads used in our evaluation include the `sledge` application from the **SmashBench** contentious kernel suite [19, 20] (developed at Google) and applications from SPEC CPU2006. All benchmarks are compiled using GCC at the O2 level. All SPEC applications are run using **ref** inputs. Each experiment is conducted three times to calculate the average performance. Benchmark runs are stable with a performance variance of 1% or less between runs. As shown in prior work [19, 32, 33], key large-scale Google workloads including web search, bigtable and ad-servlets degrade significantly due to memory resource contention such as shared caches and bandwidth (5%-40%), similar to the amount of degradation SPEC CPU2006 (especially several SPEC memory intensive benchmarks) suffer on this architecture. In addition, prior work [32] shows that throttling down low-priority applications using inserted naps at millisecond granularity has a similar effect of improving the performance of both SPEC and Google applications.

5.2 Effectiveness of ReQoS: QoS and Utilization

In this section, we evaluate the effectiveness of ReQoS using both `simple` and `targeted` heuristics and discuss the tradeoffs between the two heuristics.

5.2.1 Heuristic 1: Simple

As mentioned in Section 4.2, our `simple` heuristic provides "knobs" that control whether the emphasis of ReQoS is biased towards QoS or machine utilization. Table 1 presents the three configurations we use in our evaluation. These include `util_bias`, `balanced`, and `QoS_bias`, representing an emphasis on higher utilization, a balance between utilization and QoS, and higher QoS respectively. `Threshold_low`, `threshold_high` and `nap_ration` are parameters for Algorithm 1 to control the binning of monitored instructions-per-cycle (IPC) of the high-priority application and the nap duration of the low-priority application. In general the longer the nap ratio, the more throttling down the heuristic applies to the low-priority applications, and the more biased the heuristic is towards the QoS of high-priority applications.

Figures 6, 7 and 8 present the QoS of the high-priority applications when we apply ReQoS to their co-running low-priority applications with three configurations of `simple` heuristic. In each of these figures, the x-axis shows the high-priority applications and the y-axis shows their QoS when each of them is co-running with a low-priority application, normalized to its QoS performance when running alone on the machine. For each high-priority application, a cluster of four bars demonstrates four settings for the co-running low-priority application. The first bar shows the QoS of the high-priority application when it is co-running with the original low-priority application without the ReQoS. The rest of the three bars show its QoS when we apply ReQoS with three con-

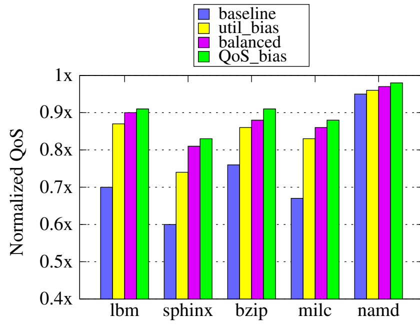

Figure 6. Normalized QoS of each benchmark co-running with `sledge` (`RQ_simple`)

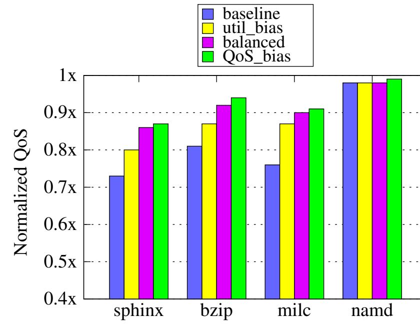

Figure 7. Normalized QoS of each benchmark co-running with `lbm` (`RQ_simple`).

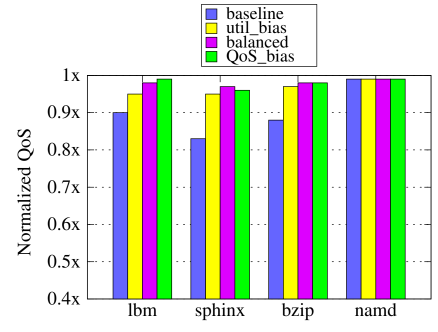

Figure 8. Normalized QoS of each benchmark co-running with `milc` (`RQ_simple`)

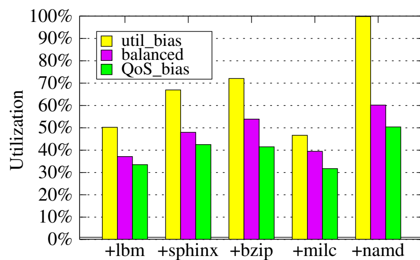

Figure 9. Utilization of `sledge` (`RQ_simple`)

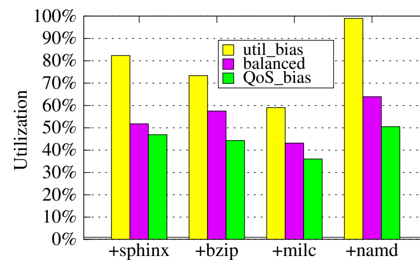

Figure 10. Utilization of `lbm` with each configuration. (`RQ_simple`)

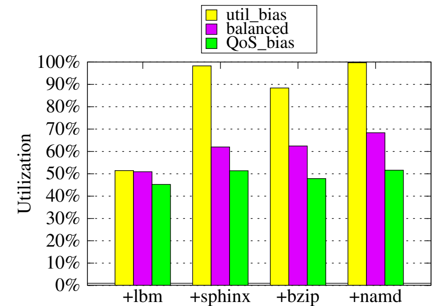

Figure 11. Utilization of `milc` with each configuration (`RQ_simple`)

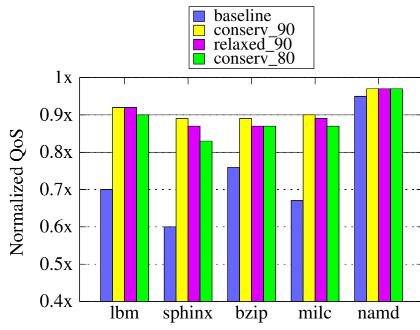

Figure 12. Normalized QoS of each benchmark co-running with `sledge` (`RQ_targeted`)

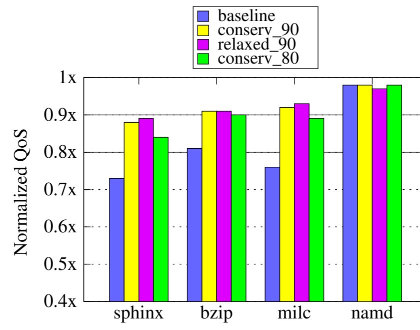

Figure 13. Normalized QoS of each benchmark co-running with `lbm` (`RQ_targeted`)

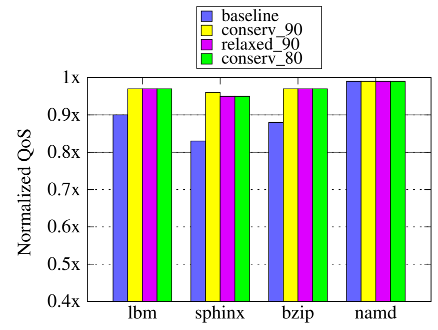

Figure 14. Normazlied QoS of each benchmark co-running with `milc` (`RQ_targeted`)

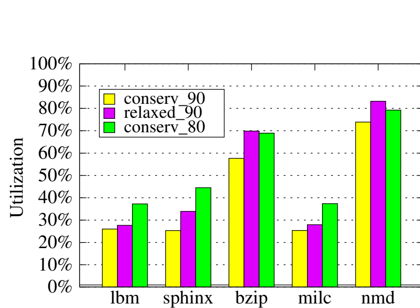

Figure 15. Utilization of `sledge` with each configuration (`RQ_targeted`)

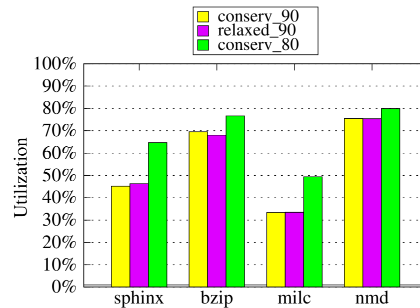

Figure 16. Utilization of `lbm` with each configuration (`RQ_targeted`)

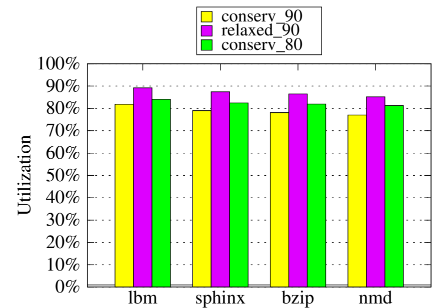

Figure 17. Utilization of `milc` with each configuration (`RQ_targeted`)

Configurations	ex_period(ms)	conserv_factor	QoS_goal
conservative_90	6	0.4	90%
relaxed_90	12	1.0	90%
conservative_80	9	0.4	80%

Table 2. Three configurations of `targeted` heuristic

figurations of the `simple` heuristic to its co-running low-priority application. Each of three low-priority applications, `sledge`, `lbm`, and `milc` is used in Figures 6, 7, 8 respectively. `Sledge` is a highly contentious kernel that contains two large arrays copying data back and forth between arrays with this sledgehammer pattern. `Lbm` and `milc` are top contentious benchmarks in the SPEC 2006 suite. The five high-priority applications in each graph present a wide range of sensitivity to contention in SPEC suite.

Figures 9, 10 and 11 show the corresponding utilization gained for each low-priority application. We measure the execution rate of the low-priority application normalized to its speed when running alone on that architecture to indicate the utilization of the computing resources used by the application. For example, 50% utilization for `lbm` indicates that `lbm` is now running at half of its speed when it is running alone.

From Figure 6-11, we observe that when applying ReQoS with our `simple` heuristic, the QoS of each high-priority application is significantly improved (by up to 26%) compared to simply allowing the co-location of both applications without ReQoS (first bars of each cluster in Figures 6, 7 and 8). Without ReQoS, such colocation of low- and high-priority applications would be disallowed due to the potentially significant QoS degradation. Compared to this commonly adopted approach of disallowing co-location, we gain a significant amount of utilization when allowing co-location with ReQoS, often more than 50%. Various configurations in `simple` heuristic also provide a wide range of options for balancing QoS and utilization. In this experiment, the utilization-biased configuration achieves significantly higher utilization than other two configurations, and the QoS of each high-priority application only slightly degrades. This demonstrates that the `simple` heuristic can be effective in improving QoS while gaining a significant amount of machine utilization.

5.2.2 Heuristic 2: Targeted

Our more sophisticated `targeted` heuristic enables a more precise enforcement to achieve the desired QoS requirements. This heuristic has effectively three "knobs," one for the specific QoS threshold to enforce, and the other two for how conservatively (strictly) this QoS threshold must be enforced (parameters `QoS goal`, `conservative factor` and `execution period` in Algorithm 2). With more conservative parameters, a larger amount utilization may be sacrificed; however, the application QoS is less likely to drop below the specified threshold. We explore this tradeoff in our evaluation.

Figures 12 – 17 are similar to those presented above. For this set of graphs we use our `targeted` heuristic with the three configurations presented in Table 2. The configurations `conserv_90`, `relaxed_90`, and `conserv_80` represent a conservative setting at a 90% QoS threshold, a relaxed setting at 90%, and a conservate setting at an 80% QoS threshold, respectively. Figures 12, 13 and 14 show the effect of using our `targeted` heuristic on the QoS of the high-priority applications. Note that two horizontal lines are drawn in each graph denoting the 90% and 80% QoS thresholds. Figures 15, 16 and 17 show the corresponding processor utilization gained for each configuration.

As shown in these figures, the `targeted` heuristic is quite effective in bringing the QoS of the high-priority applications to the desired QoS threshold, beating it in many cases and coming very close in the worst cases with our conservative settings. When using a relaxed setting, we observe a bump in the utilization, and our QoS target is often met. The decision as to how conservative or relaxed the QoS target is depends on the objectives and discretion of the application service provider and whether higher utilization is desired or stricter QoS polices are specified.

[Simple vs. Targeted] Our `simple` and `targeted` heuristics offer two options to application service providers: one providing the flexibility and the tradeoffs between utilization and QoS, the other allowing specifying a QoS target. When configured appropriately, the `simple` heuristic can perform quite well. However, it may require a significant amount of parameter tuning to search for the appropriate configuration if certain QoS level is required. The `targeted` does not require such parameter tweaking because it is self-tuning and feedback directed. More comparison between `simple` and `targeted` is presented in the following section.

5.3 Phase Level Behavior

We further evaluate the phase-level effectiveness of RQ-Runtime including our two heuristics in improving the QoS of high-priority applications and machine utilization.

[Improving QoS] Figure 18 presents the IPC of `sphinx` when it is running with the original `sledge`, comparing to its IPC when running with `sledge` on RQ-Runtime using the `simple` heuristic. The IPC samples are normalized to `sphinx`'s IPC profile when running alone to demonstrate the IPC degradation due to contention. In this experiment, `simple` heuristic is using `balance` configuration and `sphinx` is using `ref` input. To calculate the normalized IPC, we collect the IPC profiles of `sphinx` when it is running alone (solo) and running with `sledge`. IPC is sampled every 1 `ms` and all profiles of the entire execution of `sphinx` are down sampled to 1000 data points. The normalized IPC at point i is calculated as $\frac{IPC_{co-run_i}}{IPC_{solo_i}}$. Therefore, the closer the normalized IPC to 1, the less the degradation. In Figure 18, the line denoting the original `sledge` shows phase-level changes of the IPC degradation due to contention. For example, around samples 100 to 200, and 300 to 400, there are noticeable phases of degradation increase. The degradation is also less significant during the later half of the execution. Figure 18 clearly demonstrates the IPC improvement achieved by RQ-Runtime throughout the entire execution of `sphinx`. Instead of around 60%-70% of the normalized IPC when running with the original `sledge`, ReQoS improves the normalized IPC to above 80% during most of the execution.

Similar to Figure 18, Figure 19 presents `sphinx`'s normalized IPC when it is running with `sledge` using `targeted` heuristics (`conservative_90` configuration), also compared with its normalized IPC when running with the original `sledge`. Despite the distinctive phases of varying levels of degradation when running with the original `sledge` as discussed previously (for example, samples 100-200 and 300-400), `targeted` heuristic consistently achieves around 90% IPC for `sphinx` during the entire execution. This is different from the `simple` heuristic shown in Figure 18 where the improved normalized IPC fluctuates between 70% and 90%. This comparison highlights the difference between `simple` and `targeted` heuristics. While `simple` is effective in improving the QoS, `targeted` heuristic is effective in adapting, achieving and maintaining a stable QoS level as specified.

[Improving Utilization] Similar to Figures 18 and 19, Figures 20 and 21 present the normalized IPC of `sphinx` when it is running with the original `milc`, as well as `milc` with ReQoS. In Figures 20 and 21, the RQ-Runtime for `milc` uses the `simple` heuristic and `targeted` heuristic respectively. The normalized IPC of `sphinx` when running with the original `milc` demonstrates the phases with varying levels of contention and degradation. For example, during samples 600 to 800, the degradation is significantly

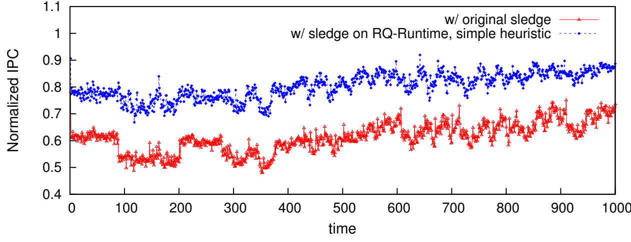

Figure 18. `Sphinx` normalized IPC with original `sledge` and with `simple sledge`

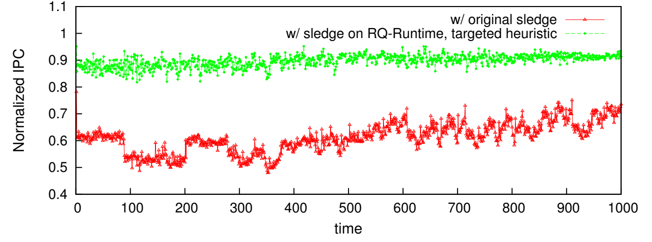

Figure 19. `Sphinx` normalized IPC with original `sledge` and with `targeted sledge`. Compared to `simple`, `targeted` heuristic is effective in achieving and maintaining a more stable QoS level as specified

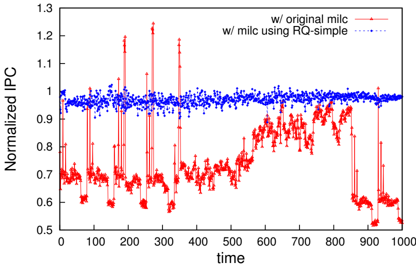

Figure 20. `Sphinx` normalized IPC with original `milc` and with `simple milc`

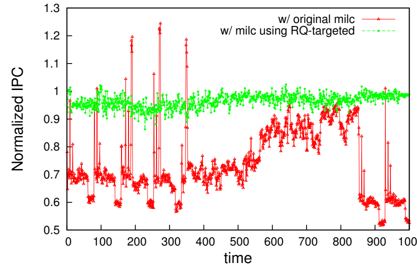

Figure 21. `Sphinx` normalized IPC with original `milc` and with `targeted milc`

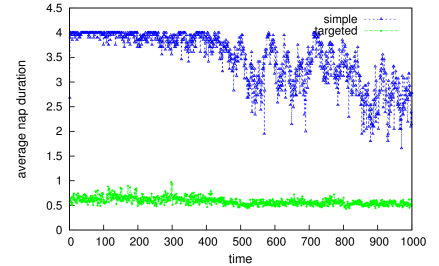

Figure 22. Average nap duration for `milc` with `simple` vs. `milc` with `targeted`. `Targeted` adaptively determines the necessary amount of napping, attaining higher utilization.

smaller (normalized IPC close to 1) than the rest of the execution. A few samples of normalized IPC are higher than 1 due to aliasing of downsampling. In this set of experiments, `targeted` heuristic is configured as `conservative_90`, meaning RQ-Runtime aims at less than 10% of the QoS degradation for `sphinx`. `Simple` heuristic is configured with `QoS_biased` configuration to achieve the similar QoS goal as `targeted`. Figures 18 and 19 demonstrate similar effectiveness of both heuristics. The QoS of `sphinx` is significantly improved after applying either heuristic to the co-running `milc`; the normalized IPC of `sphinx` is stable and between 0.9 and 1.

However, there is significant difference in nap duration with two heuristics. Figure 22 presents the corresponding average nap duration of `milc` for every 2 `ms`' execution, decided dynamically by RQ-Runtime based on dynamic contention detection. The longer the nap duration, the lower the utilization. Figure 22 shows that `simple` heuristic demonstrates certain adaptability. After sample 600 when the contention is not as significant, naps become shorter. However, when using the `targeted` heuristic, the nap duration is significantly shorter, and thus the utilization is much higher, while achieving similar QoS improvement as the `simple` heuristic. This demonstrates that `simple` heuristic may over-conservatively throttle down the low-priority application, while `targeted` heuristic can intelligently estimate and adaptively adjust the necessary amount of nap/throttling, attaining much higher utilization while achieving the QoS goal. This also demonstrates the importance of using dynamic feedback control (`targeted`) to prune the potential false positive contention detection instead of detecting purely on QoS degradation (`simple`).

5.4 Comparing to the prior art

Figures 23 and 24 compare the effectiveness of ReQoS against the state-of-the-art technique QoS-Compile [32]. Figure 23 presents the normalized QoS of high-priority applications when each of them is co-running with a low-priority application. The x-axis presents the workloads, each composed of a high-priority (HP) and a low-priority (LP) application. For example, `sphinx - lbm` denotes a HP application `sphinx` co-running with a LP application `lbm`. The first bar in each cluster of bars presents the QoS of the high-priority application running with the original LP application. The second and the third present its QoS when QoS-compile or ReQoS is applied to the LP application, respectively. C1 and C2 denote two configurations we use for QoS-Compile and ReQoS. For ReQoS, configuration 1 (C1) targets 80% QoS, and configuration 2 (C2) targets 90% QoS. Note that we can specify a QoS target in ReQoS and the RQ-Runtime can adaptively adjust the throttling to achieve the target. However, QoS-Compile does not have that functionality. To achieve a specified QoS, many trial runs of parameter tuning are needed. Here we carefully tune the configurations of QoS-Compile to achieve similar QoS improvement as ReQoS.

As Figures 23 and 24 demonstrate, while with careful tuning, QoS-Compile can be as effective as ReQoS in reducing contention and improving QoS, ReQoS attains significantly higher machine utilization than QoS-Compile. For example, for `sphinx-milc_C2`, ReQoS achieves 80% execution rate for `milc` as opposed to 40% using QoS-Compile while providing better QoS. On average, ReQoS attains 65% utilization compared to 43% by QoS-Compile, 51% better while providing similar QoS. This is mainly due to the fact that ReQoS can adaptively adjust the necessary throttling down as contention phase changes, avoiding unnecessary throttling down while maintaining stable QoS. However, QoS-compile determines the amount of throttling down statically. Even with tuning, it does not handle phase changes and may perform either over conservatively and waste utilization, or over-optimistically and fail to effectively mitigate degradation to achieve the QoS goal. In addition,

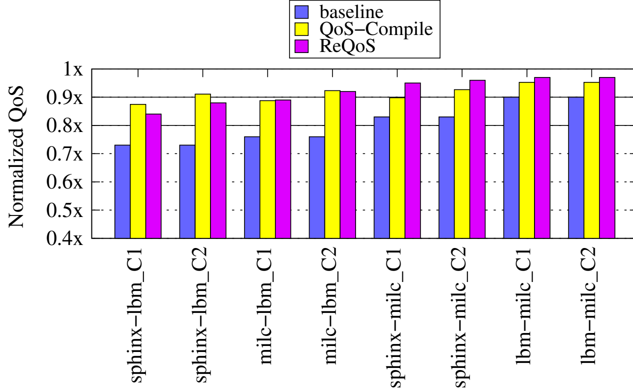

Figure 23. QoS of high-priority applications. When carefully tuned for each workload, QoS-Compile achieves similar QoS as ReQoS.

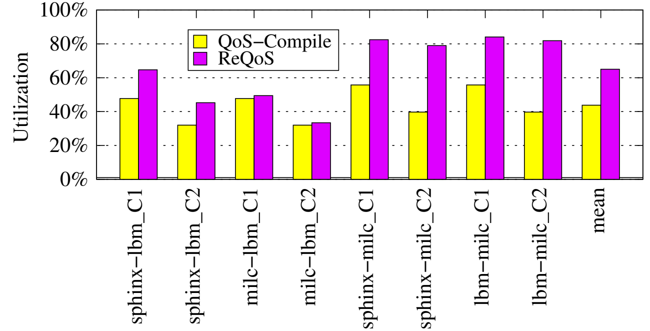

Figure 24. Utilization. ReQoS attains higher machine utilization than QoS-Compile.

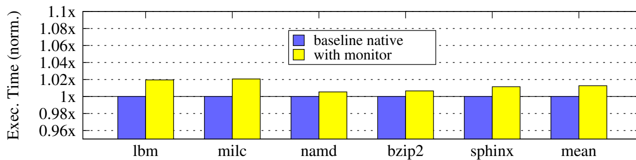

Figure 25. Overhead of monitoring for high-priority application.

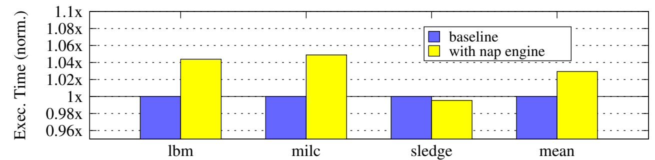

Figure 26. Overhead of nap engine for low-priority application.

due to its static nature, QoS-Compile also cannot adapt to new situations such as changes of the co-running high-priority applications or their QoS requirements. In comparison, ReQoS can adapt to these changes without any tuning.

5.5 Overhead

Figure 25 presents the performance costs of the monitoring the QoS of the high-priority application. The overhead is minimal. The overhead suffered by high-priority applications is less than 1% on average with a max of 2% in the cases of `milc` and `lbm`.

Figure 26 shows the performance overhead of invoking the Nap Engine to throttle down low-priority applications. The overhead of probing the Nap Engine is slightly more costly, approaching 5% for `milc`. However, the Nap Engine only causes overhead to the low-priority application, and thus the performance cost is not as important.

The low cost of our runtime approach for the high-priority application is due to the fact that the overhead of reading and recording performance counters at 1 `ms` granularity is minimal. The cost is slightly higher for the low-priority application because we add a lightweight check at the point of every compiler-inserted marker. However the runtime is only invoked when the marker is detected (when the identified contentious region is executing) instead of every 1 `ms`. Coarsening the granularity can further reduce these overheads; but the tradeoff must be made between a lower overhead and a higher penalty for potential delays in detecting contention as it occurs.

5.6 Energy Efficiency of ReQoS

Figure 27 presents the improved energy efficiency when allowing co-location with ReQoS. These experiments were performed using a P3 International Kill A Watt® power meter connected to our Quad Core Intel Nehalem machine to measure whole system watt consumption during execution. For each cluster of bars in the figure, the energy efficiency is calculated by the instructions processed per watt for a three minute time period after the machine wattage stabilizes during each run. The higher the bar, the more energy efficient. The x-axis shows the workloads, the high-priority and low-priority application pairs. The first bar for each workload shows the energy efficiency when using separate machines for low and high-priority applications; the second bar shows the energy efficiency of co-locating both high and low-priority applications using ReQoS with the `targeted` policy and the `conserv_90` configuration shown in Table 2. We observe a significant energy efficiency improvement for many workloads. Application pairs that include less contentious applications, such as `namd`, produce a greater benefit as there is less napping occurring. Meanwhile, highly contentious pairs, such as `sphinx-lbm`, show a more modest benefit. On average there is a 47% improvement of using ReQoS to allow co-location over using two separate machines for low- and high-priority applications.

5.7 Varying Architecture

To investigate the effectiveness of ReQoS across architectures, we performed experiments on a 2.6GHZs Quad Core AMD Phenom X4 system with 6MB last level cache and 3GB of main memory. This machine is also running Linux 2.6.29.6 and our customized GCC 4.4.6.

Figures 28 and 29 show the results for our `targeted` heuristic using the same configurations shown in Table 2. As shown in these figures, ReQoS is also quite effective on this platform. For both `lbm` and `milc` we achieve 80% to 90% utilization while significantly reducing the performance interference to our high-priority applications. The contentiousness of `sledge` is severe on this processor. For the `lbm-sledge` pair, we observe that when lowering the QoS threshold to 80% from 90%, we achieve more than 2x improvement for the utilization. Overall, as shown in Figure 28, our conservative settings meet and exceed our QoS requirements in all experiments,

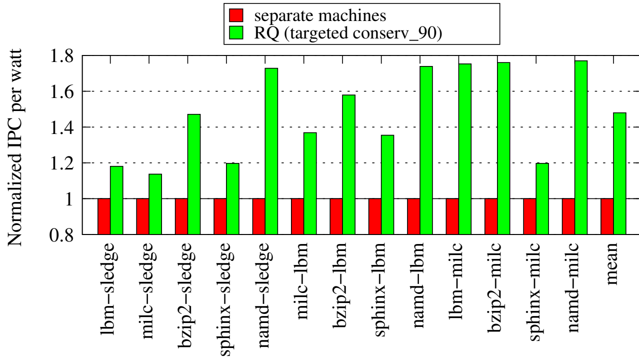

Figure 27. Energy efficiency of allowing co-location with ReQoS (`targeted`) vs. over-provisioning.

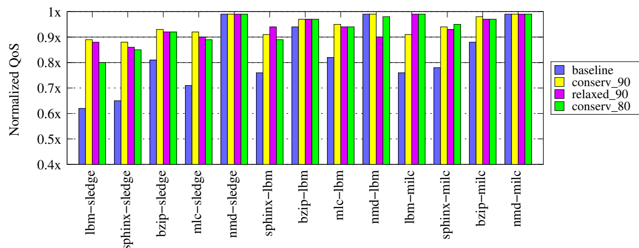

Figure 28. Effectiveness of `RQ_targeted` - QoS of each benchmark co-running with `sledge`, `lbm`, and `milc` on AMD X4

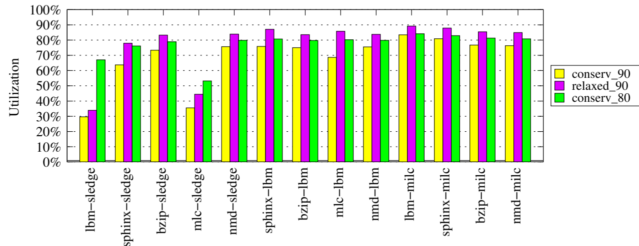

Figure 29. Effectiveness of `RQ_targeted` - Utilization of `sledge`, `lbm` and `milc` with each configuration on AMD X4

and our relaxed configuration satisfies the QoS constraint for the majority of the applications.

6. Related Work

Contention Aware Scheduling An important software approach to mitigating contention is contention aware scheduling [2, 4, 7, 11, 15, 23, 34, 37]. Our work is orthogonal and complementary to existing contention-aware scheduling approaches. The goal of scheduling is to, for a given set of applications, decide what applications should be co-running together to improve performance or performance isolation. The goal of ReQoS is to, for a given co-run schedule (a set of applications that are scheduled to run simultaneously), make sure that the high priority application achieves a pre-specified QoS goal by manipulating the contentiousness of low-priority applications. ReQoS can be used after the schedule is decided. To illustrate the difference, imagine a simple scenario where only two applications are available for scheduling and these applications do not "play nicely" together. In this situation, scheduling can either run these two appliations simultaneously (thus violating the QoS of the high-priority application), or only allow one to run to ensure QoS. ReQoS is complimentary in that it enables "safe colocation" by allowing two applications to run simultaneously and guaranteeing the QoS of the high-priority application. It is also important to note that, unlike ReQoS, prior work on contention-aware OS scheduling does not precisely guarantee application QoS. When the scheduler selects applications to co-run, the resulting QoS depends on the composition of the workloads. Different from scheduling, our approach does not require a balanced mix of high-contention and low-contention applications since we directly manipulate the contentiousness of an application to improve QoS. Even if only contentious co-runners are left to be scheduled our approach remains effective. Yang et al. present Redline [35], an OS technique to guarantee the QoS of interactive applications on time-sharing systems. However, we focus on contention among applications simultaneously executing across multiple cores. Software solutions to reduce cache contention using page coloring/remapping have also been proposed [5, 16, 29]. Most page coloring methods require significant modifications to the kernel and the knowledge of the cache design details.

QoS-compile Tang et al. [32] propose QoS-Compile, a state-of-the-art technique to statically throttle down the memory contentious regions to achieve the same goal as this paper. As shown in our evaluation (Section 5.4), because of its dynamic and reactive nature, ReQoS significantly outperforms QoS-Compile. Mars et al. [21] propose a shutter approach to detect contention, while our approach first identifies contentious code regions and then applies compilation techniques to those regions.

Researchers recently have started to explore using code transformations and restructuring to improve cache sharing and reduce contention on multicores [12, 13, 27, 28, 30, 36]. Most such research focuses on compilation techniques to improve cache sharing for a multi-threaded application. Differently, our approach manipulates how applications interact with each other in terms of contending for the memory resources. Hardware techniques such as cache/bandwidth partitioning and source throttling to improve performance and fairness on multicores have received much research attention [6, 8–10, 14, 17, 24, 25, 31]. These studies have shown promising future directions for hardware designers; however they require hardware changes and are not yet available in commodity chips.

7. Conclusion

In this work, we have shown that static compilation and dynamic adaptation can be combined to address the challenge of cross-core interference on the QoS of high-priority applications. We have presented ReQoS, a statically enabled dynamic compilation approach to improve machine utilization in WSCs by enabling the adaptive manipulation of the contentiousness of low-priority applications to ensure the QoS of high-priority co-runners. Using a profile guided compilation technique that identifies and inserts markers in contentious code regions, and a lightweight runtime that monitors the QoS of high-priority applications and reactively triggers short naps of low-priority applications when cross-core interference is detected we were able to improve utilization by more than 70% in many cases, and more than 50% on average, while enforcing a 90% QoS threshold. In this work, we argue that in addition to providing a compilation strategy for an application's individual performance, it is also desirable to include an application's "niceness" to other co-runners as a compilation objective, and show that ReQoS can significantly reduce performance interference to ensure application QoS.

8. Acknowledgements

We would like to thank the reviewers for their helpful feedback and suggestions. Lingjia Tang and Jason Mars were supported by Google research awards. This work was also partially supported by NSF CNS-0964627 and CNF-0811689 to the University of Virginia.

References

[1] Intel 64 and ia-32 architectures software developer's manual volume 2b: Instruction set reference, m-z.

[2] M. Banikazemi, D. Poff, and B. Abali. Pam: a novel performance/power aware meta-scheduler for multi-core systems. *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, Nov 2008.

[3] L. Barroso and U. Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis Lectures on Computer Architecture*, 4(1):1–108, 2009.

[4] M. Bhadauria and S. McKee. An approach to resource-aware co-scheduling for cmps. *ICS 2010*, Jun 2010.

[5] S. Cho and L. Jin. Managing distributed, shared l2 caches through os-level page allocation. *MICRO 39*, Dec 2006.

[6] E. Ebrahimi, C. Lee, O. Mutlu, and Y. Patt. Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. *ASPLOS 2010*, Mar 2010.

[7] A. Fedorova, M. Seltzer, and M. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. *PACT 2007*, Sep 2007.

[8] F. Guo, Y. Solihin, L. Zhao, and R. Iyer. A framework for providing quality of service in chip multi-processors. *MIRCO 2007*, pages 343–355, 2007.

[9] A. Herdrich, R. Illikkal, R. Iyer, D. Newell, V. Chadha, and J. Moses. Rate-based qos techniques for cache/memory in cmp platforms. *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, Jun 2009.

[10] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. Qos policies and architecture for cache/memory in cmp platforms. *SIGMETRICS '07: Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, Jun 2007.

[11] Y. Jiang, K. Tian, and X. Shen. Combining locality analysis with online proactive job co-scheduling in chip multiprocessors. *High Performance Embedded Architectures and Compilers*, pages 201–215, 2010.

[12] M. Kandemir, S. Muralidhara, S. Narayanan, Y. Zhang, and O. Ozturk. Optimizing shared cache behavior of chip multiprocessors. *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on DOI - UR -*, pages 505–516, 2009.

[13] M. Kandemir, T. Yemliha, S. Muralidhara, S. Srikantaiah, M. Irwin, and Y. Zhnag. Cache topology aware computation mapping for multicores. *PLDI '10*, Jun 2010.

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

[15] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using os observations to improve performance in multicore systems. *IEEE Micro*, 28(3):54–66, 2008.

[16] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. *HPCA 2008*, pages 367–378, 2008.

[17] F. Liu, X. Jiang, and Y. Solihin. Understanding how off-chip memory bandwidth partitioning in chip multiprocessors affects system performance. *HPCA 2010*, pages 1–12, 2010.

[18] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[19] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *MICRO '11: Proceedings of The 44th Annual IEEE/ACM International Symposium on Microarchitecture*, New York, NY, USA, 2011. ACM.

[20] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Increasing utilization in warehouse scale computers using bubble-up! *Special Issue: IEEE Micro's Top Picks from 2011 Computer Architecture Conferences*, 2012.

[21] J. Mars, N. Vachharajani, R. Hundt, and M. Soffa. Contention aware execution: online contention detection and response. *CGO '10: Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, Apr 2010.

[22] D. Meisner, B. Gold, and T. Wenisch. Powernap: eliminating server idle power. *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, Feb 2009.

[23] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-clouds: managing performance interference effects for qos-aware clouds. *EuroSys '10*, Apr 2010.

[24] K. Nesbit, N. Aggarwal, J. Laudon, and J. Smith. Fair queuing memory systems. *MICRO 2006*, pages 208 – 222, 2006.

[25] M. Qureshi and Y. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2006.

[26] G. Ren, E. Tune, T. Moseley, Y. Shi, S. Rus, and R. Hundt. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE Micro*, 30:65–79, 2010.

[27] S. Rus, R. Ashok, and D. Li. Automated locality optimization based on the reuse distance of string operations. CGO '11, pages 181 –190, Apr 2011.

[28] A. Sandberg, D. Eklöv, and E. Hagersten. Reducing cache pollution through detection and elimination of non-temporal memory accesses. *SC 2010*, Nov 2010.

[29] L. Soares, D. Tam, and M. Stumm. Reducing the harmful effects of last-level cache polluters with an os-level, software-only pollute buffer. *Micro 2008*, pages 258 – 269, 2008.

[30] S. Son, M. Kandemir, M. Karakoy, and D. Chakrabarti. A compiler-directed data prefetching scheme for chip multiprocessors. *PPoPP 2009*, Feb 2009.

[31] S. Srikantaiah, M. Kandemir, and M. Irwin. Adaptive set pinning: managing shared caches in chip multiprocessors. *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, Mar 2008.

[32] L. Tang, J. Mars, and M. L. Soffa. Compiling for niceness: mitigating contention for qos in warehouse scale computers. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 1–12, New York, NY, USA, 2012. ACM.

[33] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. The impact of memory subsystem resource sharing on datacenter applications. In *ISCA*, pages 283–294, 2011.

[34] D. Xu, C. Wu, and P.-C. Yew. On mitigating memory bandwidth contention through bandwidth-aware scheduling. *PACT 2010*, Sep 2010.

[35] T. Yang, T. Liu, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. Redline: first class support for interactivity in commodity operating systems. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 73–86, Berkeley, CA, USA, 2008. USENIX Association.

[36] E. Zhang, Y. Jiang, and X. Shen. Does cache sharing on modern cmp matter to the performance of contemporary multithreaded programs? *PPoPP 2010*, pages 203–212, 2010.

[37] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. *ASPLOS 2010*, Mar 2010.