Compiling For Niceness: Mitigating Contention for QoS in Warehouse Scale Computers

Lingjia Tang
University of Virginia
lt8f@cs.virginia.edu

Jason Mars
University of Virginia
jom5x@cs.virginia.edu

Mary Lou Soffa
University of Virginia
soffa@cs.virginia.edu

ABSTRACT

As the class of datacenters recently coined as *warehouse scale computers* (WSCs) continues to leverage commodity multicore processors with increasing core counts, there is a growing need to consolidate various workloads on these machines to fully utilize their computation power. However, it is well known that when multiple applications are co-located on a multicore machine, contention for shared memory resources can cause severe cross-core performance interference. To ensure that the *quality of service* (QoS) of user-facing applications does not suffer from performance interference, WSC operators resort to disallowing co-location of latency-sensitive applications with other applications. This policy translates to low machine utilization and millions of dollars wasted in WSCs.

This paper presents QoS-Compile, the first compilation approach that statically manipulates application contentiousness to enable the co-location of applications with varying QoS requirements, and as a result, can greatly improve machine utilization. Our technique first pinpoints an application's code regions that tend to cause contention and performance interference. QoS-Compile then transforms those regions to reduce their contentious nature. In essence, to co-locate applications of different QoS priorities, our compilation technique uses *pessimizing* transformations to throttle down the memory access rate of the contentious regions in low priority applications to reduce their interference to high priority applications. Our evaluation using synthetic benchmarks, SPEC benchmarks and large-scale Google applications show that QoS-Compile can greatly reduce contention, improve QoS of applications, and improve machine utilization. Our experiments show that our technique improves applications' QoS performance by 21% and machine utilization by 36% on average.

1. INTRODUCTION

As more of today's computing moves into the cloud, the emerging class of datacenters recently coined as modern warehouse scale computers [3] (WSCs) continues to embrace commodity multicore processors as the dominating platform [2, 3]. In currently available multicore designs, much of the memory sub-system is shared. These shared components include on-chip caches, data prefetchers, the memory bus, memory controllers, and underlying interconnect. When multiple applications are co-running on a multicore platform, contention for these shared resources often cause a significant amount of performance interference [7, 23, 40]. This interference proves particularly problematic to large-scale web service applications as it may prevent these applications from providing satisfactory quality of service (QoS). Throughout this work, QoS is defined using each application's performance metric as specified in its *service level agreement* (SLA) (e.g., a job's QoS level of 95% corresponds to 95% of the performance when an entire machine is dedicated to that job).

Mitigating the impact of contention on an application's QoS and enforcing the relative QoS priorities of co-running applications, while maximizing machine utilization, remains a key challenge in modern warehouse scale computers. On one hand, in order to reduce machine and operational cost, it is essential for datecenters to consolidate various workloads on multicore servers to improve machine utilization [28]. On the other hand, warehouse scale computer workloads are composed of diverse applications with varying QoS requirements and priorities. Key applications, usually those that are user-facing and provide interactive service such as search, mail and maps, are latency sensitive and have fairly strict QoS requirements. Other applications such as backup service and file compression are batch applications that are not latency sensitive and have a lower QoS priority. When co-locating applications on a multicore platform without an effective QoS mechanism, the QoS of high priority applications may suffer unacceptable amounts of degradation [22, 34]. Also, high priority applications may even suffer more QoS degradation than low priority applications, resulting in unacceptable priority inversion. As a result, modern warehouse scale computers often resort to disallowing co-location of applications of a high QoS priority with any other applications, which translates to low machine utilization at the cost of millions of dollars [3]. This over-provisioning of compute resources is one of the major reasons the utilization in modern WSCs remains low, recently reported to be around 20% [24].

[Goal] The goal of this work (summarized in Figure 1) is to enable the direct manipulation of the contentiousness of low priority applications to ensure the QoS of a higher priority co-runner. Figures 1 (a) and (b) show the current options available to WSC operators. Figure 1 (a) shows the

CGO '12, March 31-April 4, San Jose, US

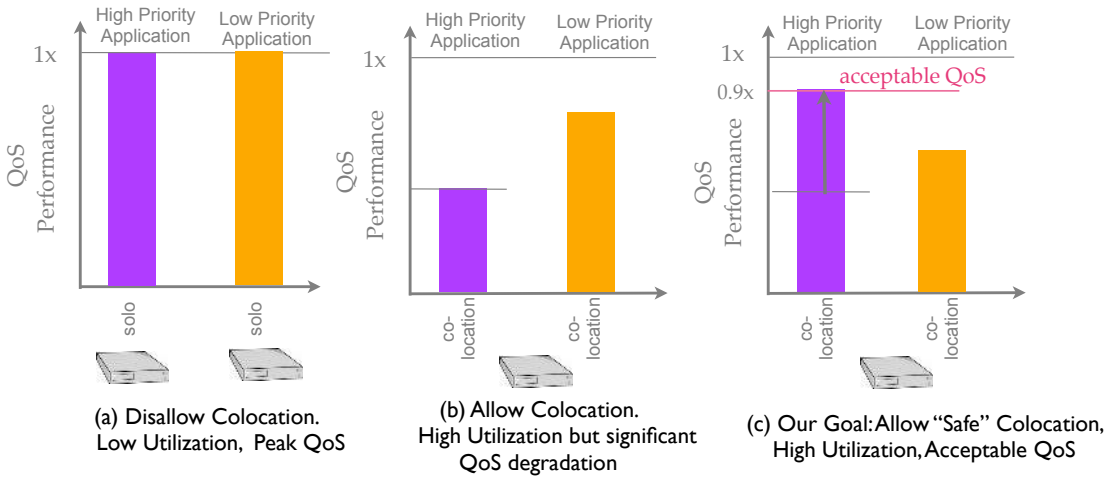

Figure 1: Motivation and Goal

impact of disallowing co-location. Although both low and high priority applications achieve a QoS of 100%, two machines are used. Figure 1 (b) shows the impact of simply allowing co-location. Both applications are co-located on one machine; however the QoS of the high priority application suffers greatly. We also have QoS priority inversion in this example. The goal of our approach is presented in Figure 1 (c). With the capability of dampening an application's contentiousness, we can reduce the interference of the low priority application to improve the QoS of the co-running high priority application. In this case, co-location can be allowed while achieving satisfactory QoS.

Although there has been much research attention on the problem of contention on multicore processors, there are currently no general software solutions for achieving the QoS management and enforcement of QoS priorities as described in Figure 1. Most of the prior research focuses on either novel architecture [6, 8, 9, 11] or scheduling [1, 4, 7, 12, 13, 17, 36, 40]. Although the hardware solutions have shown promising results using simulations, they cannot be applied to multicore platforms that are already in production or to be deployed in the near future. Meanwhile, although contention-aware scheduling may improve the overall performance or fairness of a multiprogrammed workload composed of a mixture of high contention and low contention applications, its effectiveness is dependent on the composition of the workload and it does not provide direct manipulation of the contentious nature of an application. In this work, we aim to provide a mechanism to trade a small amount of QoS of low priority applications to enable more "safe" co-locations and thus improve machine utilization in the WSC.

There are two key insights of our approach. Firstly, WSCs typically house a known set of long running applications, such as web search and maps, running for weeks and months at a time. A cluster-level scheduler maps multiple applications to each individual machine, and thus the co-location persists for this period until a job finishes running. The various QoS priorities of these applications are known throughout the lifetime of the WSC. In addition, binaries of these applications are available and the profiling can be performed continuously both in production and in test settings. Within this environment, a compilation approach is particularly useful for tailoring the binaries of these applications to "play nice" together. Secondly, in the era of multicores and the emerging computing domain of WSCs, the objectives of compiler optimization ought to be multifaceted. Simply optimizing each application for its own individual performance irrespective of the surrounding execution environment may not be ideal. In this work, we argue for the additional objective of optimizing for an application's "niceness," to reduce its potential interference to its co-running applications.

In this work, we present **QoS-Compile**, the first compilation approach to mitigating contention and QoS degradation for improved utilization in modern WSCs. The basic idea of QoS-Compile is firstly, to identify code regions that aggressively demand memory resources and may cause resource contention; and secondly, transform their code layouts to reduce their contentiousness thus reducing its interference to the QoS of its co-runners. To identify contentious code regions, we establish a model using performance counters and a regression analysis to score the contentiousness of a given sequence of executed code. A profiling run uses this model to pinpoint the static code regions that are most contentious. QoS-Compile then reduces the contentiousness of an application by throttling down the memory request rate of only the contentious regions. When no such regions are detected, our rate reduction transformations are not applied. This rate reduction in low priority applications improves the QoS of high-priority applications and enforces their relative priorities. QoS-Compile also provides a mechanism for users to control the tradeoffs between QoS and machine utilization by controlling the amount of rate reduction.

To the best of our knowledge, this paper is the first to address the QoS challenges caused by contention for multi-programmed workloads using compilation techniques. Specifically, this paper makes the following contributions:

- We present a prediction model and profiling analysis to identify code regions that may cause contention and performance interference. This allows us to target our

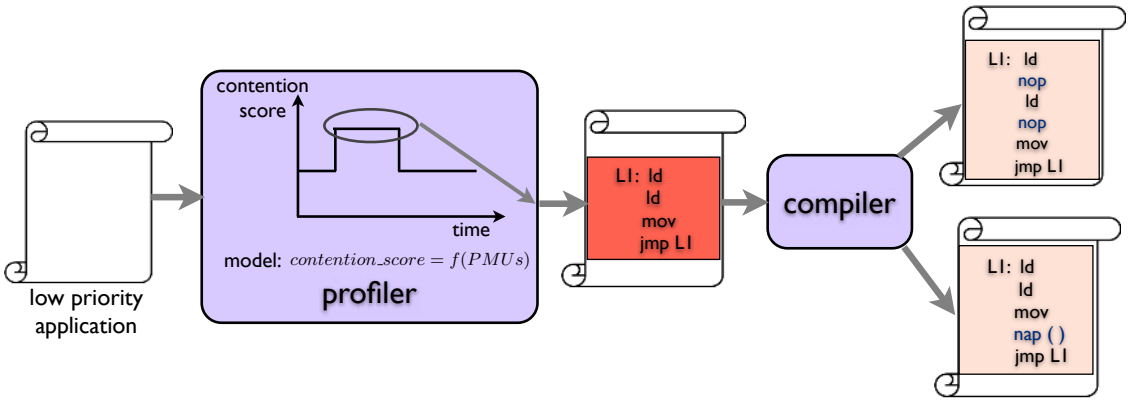

Figure 2: QoS-Compile Overview

compilation techniques only to the pinpointed code regions.

- We present two compiler techniques that dampen a code region's contentiousness and the potential performance interference it can cause to co-runners.
- We present the design of QoS-Compile and discuss key design decisions impacting the effectiveness of compiler-based dampening of memory pressure.
- We evaluate QoS-Compile using microbenchmarks, SPEC benchmarks and Google applications on several commodity multicore platforms and demonstrate its effectiveness.

Our experiments show that QoS-Compile can effectively pinpoint an application's contentious code regions, reduce the interference it causes to co-running applications and thus improve the QoS of its co-runners. In our experimentation, we find that QoS-Compile can improve the QoS of high priority co-running applications by 21% and also improve the machine utilization by 36% on average.

The rest of the paper is organized as follows: Section 2 presents the overview of QoS-Compile. Section 3 presents the profiling technique to identify contentious code regions. Section 4 presents the compilation techniques to reduce contention. Section 5 presents the evaluation. Section 6 presents prior work and Section 7 concludes.

2. QOS-COMPILE OVERVIEW

QoS-Compile consists of two steps. First, the application is profiled to identify its contentious code regions. Second, transformations are applied to these regions to reduce their contentiousness.

[**Identifying Contentious Regions**] Resource contention is only manifested during runtime, and as a result, a static code analysis to identify such code regions may not be feasible. Our technique uses a profiling analysis to characterize the memory resources usage of an application when it is running alone. The intuition is that if a code region aggressively uses shared memory resources (shared caches and memory bandwidth, etc) when executing, this region may interfere with a co-runner that is sensitive to contention. To predict a code region's contentiousness, we established a performance monitoring unit (PMU) based prediction model via regression. As Figure 2 shows, our profiler dynamically samples PMUs when an application is executing, estimates the contentiousness of code regions using the prediction model, and selects code regions that are above a certain contention threshold. Being able to pinpoint just the regions responsible for contention is a key benefit of QoS-Compile as we only throttle down the memory access rate of these regions. Applications may have short bursts of contentiousness or be contentious only during certain phases. Our compiler transformations are applied only to the code that is responsible for these bursts or phases.

[**Compiling for "Niceness"**] After identifying the contentious code regions, QoS-Compile then specializes the code layout of these regions to reduce their contentious nature, as shown in Figure 2. QoS-Compile is essentially a software *rate-based* technique as it throttles down the memory request rate of a low priority application, reducing the resulting pressure on the memory subsystem and allowing the neighboring high priority applications to consume more of these resources. In this work, we develop two transformations for memory request rate reduction: **padding** and **nap insertion**. Using these two transformations, QoS-Compile provides a wide range of *throttling granularities*. These granularities range from intermittent bursts of just a few instructions before a brief pause, to thousands of instructions before each longer nap. Our padding transformation provides fine granularity throttling while nap insertion provides coarser granularities. Both of these transformations include parameters for adjusting the amount of rate reduction, which in turn controls the amount of interference and QoS degradation suffered by co-runners. This tunability is important for achieving the desirable balance between QoS and machine utilization.

[**Using QoS Compile in a Modern WSC**] In modern WSCs, high priority latency-sensitive jobs, such as web-search and maps, are run on machines for weeks and months at a time. These jobs often use a fraction of the cores on a single machine. However, to protect their QoS, the co-location of other jobs on these machines is often disallowed. QoS-Compile can be used, on demand, to compile low priority batch jobs, such as video encoding/decoding and com-

pression, to enable their co-location on these underutilized machine resources. QoS-Compile can also be composed with a number of multi-versioning schemes [21] to enable its rate reduction transformations only when co-running with a high priority application.

3. IDENTIFY CONTENTIOUS REGIONS

In this section, we present the profiling analysis used to identify contentious code regions. The core component of our analysis is a model for the dynamic scoring of sequences of executed code. First we discuss how we constructed the model. We then describe how this model is used during a profiling run to identify the static code regions that are most contentious.

3.1 Modeling Contentiousness

[General Model] We use a linear model to combine the impact of contention in multiple shared resources, including last level cache (LLC), memory bandwidth and prefetchers. The contentiousness of a dynamically executed code region is determined by the amount of pressure the region puts on the shared memory subsystem. Thus, it can be predicted based on usage of shared resources, shown as the following equation,

$$C = a_1 \times LLC\_usage + b_1 \times BW\_usage + c_1 \times Pref\_usage, \quad (1)$$

where C is contention score, BW is bandwidth and $Pref$ is prefetchers.

Each code region may have a different combination of cache, bandwidth and prefetch usage. How contentious each code region is relative to other regions depends on the relative importance between cache, bandwidth and prefetcher contention. The relative importance is reflected as coefficients a_1, b_1 and c_1.

[Leveraging PMUs] Modern architectures provide numerous performance counters for various aspects of the microarchitecture. Our second step is to identify the appropriate performance monitoring units (PMUs) to estimate the terms in Equation 1.

BW_usage: It is fairly easy to quantify and measure bandwidth usage using PMUs. For example, we can use the number of cache lines the last level cache brings in from memory per second.

LLC_usage: It is challenging to measure cache usage using PMUs. PMUs can provide information on the cache access frequency and the cache miss rate, but currently they do not provide information on the cache footprint or occupancy. To approximate LLC usage, we measure how much data is fetched from the shared cache and not the memory for a given interval.

Prefetcher_usage: Not all architectures provide performance counters for all prefetchers. However, the main impact of prefetchers is reflected as increased bandwidth and cache usage. Thus, prefetcher usage can be estimated using cache and memory bandwidth usage.

Guided by the above insights, we identify the appropriate PMUs on the Intel Core i7 (Nehalem). On this platform, we identify the number of cache lines the last level cache brings in per millisecond (LLCLinesIn/ms), as shown in Figure 3, to capture the aggregate pressure an application puts on the bandwidth. We identify (L2LinesIn - L3LinesIn)/ms to estimate the shared L3 cache usage. It reports the rate of data being fetched into private caches from the shared cache. Because both L3LinesIn and L2LinesIn include the prefetchers' traffic, we do not need an extra PMU to measure the prefetcher usage. Using the above PMUs, Equation 1 becomes:

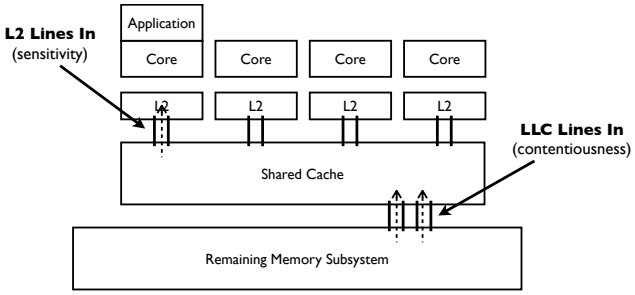

Figure 3: PMUs used for predicting contentiousness

$$C = a_1 \times (L2LinesIn\_rate - L3LinesIn\_rate) + b_1 \times L3LinesIn\_rate \quad (2)$$

where C is contention score.

[Regression] After identifying the appropriate PMUs, we use multiple regression to determine the coefficients in Equation 2. We use the **SmashBench** suite (Table 1), developed in Google, to train our model. SmashBench is composed of contentious kernels that span a spectrum of contentious memory access patterns and working set sizes. We measure each kernel's contentiousness using the average performance degradation it causes to other kernels within the suite when co-running. Using the measured contentiousness and the measured PMUs profile, including the average L2LinesIn/ms and L3LinesIn/ms, we then conduct regression analysis to determine the model coefficients (Equation 1). The regression result is:

$$C = 1.663 \times (L2LinesIn/ns - L3LinesIn/ns) + 8.890 \times L3LinesIn/ns + 0.044 \quad (3)$$

The `p` value for ($L2LinesIn/ns$ - $L3LinesIn/ns$) is 0.018, 5.11e-07 for $L3LinesIn/ns$, and 2.015e-06 for the entire regression. All are smaller than 0.5, indicating statistically significant effects. The `R-squared` is 0.8876, indicating a strong fit. The coefficients show the relative importance between the bandwidth usage and the LLC usage, indicating that memory bandwidth contention has a more dominating effect.

The regression results show that our model combines the contention of multiple resources and is highly indicative of the performance interference a code region may cause. The prediction accuracy of the model is evaluated in Section 5.

3.2 Identifying Code Regions

Identifying code regions based on the PMU model is fairly straightforward, and involves correlating PMU information with its corresponding source code. There are a number of approaches for conducting the correlation. In this work, we use a simple approach. We first record the application's

Benchmark	Footprint	Description
bst	4mb, 8mb, 50mb	random accessing a binary search tree
naive	4mb, 8mb, 50mb	random accessing an array
er-naive	4mb, 8mb, 50mb	fast random accessing an array
blockie	small, medium, large	a number of large 3D arrays. A portion of one array is continuously copied to another.
sledge	small, medium, large	two large arrays, copies data back and forth between arrays with this sledgehammer pattern.

Table 1: Contention Benchmarks Suite: Smash-Bench

PMU statistics (L2 and L3 lines in rate) every 1ms. Meanwhile, we record the number of instructions executed in every sample interval. These serve as markers in the dynamic instruction trace for the sequence of instructions that are responsible for the PMU data. We use the collected PMU profile and Equation 3 to calculate a contention score for every 1ms instruction interval. We then use a PIN [20] tool to replay the execution. Using the recorded interval markers we analyze the set of source level basic blocks that comprise the 1ms interval. We select the hottest set of basic blocks of that region, typically comprising more than 90% coverage of the interval, and assign these blocks the corresponding contentiousness score that was produced by our model.

4. COMPILER TRANSFORMATIONS FOR RATE REDUCTION

QoS-Compile provides two compilation techniques, *padding* and *nap insertion*, for both fine-grain and coarse-grain memory request rate reduction. In this section, we describe both of these techniques and discuss the tradeoffs between them.

4.1 Padding

Our *padding* transformation inserts non-memory instructions between memory instructions in a contentious code region. These instructions consume CPU cycles but do not issue memory requests. Therefore, in essence, they limit the amount of memory requests issued in a given time interval. When the amount of padding increases, the code region's pressure on the memory subsystem decreases. We implement padding by inserting no operation instructions (`nop`) in contentious code regions at the basic block level using MAO [10]. Padding provides a fine grain mechanism for reducing a code region's execution rate, memory request rate, and its interference to co-runners. Inserting these `nops` artificially inflicts a slowdown that can be as small as the number of cycles consumed by a single `nop`.

Application specific and microarchitecture specific factors need to be considered when deciding a sensible padding policy for a given interference reduction goal. The application specific factors include:

1. *The code region's memory characteristics.* The contentious level of a code region affects the amount of padding needed. The more contentious, the more padding needed. In addition, many memory characteristics such as the footprint affect the latency of memory instructions, which in turn affects the amount of padding needed. We discuss more about this effect shortly.

2. *Binary instruction characteristics.* The instruction mix, for example, the ratio of memory instructions (loads, etc) versus other instructions (CPU instructions) also needs to be considered. For a given amount of instructions, the more dense memory instructions are, the more padding may be required to reduce the pressure they cause to the memory system.

In addition, microarchitecture specific factors include:

1. How `nop`s are executed on the architecture;

2. The memory hierarchy design and the access latencies for different levels in the memory hierarchy.

Many of the above factors essentially affect the memory latency of instructions, which is important when deciding a padding policy for a given interference reduction goal. This is mostly because that an application can be stalled on the memory instructions when the data is being fetched. During this period, `nop`s may not have an effect on slowing down the application execution rate or memory request rate because the program is already stalled. For example, a `load` may take hundreds of cycles to complete. When stalled on a use, a large amount of `nop`s after this `load` may be useless for rate reduction. Therefore, each `nop`, depending on where it is inserted and the latency of memory instructions before it, may have a different impact on the memory request rate. This makes it difficult to accurately predict the rate reduction effect for a padding policy.

There are two main parameters for padding: *granularity* and *thickness*. Padding granularity is how often to pad (for example, every 3 instructions) and the thickness is how much `nop`s to insert at every insertion point. In this paper, given a list of contentious basic blocks identified by the QoS-Compile's profiler, we instrument padding at the beginning of each basic block. If a basic block contains more instructions than the specified padding granularity, we instrument within the basic block as well. The amount of padding inserted is determined by the thickness parameter. Generally, as discussed, the more dense memory instructions are, the longer latency they incur, the thicker padding is needed.

4.2 Nap Insertion

Our *nap insertion* technique inserts intermittent sleep to contentious code regions. Putting a contentious code region to epochal short "nap" mode reduces the pressure it puts on the memory subsystem and the interference it can cause to its co-runners. Similar to padding, two important parameters for nap insertions are *granularity* (how often the contentious code should nap) and *nap duration* (how long a nap interval should be, which is similar to padding thickness). However, comparing to padding, nap insertion is a much coarser-grain rate control as naps can occur for milliseconds at a time.

Another difference between nap insertion and padding is that, while padding indirectly controls the execution rate

by inserting instructions to prolong the execution time, nap insertion on the other hand, directly controls the time allotted between naps and the duration of the nap, thus having a more accurate and predictable rate reduction control than padding. To estimate the effect of nap insertion on memory request or execution rate reduction, we use the following equation:

$$R_{execution} = \frac{nap\_granularity}{nap\_granularity + nap\_duration} \quad (4)$$

where *nap_granularity* is the duration of the execution interval between inserted naps and *nap_duration* is the length of a nap. Given the execution rate $R_{execution}$ of a low priority application, L, we can estimate the improved QoS of its high priority co-runner, H. We denote H's improved QoS using QoS_{imprd_co-run}:

$$QoS_{imprd\_co-run} = 1 - (1 - QoS_{orig\_co-run}) \times R_{execution} \quad (5)$$

where QoS_{imprd_co-run} and QoS_{orig_co-run} are both normalized by H's QoS when running alone, and QoS_{orig_co-run} is H's QoS when co-running with the original L; QoS_{imprd_co-run} is H's QoS when co-running with the napping L. Padding can also use Equation 5 to predict the improved QoS when padded code region is reducing to a certain execution rate $R_{execution}$. However, as we discuss later, because of the coarse grain control, nap insertion is less skewed by the cooldown/ warmup effect.

Algorithm 1: Nap Insertion Algorithm

Input : *Binary*, *nap_granularity*, *nap_duration*
Output: *Binary* with inserted nap

```
1 instrument a global variable counter;
2 foreach BasicBlock in Binary do
3 |   if (BasicBlock.contention_score > contention_threshold) and
  |   (BasicBlock.coverage > coverage_threshold) then
4 |   |   InstrumentNap(BasicBlock, nap_granularity,
  |   |   nap_duration);
5 |   end
6 end
```

Algorithm 2: InstrumentNap

Input : *BasicBlock*, *nap_granularity*, *nap_duration*
Output: *BasicBlock* with inserted nap

```
1 At the beginning of the BasicBlock, instrument the following code:
  counter++;
2 if (counter > counter_threshold) then
3 |   cur_time ← read_time_stamp_register ;
4 |   if (cur_time − pre_time >= nap_granularity) then
5 |   |   sleep(nap_duration);
6 |   |   prev_time ← read_time_stamp_register ;
7 |   |   counter ← 0;
8 |   end
9 end
```

The main algorithm to conduct nap insertion is presented in Algorithm 1 and the instrumentation function is presented in Algorithm 2. The nap is only inserted to top basic blocks which are above a contention score threshold and are above a certain execution time coverage. The contention score of each basic block is generated by our profiling approach in Section 3. To reduce the overhead of checking the time stamp, we also use a counter to keep track of how many times the selected contentious basic blocks are executed and only to check the elapsed execution time when the counter is above a threshold.

4.3 Understanding Cooldown and Warmup

When applying a given amount of rate reduction to a code region, it may seem intuitive that it should provide the same amount of the interference reduction to a given co-runner. However, the granularity at which the intermittent rate reduction is conducted indeed matters. This is because of the *memory pressure cooldown and cache warmup* effect. Again, we use L to denote a low priority application to which we conduct padding or nap insertion, and H to denote a high QoS priority application whose QoS we are aiming to improve. When padding or a nap just starts to throttle down memory requests, it would take a while for L's pressure on the memory subsystem to cool down, especially if the data are residing below the cache. The memory system will still be serving L's requests issued before the padding or nap for a short period of time. Meanwhile, it takes a while for H to warm up the cache to achieve its optimal performance when it is running alone. We call this period the *cooldown/warmup window*. During this window, the yielding of shared resources is not instant and may negatively impact the effectiveness of the rate reduction mechanism. This effect may not be negligible, especially for padding, because padding happens at a fine granularity (a number of cycles or ns). However, the severity of this window may be greatly reduced for nap insertion because nap insertion can be at a coarser granularity. In Evaluation (Section 5), we will further investigate the interaction of nap granularity and this cooldown/warmup effect.

5. EVALUATION

In this section, we first evaluate the effectiveness of our prediction model and profiling technique in identifying contentious code regions. We then evaluate the application of our *padding* and *nap insertion* compiler transformations to reduce the contentiousness of an application and improve its co-runner's QoS. We then investigate the impact of leveraging QoS-Compile to improve utilization using both SPEC benchmarks and Google applications.

5.1 Setup and Methodology

Our evaluation is conducted on two platforms:

- *Intel Nehalem.* Intel Core i7 920 Quad Core with 2.67GHZ processors, 8MB last level cache shared by four cores and 4GB memory. This platform runs Linux 2.6.29.6 and GCC 4.4.6.
- *Intel Clovertown.* A dual socket Intel Clovertown (Xeon E5345). Each socket has 4 cores. Each 2 cores on the same socket are sharing a 4MB 16 way last level cache (L2). This platform runs Linux kernel version 2.6.26 and a customized GCC 4.4.3.

The workloads used in our evaluation include the *SmashBench* contentious kernel suite (summarized in Table 1), SPEC CPU2006, and large-scale Google applications such as websearch. SmashBench and SPEC experiments are conducted on the Intel Nehalem configuration and the Google experiments are conducted on production servers hosting the Intel Clovertown configuration. Each benchmark is compiled using GCC at the O2 level. All SPEC applications are run using **ref** inputs. Each experiment was conducted three times to calculate the average performance. SmashBench, SPEC and Google benchmark runs are fairly stable with a variance of 1% or less between runs.

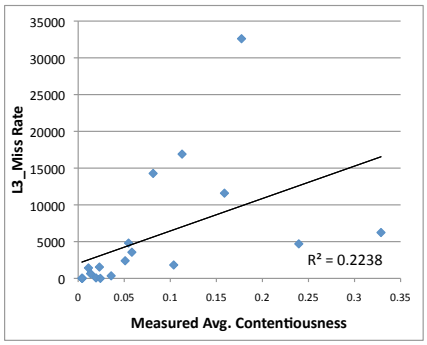

Figure 4: L3 Miss Rate is not strongly correlated with the real measured contentiousness

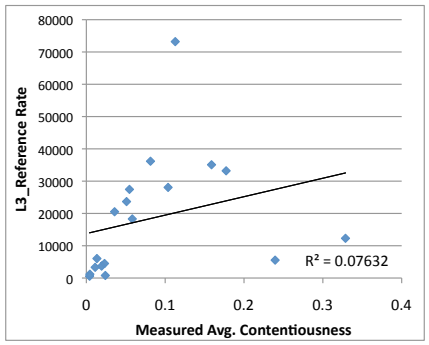

Figure 5: L3 Reference rate is not strongly correlated with the real measured contentiousness

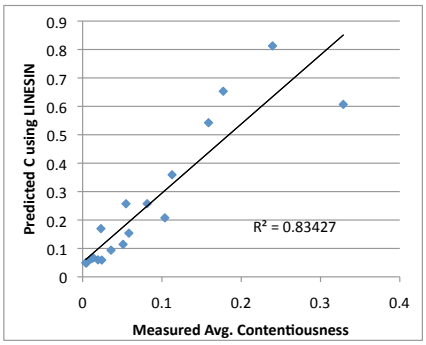

Figure 6: Predicted contention score using our model is highly correlated with the real measured contentiousness for SPEC benchmarks

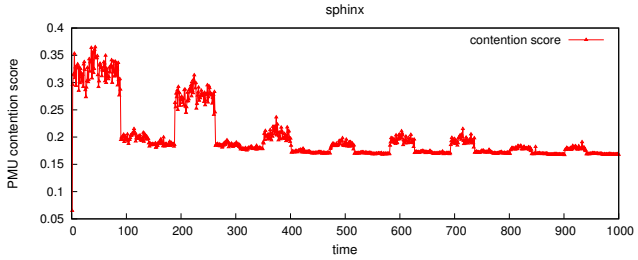

Figure 7: `Sphinx`'s PMU contention score calculated using our prediction model

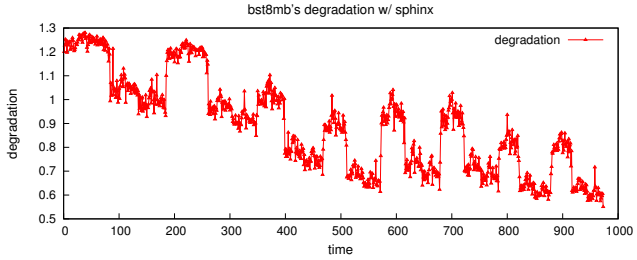

Figure 8: `Bst8mb`'s degradation when running with `sphinx`. The higher, the more degradation. Figure 7 trends similarly with this figure, indicating the profiler is identifying the correct contentious code regions.

5.2 Model for Code Region Identification

The key component of the profiling system is the PMU model used to correlate the memory subsystem activity of a code region to its contentious nature and potential for causing interference.

[**Model Accuracy**] To evaluate the accuracy of our PMU model (Equation 3), we compare our PMU model's predicted contentiousness of SPEC benchmarks with their real measured contentiousness. We profile each benchmark's PMUs (*L2LinesIn_rate* and *L3LinesIn_rate*), and calculate the predicted contentiousness using Equation 3 with the acquired PMU profiles. The prediction is then compared against each benchmark's observed contentiousness, measured as the average performance degradation it causes to a set of co-runners.

As a baseline, we compare our predictive model to state of the art estimators proposed by prior work [40]. Figure 4 and 5 show the results when using LLC miss rate and LLC reference rate to predict applications' contentiousness. The correlation coefficients (R) are 0.47 and 0.28, respectively, showing that neither LLC miss rate nor LLC reference rate alone can accurately indicate application contentiousness. Figure 6 presents our prediction results compared to the real measured contentiousness for SPEC CPU2006 benchmarks. Recall that our model is trained using a different set of benchmarks (e.g., SmashBench) and here we evaluate it on SPEC. For SPEC, the prediction's linear correlation coefficient R is 0.91, indicating that our prediction model can accurately score contentiousness.

[**Pinpointing Code Regions**] To evaluate the effectiveness of pinpointing the contentious code regions using our PMU model, we compare benchmarks' PMU model results with the degradation they cause to their co-runners. Figure 7 presents `sphinx`'s contention score calculated using its performance counter profile when it is running alone, based on Equation 3. The x-axis is time. Here `sphinx` is using `ref` input. The y-axis is the contention score using PMU model of `sphinx`'s execution phases. Figure 7 shows that `sphinx` is not evenly contentious through the entire execution, but, instead, there are several phases (humps in the figure) that are more contentious than the rest. Figure 8 presents `bst8mb`'s degradation when running with `sphinx`. This figure also presents the entire execution of `sphinx` using `ref` input. Comparison between Figure 7 and 8 shows that PMU contention score correctly identifies execution phases that are contentious (e.g cause more degradation to a co-runner). The execution phases with higher PMU contention score (humps in Figure 7 are consistent with the higher degradation (humps in Figure 8).

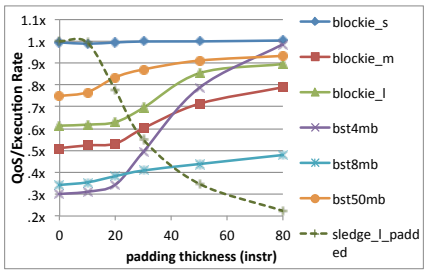

Figure 9: Padding `sledge_l`'s effect on its co-runner `blockie` and `bst`. As padding thickness increases, `sledge_l`'s execution rate deceases, `blockie` and `bst`'s QoS improves. The padding granularity is every 5 instructions

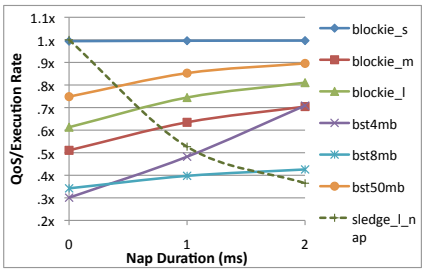

Figure 10: Napping `sledge_l`'s effect on co-runners, `blockie` and `bst`. Nap granularity is 1ms. As nap duration increases, `sledge_l`'s execution rate deceases, `blockie` and `bst`'s QoS improves.

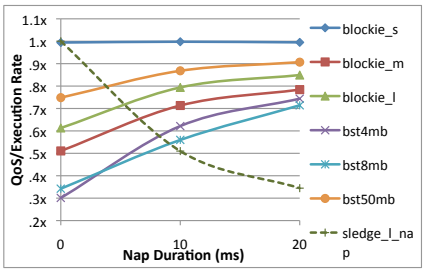

Figure 11: Napping `sledge_l`'s effect on co-runners. Nap granularity is 10ms.

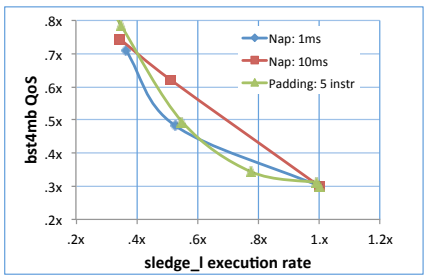

Figure 12: `sledge_l` padding vs. nap for `bst4mb`

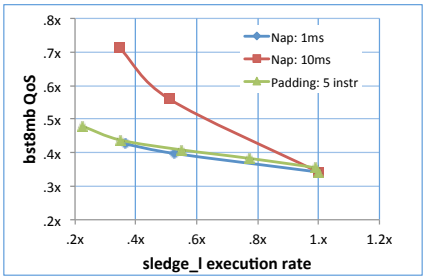

Figure 13: `sledge_l` padding vs. nap for `bst8mb`

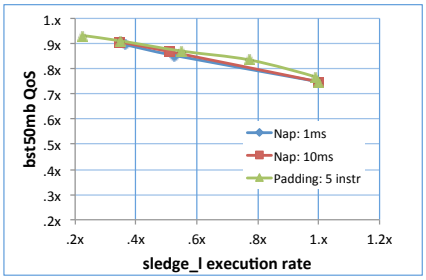

Figure 14: `sledge_l` padding vs. nap for `bst50mb`.

5.3 Compiler Transformations

In this section, we evaluate the two transformations used in QoS-Compile, *padding* and *nap insertion*, using the SmashBench suite. This evaluation focuses on the effectiveness of our transformations for improving a co-running application's QoS. We applied our transformations to the whole program of the contentious kernels without the use of the model to identify specific regions. All experiments in this section were conducted on the Intel Nehalem described in Section 5.1.

In Figures 9, 10, and 11 we show the QoS (in terms of execution rate) impact of allowing pairwise co-location of `sledge_l` (`sledge large`) with 6 co-runners when leveraging QoS compile. The dashed line shows the QoS of `sledge_l` and the solid lines shows the QoS of each of the 6 corunners when colocated with `sledge_l`. In these experiments, `sledge_l` is assumed to be our low priority applications while each of its 6 co-runners are assumed to be high priority. The x-axis shows various settings for padding and nap insertion. Figure 9 presents the results of applying padding to `sledge_l`, and Figures 10 and 11 show the results when applying nap insertion to `sledge_l`.

Figure 9 shows that, as the padding thickness increases, `sledge`'s execution rate decreases, and the QoS of `blockie` and `bst` improves. For example, when running with the original `sledge_l`, `blockie_l`'s normalized QoS is 0.6x of its solo optimal QoS. After we apply padding to `sledge_l`, `blockie_l`'s QoS is improved to almost 0.9x , which is a 50% improvement. An interesting observation is that the amount of improvement is not the same for various co-runners. For example, `bst8mb`'s normalized QoS when running with the original `sledge_l` is 0.35x, almost 3 times slower than when it is running alone. However after applying padding, its QoS is only improved to 0.5x. Another interesting observation is that the amount of interference reduction and QoS improvement slows down as padding thickness increases. The improvement is more significant around padding thickness 30 to 50, but for some benchmarks the improvement plateaus after 50. This indicates a potentially diminishing return of increasing padding thickness beyond a certain point.

Figures 10 and 11 show the results when applying nap insertion to `sledge_l`. The difference between these two figures is the napping granularity. Figure 10's granularity is 1ms, meaning that nap is inserted every 1ms of the execution. The x-axis shows the nap duration, ranging from no nap at all to 2 ms nap every 1ms of execution. Figure 11 shows the results when the nap granularity is 10ms. These figures demonstrate the effectiveness of nap insertion: as nap duration increases, co-runner's QoS improves. Comparing Figure 10 and Figure 11 also demonstrates the impact of the nap granularity. Interestingly, napping every 10ms performs significantly better than napping every 1ms for several co-runners. For example, for `bst8mb`, when running with `sledge_l_nap_10ms_20ms` (nap 10 ms every 20ms), its normalized QoS is above 0.7x of its solo optimal QoS, compared to only 0.5x when it is running with `sledge_l_nap_1ms_2ms`. This improvement is consistent with the *cooldown and warmup effect*.

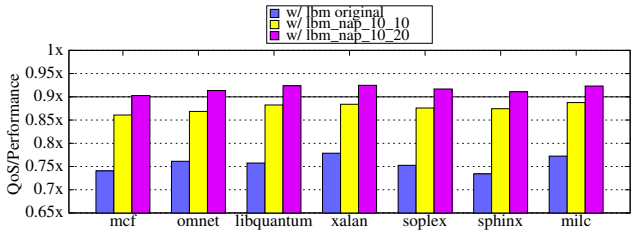

Figure 15: SPEC benchmark's performance when it is co-located with the original lbm, lbm with nap insertion (10ms, 10ms) and nap insertion (10ms, 20ms), normalized by each benchmark's performance when it is running alone

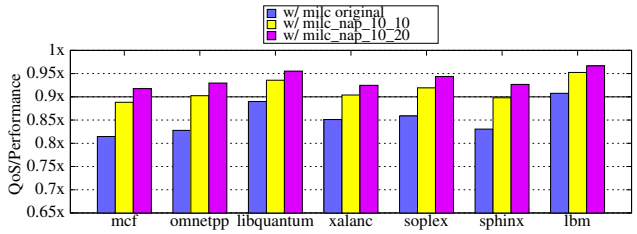

Figure 16: SPEC benchmark's performance when it is co-located with the original milc, milc with nap insertion (10ms, 10ms) and nap insertion (10ms, 20ms), normalized by each benchmark's performance when it is running alone

Figures 12, 13 and 14 further illustrate the different impact of padding and nap with various configurations. In each figure, the x-axis shows the `sledge_l`'s normalized execution rate. The y-axis shows its co-runners' normalized QoS. In each figure, we plot three lines showing the effect of three compilation techniques, padding, nap_1ms and nap_10ms. From these figures we can compare, with the same reduced execution rate for `sledge_l`, which technique achieves the best QoS improvement. Figures 12 and 14 show that nap and padding perform similarly for `bst4mb` and `bst50mb` as the three lines are very close to each other. However, Figure 13 shows that nap_10ms performs significantly better than the other two. For example, when `sledge_l` is running at 0.4x (40% of its original execution speed), nap_10ms improves `bst8mb`'s QoS to 0.65x compared to only 0.4x for both padding and nap_1ms. This result is consistent with the *cooldown and warmup* discussion in Section 4.2. Longer padding or napping granularity allows co-runners to warm up the cache and achieve better QoS performance. Since the experimental platform has a 8MB last level cache, among `bst4mb`, `bst8mb` and `bst50mb`, `bst8mb` is the most cache contentious benchmark, and therefore benefit the most from longer nap granularity. We also observe similar results when applying padding and nap insertion to other synthetic benchmarks, which are not shown here.

5.4 QoS-Compile: Put it All Together

In this section, we evaluate QoS-Compile, the combination of profiling to identify contentious code regions and compilation techniques to dampen contentiousness and improve the QoS of co-runners. The goal of this evaluation is to study the effectiveness of QoS-Compile in 1) improving the QoS of high priority applications when running with low priority applications; and 2) improving machine utilization, meaning that the low priority applications can still reasonably utilize the machine under the constraints of maintaining the QoS of high priority applications at a satisfactory level. We conduct this series of experiments using 8 memory-intensive benchmarks from SPEC CPU 2006 on the Intel Nehalem described in Section 5.1. Our evaluation in Section 5.3 shows that nap insertion performs better than padding. As such, we focus on nap insertion in this section.

[**Application level**] For each benchmark, we first profiled to sample its PMUs and calculated its contention score using our PMU model (Equation 3). We then identified its code regions (basic blocks) with contention scores that are above a specified threshold. In our experimentation, we used

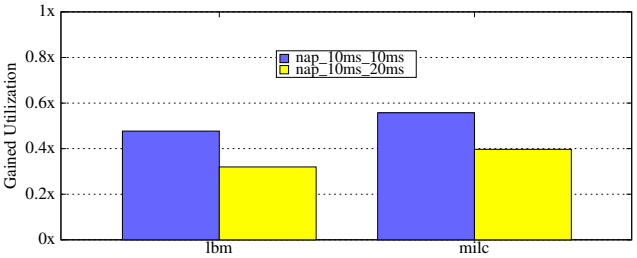

Figure 17: Gained Utilization when allow co-location.

0.3 as the threshold. We conducted nap insertion to those basic blocks using the algorithm presented in Section 4.2. To evaluate QoS-Compile's effectiveness, we conducted pairwise co-run experiments to co-locate a benchmark, presumed to be our low priority application, with 7 other benchmarks, presumed to be the high priority application, and measured the QoS degradation due to its interference.

Figures 15 and 16 present results for `lbm` and `milc`. Figure 15 shows the normalized performance of each SPEC benchmark when it is running with `lbm`. The x-axis shows each benchmark presumed to be the high priority co-runner. The y-axis shows its normalized performance. The higher the bars, the better. For each co-runner benchmark, a cluster of three bars show its performance when it is running with `lbm`, with `lbm_10_10` (`lbm` is napping 10ms every 10 ms) and with `lbm_10_20`, normalized by its performance when it is running alone. These 7 co-runner benchmarks are the memory-intensive SPEC benchmarks. We did not present results for other CPU bound SPEC benchmarks because in general they do not suffer degradation from memory resource contention. These figures demonstrate the effectiveness of QoS-Compile. QoS-Compile greatly improves `lbm`'s "niceness": reducing `lbm`'s interference to its co-runner and improving co-runner's QoS performance. For example, `mcf`'s QoS is improved 22%, from only 0.74x of its solo optimal QoS when it is running with the original `lbm` to above 0.9x of the optimal when it is running with the napping `lbm`. In general, every benchmark's QoS when running with `lbm_10_20` is above 90% of the solo optimal QoS. Figure 16 presents similar results for `milc`.

Because QoS-Compile can greatly improve QoS, it provides opportunities for warehouse scale computers to allow co-location knowing that using QoS-Compile, the QoS degra-

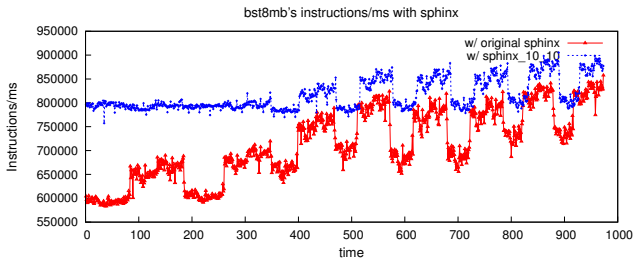

Figure 18: `bst8mb` running with `sphinx`

dation of the co-located high priority application would be within an acceptable threshold (10%, for example). Figure 17 shows the gained machine utilization when allowing co-location facilitated by QoS-Compile. Utilization is measured using `lbm_nap`'s normalized performance (execution rate normalized by the original `lbm` performance when it is running alone). For example, 48% gained utilization for napping `lbm_10_10` indicates that `lbm` is running at 48% of its original execution rate. That is, as opposed to disallowing co-location to ensure the QoS of the high priority application, using QoS-Compile, we allow 48% additional computation while protecting the QoS of its co-runner.

As we mentioned previously in Section 1, without QoS-Compile, WSC operators currently have only two options, either allow co-location and suffer a significant QoS penalty or disallow co-location and suffer a utilization penalty. As these figures together demonstrate, QoS-Compile allows users to trade a small amount of QoS to improve machine utilization. In this experiment, we allow 10% QoS degradation, and in return, gain 40% of utilization of the extra otherwise idle core. Changing the nap granularity and nap interval provides a knob that can be used to tune the tradeoff between QoS degradation and the amount of utilization gained. The more QoS degradation headroom, the more utilization.

[Phase level] QoS-Compile not only reduces the overall average QoS degradation, it also pinpoints the contentious regions and mitigates the QoS degradation those regions can cause when executing. This makes QoS-Compile also suitable for applications that only have phases of contention. To further evaluate QoS-Compile's effectiveness in pinpointing and managing the contentious phases, we sample the performance of co-runners throughout the entire execution to observe their performance variability due to interference. Figure 18 presents `bst8mb`'s performance (instructions/ms) when it is running with the original `sphinx`, compared to its performance when running with napping `sphinx` (`sphinx_10_10`, napping 10ms every 10ms). The x-axis shows time. We sample the entire execution of `sphinx` with `ref` input. The y-axis is `bst8mb`'s performance. `Bst8mb` is a contentious kernel and when it is running alone it has quite stable performance. Therefore the performance variability shown in the figure is purely due to interference from `sphinx`. As the figure shows, during the early half of the execution, original `sphinx` causes significant performance degradation to `bst8mb`, demonstrated by the low IPS during the first 400 samples. QoS-Compile correctly identifies the contentious phase and improves the `bst8mb`'s IPS greatly. For the later half of the execution, the QoS-Compile also identifies `bst8mb`'s performance valleys and improves it greatly.

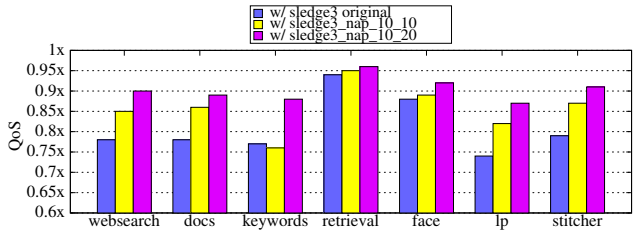

Figure 19: Google benchmark's performance when it is co-located with the original sledge3, sledge3 with nap insertion (10ms, 10ms) and nap insertion (10ms, 20ms), normalized by each benchmark's performance when it is running alone

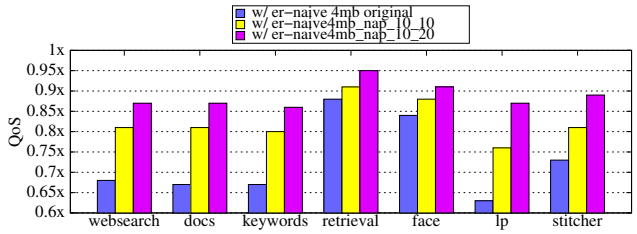

Figure 20: Google benchmark's performance when it is co-located with the original er-naive4mb, er-naive4mb with nap insertion (10ms, 10ms) and nap insertion (10ms, 20ms), normalized by each benchmark's performance when it is running alone

5.5 Google Applications

To evaluate our compilation technique's effectiveness in improving co-runner's QoS, we also conducted experiments using several large-scale warehouse scale computer applications. The experimental platform is an Intel Clovertown machine used in production (as described in Section 5.1). The production applications are presented in Table 2. The QoS metric for each application is the application-specific performance metric in its internal SLA, also presented in Table 2. The load for each application is a trace of large amount of real world queries in production WSCs. A load generator was set up to feed the queries to these applications. The performance shown is applications' stable behavior after the initialization phase, and the performance is stable between runs. Figure 19 and Figure 20 present results. In these experiments, each Google application is co-located with 2 threads of SmashBench benchmarks. Figure 19 presents Google applications' QoS when co-located with `sledge_1`. The x-axis shows each Google application. And the y-axis is each application's normalized performance. Each application's QoS are measured in 3 running scenarios presented as a cluster of three bars: when it is co-located with 2 threads of original `sledge_1`, with 2 threads of napping `sledge` that naps 10ms every 10ms, and with `sledge` that naps 20ms every 10ms. Each application's QoS performance is normalized to its performance when it is running alone. Figure 20 presents Google application's QoS performance when co-located with a cache contentious benchmark, `er-naive4mb`. Figure 19 and Figure 20 demonstrate that nap insertion is effective in improving an application's "niceness" and improv-

workload	description	metric
websearch	Websearch scoring and retrieval	(QPS) queries per sec
cluster-docs	Unsupervised Bayesian clustering tool to take keywords or text documents and "explain" them with meaningful clusters.	throughput
cluster-keywords	Unsupervised Bayesian clustering tool to take keywords or text documents and "explain" them with meaningful clusters.	throughput
goog-retrieval	Web indexing	query latency (ms)
maps-detect-face	Face detection for streetview automatic face blurring	user time (secs)
maps-detect-lp	OCR and text extraction from streetview	user time (secs)
maps-stitch	Image stitching for streetview	user time (secs)

Table 2: Production Warehouse Scale Computer Applications

ing its co-running Google applications' QoS. For example, nap insertion improves `websearch`'s QoS from 0.77x to 0.9x when running with `sledge_l`, and from 0.68x to 0.87x when running with `er-naive4mb`. QoS-Compile can improve QoS significantly and provides warehouse scale computer operators with flexibility of allowing co-location with a slight hit on QoS. For example, if warehouse scale computer scheduler specifies that 0.9x of the optimal peak QoS is an acceptable threshold for `websearch`, with QoS-Compile, we can allow co-location of `websearch` with other co-runner such as `sledge_l` to improve the machine utilization. Without QoS-Compile, 0.65x of its solo QoS when running with the original `sledge_l` may be too significant to allow co-location, and thus leaving the machine under-utilized.

6. RELATED WORK

[Hardware Approaches] Hardware techniques such as cache and bandwidth partitioning and source throttling to improve performance and fairness on multicores have received much research attention [6, 16, 19, 26, 27, 33]. In addition, hardware platforms that enforce QoS priorities are proposed [8, 11]. These studies have shown promising future directions for hardware designers; however they require hardware changes and are not yet available in commodity chips. Herdirch et al. [9] and Zhang et al. [38] use clock modulation and core-specific dynamic voltage scaling to throttle down the execution to improve performance fairness or the performance of high priority applications. However, these techniques do not isolate contentious code regions and do not mitigate the effects of the cooldown/warmup window.

[Contention-aware Scheduling] An important software approach to mitigating contention is contention aware scheduling [1,4,7,13,17,25,36,40]. Scheduling techniques decide what applications should be co-running together to improve performance or performance isolation. In contrast to scheduling, the effectiveness of our compiler approach does not depend on having a balanced mix of high-contention and low-contention applications since we directly manipulate the contentiousness of an application to improve QoS. Even when the majority of the applications in the workload are contentious, our approach remains effective. In addition, our approach is complimentary to scheduling as it can be used after cluster-level scheduling decisions are already made (co-running applications on the same machine are decided), but the QoS goal for latency-sensitive applications is not met. In addition, our compiler approach provides tunability for adjusting the amount of rate reduction, which in turn controls the amount of interference and QoS degradation. This tunability is particularly useful in datacenters as it facilitates operators to achieve the desirable balance between QoS and machine utilization.

One important component for contention-aware scheduling is the indicators for application contention characteristics. This is related to our prediction model to identify contentious code regions. Most prior work uses last level cache miss rate or reference rate as an indicator of how contentious an application is [17, 40], which we have shown is less accurate than our prediction model. Our PMU-based approach is also more lightweight than techniques that profile footprint or reuse distance to predict applications' contention nature [13, 35]. Techniques that study the interaction between threads within a multithreaded application by dynamic instrumentation is also proposed [39].

[Other Software Approaches] Software solutions to reduce cache contention using page coloring/remapping have been proposed [5, 18, 31]. Most page coloring methods require significant modifications to the kernel and the knowledge of the cache design details, which are often highly guarded industry secrets. Recently, Mars et al. [23] propose a software runtime that detects and responds to contention online. Their technique shutters an application's execution to detect contention, while our approach first identifies contentious code regions and then applies static 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 [14, 15, 32, 37]. Most such research focuses on compilation techniques to improve cache sharing for a multi-threaded application. Our approach does not address the cache sharing and contention among sibling threads of an application. Instead, we are proposing compilation techniques that manipulate how applications interact with each other in terms of contending for the memory resources. Methods to reduce cache pollution by compiler-directed use of non-temporal move instructions [29] and non-temporal prefetch instructions are also proposed [30]. These techniques apply to a subset of contentious code that have special characteristics while our approach is more general and does not make special assumptions of the contentious code regions.

7. CONCLUSION

In this paper, we have presented QoS-Compile, the first compilation approach that statically manipulates application contentiousness to enable the co-location of applications with varying QoS requirements, and as a result, can

greatly improve machine utilization. Using a novel prediction model, QoS-Compile first pinpoints an application's contentious code regions that tend to cause performance interference. QoS-Compile then transforms those regions to reduce their contentious level. In this work we have shown that binary code transformations to throttle down the execution rate and the memory access rate of the contentious regions in low priority applications is an effective approach to reduce their interference to high priority applications. Through our experimentation, we find that QoS-Compile improves applications' QoS performance by 21% and machine utilization 36% on average. In the era of multicores and the emerging computing domain of WSCs, the objectives of compiler optimization ought to be multifaceted. In this work, we argue for the additional objective of optimizing for an application's "niceness," to reduce its potential interference to its co-running applications.

8. REFERENCES

[1] 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.

[2] L. Barroso, J. Dean, and U. Holzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2):22–28, 2003.

[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. Hundt, E. Raman, M. Thuresson, and N. Vachharajani. Mao: An extensible micro-architectural optimizer. In *CGO 2011*, pages 1 –10, Apr 2011.

[11] 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*, Jun 2007.

[12] Y. Jiang, X. Shen, J. Chen, and R. Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. *PACT '08*, Oct 2008.

[13] Y. Jiang, K. Tian, and X. Shen. Combining locality analysis with online proactive job co-scheduling in chip multiprocessors. *HiPEAC 2010*, pages 201–215, 2010.

[14] M. Kandemir, S. Muralidhara, S. Narayanan, Y. Zhang, and O. Ozturk. Optimizing shared cache behavior of chip multiprocessors. *Microarchitecture, 2009*, pages 505–516, 2009.

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

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] J. Mars and R. Hundt. Scenario based optimization: A framework for statically enabling online optimizations. CGO '09, pages 169–179, Washington, DC, USA, 2009. IEEE Computer Society.

[22] 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.

[23] J. Mars, N. Vachharajani, R. Hundt, and M. Soffa. Contention aware execution: online contention detection and response. *CGO '10*, Apr 2010.

[24] D. Meisner, B. T. Gold, and T. F. Wenisch. Powernap: eliminating server idle power. ASPLOS '09, pages 205–216, New York, NY, USA, 2009. ACM.

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

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

[27] 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.

[28] P. Ranganathan and N. Jouppi. Enterprise it trends and implications for architecture research. *HPCA 2005*, pages 253–256, 2005.

[29] 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.

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

[31] 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.

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

[33] S. Srikantaiah, M. Kandemir, and M. Irwin. Adaptive set pinning: managing shared caches in chip multiprocessors. *ASPLOS XIII*, Mar 2008.

[34] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. The impact of memory subsystem resource sharing on datacenter applications. ISCA '11, pages 283–294, New York, NY, USA, 2011. ACM.

[35] X. Xiang, B. Bao, T. Bai, C. Ding, and T. Chilimbi. All-window profiling and composable models of cache sharing. PPoPP '11, pages 91–102, 2011.

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

[37] 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.

[38] X. Zhang, S. Dwarkadas, and K. Shen. Hardware execution throttling for multi-core resource management. *Proceedings of the 2009 conference on USENIX Annual technical conference*, page 23, 2009.

[39] Q. Zhao, D. Koh, S. Raza, D. Bruening, W. Wong, and S. Amarasinghe. Dynamic cache contention detection in multi-threaded applications. *VEE 2011*, pages 27–38, 2011.

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