Contention Aware Execution: Online Contention Detection and Response

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

Neil Vachharajani   Robert Hundt
Google, Mountain View, California
{nvachhar, rhundt}@google.com

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

Abstract

Cross-core application interference due to contention for shared on-chip and off-chip resources pose a significant challenge to providing application level quality of service (QoS) guarantees on commodity multicore micro-architectures. Unexpected cross-core interference is especially problematic when considering latency-sensitive applications that are present in the web service data center application domains, such as web-search. The commonly used solution is to simply disallow the co-location of latency-sensitive applications and throughput-oriented batch applications on a single chip, leaving much of the processing capabilities of multicore micro-architectures underutilized. In this work we present a **Contention Aware Execution Runtime (CAER)** environment that provides a lightweight runtime solution that minimizes cross-core interference due to contention, while maximizing utilization. CAER leverages the ubiquitous performance monitoring capabilities present in current multicore processors to infer and respond to contention and requires no added hardware support. We present the design and implementation of the CAER environment, two separate contention detection heuristics, and approaches to respond to contention online. We evaluate our solution using the SPEC2006 benchmark suite. Our experiments show that when allowing co-location with CAER, as opposed to disallowing co-location, we are able to increase the utilization of the multicore CPU by 58% on average. Meanwhile CAER brings the overhead due to allowing co-location from 17% down to just 4% on average.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—parallel programming; 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

Keywords contention, multicore, cross-core interference, dynamic techniques, online adaptation, execution runtimes

1. Introduction

Multicore architectures are ubiquitous and have become the norm in computing systems today. These architectures dominate in many domains, including those with quality of service (QoS) and low latency requirements. Multicore architectures are composed of a number of processing cores, each with a private cache(s), and typically larger caches that are shared among many cores [13]. Other shared system resources include the bus, main memory, disk, and other I/O devices. When processes and threads are executing in parallel on a single multicore CPU we say they are *co-located*. Co-located processes and threads place varying amounts of demand on these resources; this demand can often lead to *contention* for these resources. Resource contention directly impacts application performance. When an application's performance is negatively affected by another application executing on a separate core, we call this *cross-core interference*.

Application priority and quality of service requirements often cannot withstand unexpected cross-core interference. For example, applications commonly found in the web service data center domain such as search, maps, image search, email and other user facing web applications are *latency-sensitive* [4, 7]. These applications must respond to the user with minimal latency, as having high latency displeases the user. Data centers for web services classify applications as either being *latency-sensitive* or as throughput-oriented *batch* applications, where latency is not important [4]. To avoid cross-core interference between latency-sensitive and batch applications, web service companies simply disallow the co-location of these applications on a single multicore CPU. Using this solution may leave the CPU severely underutilized, and is a contributing reason to the server utilization of these data centers often being 15% or less [18]. Low utilization results in wasted power, and lost cost saving opportunities.

Much research effort has been spent developing QoS and fairness mechanisms [5, 11, 14, 15, 17, 21], simulated approaches for cache resource management [23–25, 27] , and to better understand the algorithmic and theoretical characteristics of cache contention [2, 3, 6, 16]. However to date, there is no readily deployable approach to both minimize cross-core interference due to contention, and maximize utilization, for existing commodity multicore architectures.

In this work we propose such a solution. We present an execution environment, the **Contention Aware Execution Runtime (CAER)** environment, which is capable of online contention detection and response on current commodity hardware. The insight and opportunity that drives the design of CAER comes from the ubiquitous availability of performance monitoring capabilities in today's hardware. These performance monitors are capable of collecting information about dynamic application behavior in hardware without added overhead to the application. In this work we exploit the available performance information to gain information about cross-core interference due to application contention. Using this information we are able to detect and respond to this cross-core interference.

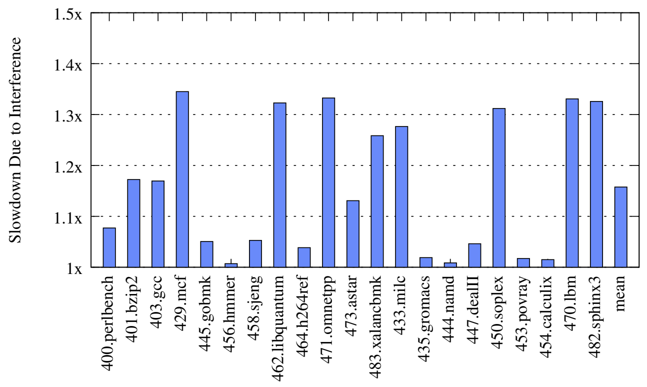


Figure 1. Performance degradation due to contention for shared last level cache on Core i7 (Nehalem) while running alongside lbm.

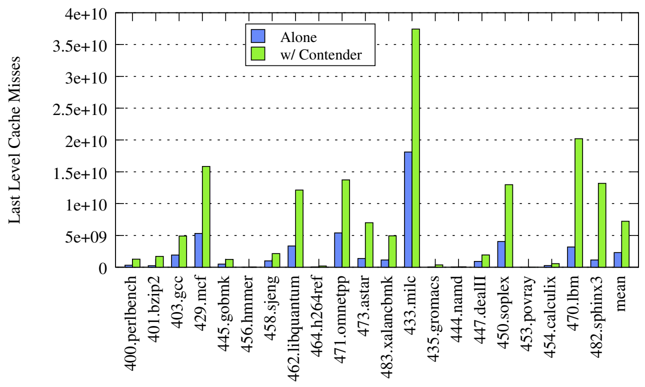


Figure 2. Increase in last level cache misses when running with contender.

CAER is composed of a lightweight runtime on which all applications of interest run. This runtime classifies these applications into the *latency-sensitive* and *batch* categories mentioned earlier. CAER dynamically probes the hardware *performance monitoring unit* (PMU) to collect information about the applications it hosts. This information is continually collected and analyzed throughout the lifetime of all applications running on CAER.

We have designed and evaluated two contention detection heuristics: **Burst Shutter**, and **Rule Based** techniques. These two heuristics are used by CAER to detect contention online. When contention is detected, CAER dynamically adapts the batch applications to minimize the contention. In our current prototype, we adapt by throttling down the execution of the batch applications to relieve the pressure on the contended resource. If no/low contention is detected, CAER allows the application to run more aggressively to maximize utilization.

To evaluate our approach we developed a working prototype of CAER including implementations of the two contention detection and response heuristics, and deployed CAER on current multicore architecture. Using the SPEC2006 benchmark suite we co-located multiple instances of different benchmarks simultaneously executing on a Intel Core i7 (Nehalem) Quad Core machine. Allowing co-location with CAER, as opposed to disallowing co-location, we are able to increase the utilization of the multicore CPU by 58% on average. Meanwhile CAER brings the overhead due to allowing co-location from 17% down to just 4% on average.

The contributions of this work include:

- A framework and runtime environment that addresses the problem of cross-core interference that can be deployed on current multicore architecture.
- The design and analysis of two online contention detection heuristics, their algorithms, and an evaluation of each.
- A thorough evaluation of the CAER runtime system on commodity hardware with the SPEC2006 benchmark suite.

Next in Section 2 we explore the problem of cross-core interference and motivate our work. We then discuss the design and architecture of CAER in Section 3. In Section 4 we describe the design of our two online contention detection heuristics and discuss responses to detection in Section 5. We describe our experimental setup and present results in Section 6, present related work in Section 7, and finally conclude in Section 8.

2. Problem and Motivation

Current multicore chip design in commodity hardware is composed of unshared and shared caches. For example, Intel's Core 2 Duo architecture has 2 cores, each with a private L1 cache and a single L2 of 4mb shared between the two cores. Intel's new Core i7 (Nehalem) architecture has 4 cores, each with private L1 and L2 caches and a single 8mb shared L3 cache for all 4 cores [13]. These types of shared memory multicore architectures are common in the data center space. When the workload of the individual application processes and threads executing on these multicore processors fits neatly into private caches, there is no cross-core interference (assuming coherence traffic is at a minimum). When the size of an application's working set exceeds the size of the private cache, the working set spills over into the larger shared caches. The shared last level cache presents the first level of possible contention. Contention can also exist later in the memory subsystem such as contention on the bus, in the memory controller, for shared memory, disk, etc. However much of the contention in these levels manifest themselves as traffic off-chip and thus show up as misses in the last level cache on the chip.

In this work, our strategy is to monitor activity in the last level of cache to detect contention and focus on minimizing contention in this level of the cache. When more than one application is using the shared last level of cache heavily, and the data is not shared, contention occurs. One way to address this problem is to increase the size and associativity of the cache. However, although cache sizes have been increasing with every generation of processors, we are still far behind the demands of today's application workloads. Figure 1 shows the degradation in performance of a set of applications due to cross-core interference caused by cache contention. This experiment was run on a state of the art general purpose processor (Intel Core i7 920 Quad Core), and demonstrates the impact of just two applications contending on a multicore chip for a large 8mb, 16way associative, shared, last level cache. The applications shown come from the SPEC2006 benchmark suite. Each application was first run alone on the quad core chip, then with the `lbm` benchmark running alongside on a neighboring core. The bars in Figure 1 shows the slowdown of each benchmark running alongside `lbm`. `Lbm` is an example of an application with aggressive cache usage. An application that is more affected by `lbm` implies that that application is also aggressive with its cache usage. Remember this data shows just two applications running on a quad core machine with a large cache designed to handle the load from four cores simultaneously doing work. In many cases we see a performance degradation exceeding 30%.

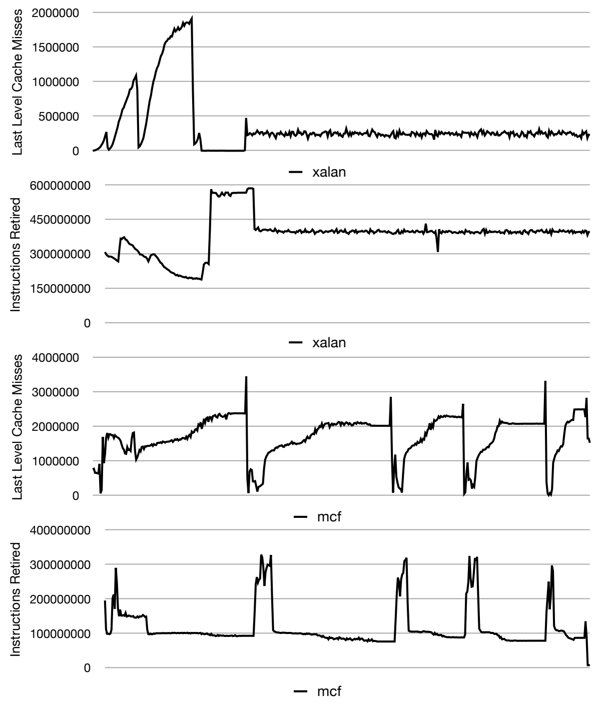


Figure 3. Correlating last level shared cache misses, and reduction in instruction retirement rate.

Figure 2 shows the increase in last level cache misses when running with a contender. It is important to notice the delta in cache misses between the application running alone and when it is running with the contender. It is also important to get a sense of the absolute number of misses for each and how that impacts its sensitivity to contention. Having a 150% increase in cache misses impacts performance much less as the absolute number of misses goes down. From this graph it is clear that the more last level cache misses an application experiences, the more sensitive it is to cross-core interference.

For the remainder of this work we define *utilization* of a multicore processor as

$$U = \frac{\sum_{i=1}^{N} \frac{R_i}{R_i+I_i}}{N} \qquad (1)$$

for some time, where N is the number of cores on the chip, R_i is the amount of time spent running on core i, and I_i is the amount of time idle on core i.

3. A Solution with CAER

Our goal is to address the contention in the shared caches of current multicore chip design by minimizing the cross-core interference penalty on latency-sensitive applications while maximizing chip utilization. To do so we have developed CAER, a *contention aware execution runtime* environment.

3.1 Inferring Contention

Hardware performance monitoring capabilities are ubiquitous in today's chip micro-architectures [13]. These hardware performance monitors provide realtime micro-architectural information about the applications currently running on chip. As the counters record this information, the program executes uninterrupted, and thus recording this online profiling information presents no instrumentation overhead. These capabilities enable new opportunities for online and reactive approaches, and can be leveraged with one of the many software APIs, such as PAPI [19] or Perfmon2 [9]. To build our solution, we use Perfmon2 as it is one of the most robust and flexible PMU interfaces, and supports a wide range of micro-architectures.

The basic premise of our solution is that information from PMUs can be used in a low/no overhead way to infer contention. In this work we focus on the shared last level cache (LLC) miss behavior. Last level cache misses directly (and negatively) impact the instruction retirement rate (i.e. IPC). Figure 3 illustrates this phenomenon with two SPEC2006 benchmarks that exhibit clear LLC miss phases. These benchmarks were run on their `ref` inputs to completion. The x-axis represents time from beginning of the application run to the end in all four of the graphs presented. Figure 3 shows two pairs of graphs, each pair correlating the LLC miss rate over time to the instruction retirement rate over time. We can see clear and compelling evidence of the inverse relationship between the number of LLC misses and the retirement rate.

CAER is based on the hypothesis that if two or more applications are simultaneously missing heavily in the last level shared cache of the micro architecture, they are both making heavy usage of the cache and probably evicting each others data (i.e. contending). This contention then leads to increased cache misses in both applications, which is evident in Figure 2. We believe that if we can dynamically monitor and analyze the chip wide information about thread/core specific impact on the last level cache misses we should be able to detect contention and thusly respond to this contention.

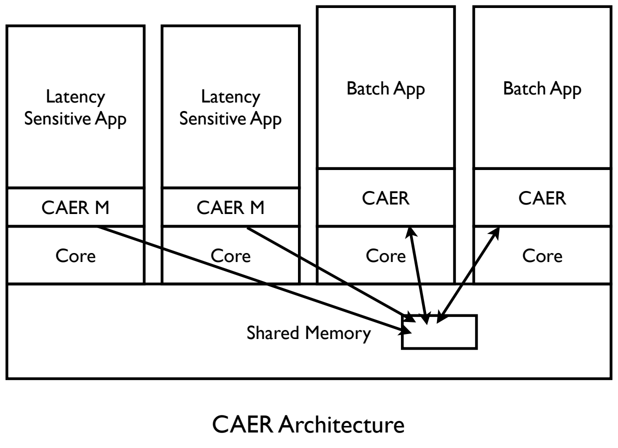
3.2 Architecture of CAER

The design and architecture of the CAER execution environment is presented in Figure 4. To the left of the diagram we present the overall design vision of the CAER environment, and to the right we present the actual working prototype we have implemented for this study. In the scenario presented on the left of the diagram we have two latency-sensitive applications or threads, and two batch applications or threads. In order to monitor and collect thread/core specific performance information on current hardware, we must issue the performance monitoring unit (PMU) configuration and collection directives on the particular core hosting the application of interest. For this reason a virtual layer must be present beneath all application threads of interest. These CAER virtual layers are cooperative and must share information, respond, and adapt to each other.

CAER's cooperation is accomplished via shared memory using a communication table as is shown in Figure 4 (arrows pointing into the table). Notice that the virtual layer (CAER M) beneath the latency-sensitive applications appear thinner in Figure 4. These (monitor) virtual layers are more light weight than the main CAER engines and only are responsible for collecting PMU data and placing this data in the communication table. The main CAER engines that lie underneath the throughput-oriented batch applications processes this information and perform the contention detection and response heuristics. CAER only applies any dynamic adaption or modifications on the batch application. The latency applications always remain untouched.

The CAER runtime employs a *periodic probing* approach [20], meaning information is gathered and analyzed by the virtual layer intermittently. Using a timer interrupt the environment periodically reads and restarts the PMU counters. Periodic probing has shown to be an extremely low overhead approach to perform lightweight online application monitoring.

In this work the CAER runtime uses a period of one millisecond. Every millisecond each CAER runtime probes their relevant performance monitoring units and reports last level cache information to the communication table. This table records a window of



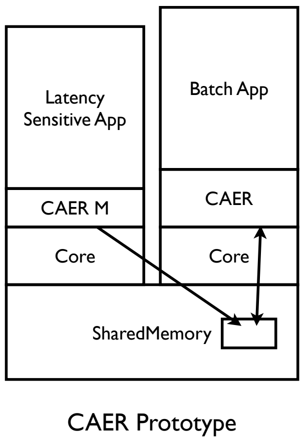


Figure 4. Architecture of our Contention Aware Execution Runtime

sample points, which allows us to observe trends of many samples. The main CAER engines that lie under the batch processes detect and react to contention. Note that all of the batch processes/threads must react together. Reaction directives are also recorded in the table, and all batch processes must adhere to the reaction directives. In the current design of CAER, these directive include pausing and staggering execution.

Our prototype is shown to the right of Figure 4. This instance of CAER supports two applications, one running atop CAER M, and the other on the main CAER engine. The CAER runtime is statically linked into the binary. Our prototype is fully functional and, as it is shown in Section 6, effective on real commodity hardware.

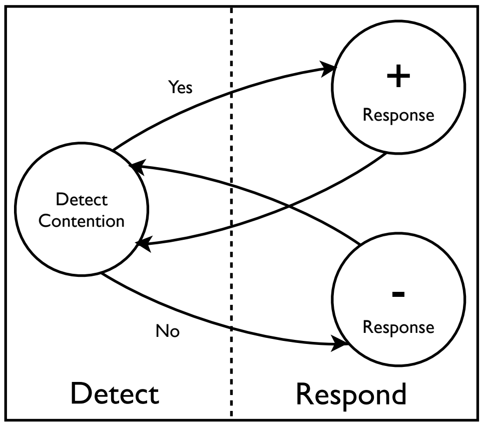


Figure 5. Basic Detection Response

The diagram in Figure 5 shows the contention detection and response phases used in the CAER runtime that lies under the batch applications. Throughout execution CAER resides in one of these states and continually transitions among these states. After CAER performs its contention detection heuristic, either contention, or the absence of contention is asserted, and we enter into the relevant response state as shown with the `yes` and `no` transitions in Figure 5. We call the state where contention is asserted the *c-positive* response, and the state where the absence contention is detected the *c-negative* response. The next section explores the heuristics and methods by which we detect contention corresponding to the left side of Figure 5, and the subsequent section explores CAER's contention responses corresponding to the right side.

4. Detecting Contention with CAER

Before CAER can react to contention in the shared cache, it must first detect that the applications are indeed contending. We have developed two heuristics for this task: a *burst shutter* approach and a *rule based* approach. These heuristics run continuously throughout the lifetime of CAER to detect and respond to contention.

4.1 Burst-Shutter Approach

If our batch application's execution is going to increase the last level cache misses in the neighboring latency-sensitive application, we should be able to see that spike in misses when the batch application has a burst of execution. That is, if the latency-sensitive application is running alone while the batch application is halted, when the batch application then has a burst of execution, we should see a sharp increase in the last level cache misses of the latency-sensitive application. We perform this analysis online as follows:

1. We have a number of periods where we halt the execution of the batch application and collect samples of the last level cache misses of the latency-sensitive application.
2. We then record the average last level cache miss rate.
3. We then have a number of periods where we execute the batch at full force (i.e. burst) and record the misses of the latency-sensitive application.
4. We calculate the average miss rate for these periods.
5. If the number of cache misses are significantly higher in the burst case, we assert the batch application is impacting the miss rate of the latency-sensitive application and report contention, else we report no contention.

The corresponding algorithm is presented in Algorithm 1. There are a number of parameters that can be tuned. We must determine how long (as in how many periods) we would like to halt the batch process's execution, how long the burst should last, and how high the sharp increase should be before asserting contention. In Algorithm 1 these parameters correspond to setting the `switch_point`, `end_point` and `impact_factor`.

4.2 Rule-Based Approach

Our rule based approach is more closely based on the premise of our hypothesis. Remember our hypothesis is that if two or more

Algorithm 1: CAER Shutter Burst Algorithm

Description: This main loop is executed throughout the lifetime of the host application. (pause_self is used to signal whether to pause execution for the next period)

```
$count \leftarrow 0$;
while application running do
    update $l\_window$ with $llc\_misses$;
    update $r\_window$ with $neighbors\_llc\_misses$;
    $count$++;
    $pause\_self \leftarrow true$;
    if count equals switch_point then
        foreach e in r_window until switch_point do
            $steady\_average \leftarrow steady\_average + (e/(\texttt{Size}(r\_window) - switch\_point)$
        end
        $pause\_self \leftarrow false$;
    end
    if (count > switch_point) and (count < end_point) then
        $pause\_self \leftarrow false$;
    end
    if count equals end_point then
        foreach e in r_window from switch_point to end_point do
            $burst\_average \leftarrow burst\_average + (e/(end\_point - switch\_point)$
        end
        if $((burst\_average - steady\_average) > noise\_thresh) and (burst\_average > (steady\_average * (1 + impact\_factor)))$ then
            $contending \leftarrow true$;
        end
        else
            $contending \leftarrow false$;
        end
    end
end
```

Algorithm 2: CAER Rule Based Algorithm

```
while application running do
    update $l\_window$ with $llc\_misses$;
    update $r\_window$ with $neighbors\_llc\_misses$;
    $contending \leftarrow true$;
    foreach e in l_window do
        $average \leftarrow average + (e/\texttt{Size}(l\_window))$
    end
    if average < usage_thresh then
        $contending \leftarrow false$
    end
    $average \leftarrow 0$;
    foreach e in r_window do
        $average \leftarrow average + (e/\texttt{Size}(r\_window))$
    end
    if average < usage_thresh then
        $contending \leftarrow false$
    end
end
```

applications are simultaneously missing heavily in the last level shared cache of the micro architecture, they are both making heavy usage of the cache and probably evicting each others data (i.e. contending). The rule based heuristic tries to test this directly. The basic intuition says, if the latency-sensitive application is not missing in the cache heavily, it is probably not suffering from cache contention, and also if the batch application is not missing heavily in the cache, it is probably not using or at least not contending in the cache very much. This heuristic works by maintaining a running average of the last level cache miss windows for both the latency-sensitive, and batch applications. When this average for either application dips below a particular threshold, we assert that we are not contending, otherwise we report contention. Algorithm 2 presents the corresponding algorithm. In this heuristic the parameters include the size of the window and defining what missing heavily means. In the algorithm these correspond to `window` and `usage_thresh`.

5. Responding to Contention with CAER

As Figure 5 shows, after detecting contention we transition into one of the response states, either *c-negative* or *c-positive*. In these states the CAER runtime environment can respond by dynamically modifying and adapting the batch application under which it runs. In this work CAER reacts to contention by enforcing a fine grained throttling of the execution of the batch application to relieve pressure in the shared cache.

Our CAER runtime environment currently employs two throttling based dynamic contention response mechanisms: a *red-light green-light* approach, and a *soft locking* approach. Our red-light green-light approach, as the name implies stops or allows execution for a fixed or adaptive number of periods, based on the outcome of our contention detection phase. The red-light part of this response technique correlates to the c-positive result, the green-light correlates to the c-negative result. An adaptive approach can be applied, increasing the length if the detection phase is consistently producing the same result. In our CAER runtime environment we use this *red-light green-light* response with our burst shutter approach.

Our soft locking response technique applies a *soft lock* on the shared last level cache until the cache is no longer being used heavily by the latency-sensitive application. The amount of pressure placed on the cache by the latency-sensitive application is measured using the same performance monitoring information used for the contention detection phase. The batch application is allowed to fully resume execution when the pressure on the cache subsides. In our CAER runtime environment we use this response technique with our rule based approach.

6. Evaluation

The goals of this work is to provide a contention aware execution runtime environment that can dynamically detect and respond to contention on today's commodity multicore processors. We aim to minimize the cross-core interference penalty (overhead of the latency-sensitive application due to contention) and maximize the utilization of the chip. We demonstrate the effectiveness of our CAER environment by showing a considerable reduction in this cross-core interference penalty when allowing co-location, while achieving a significant increase of chip utilization compared to disallowing co-location.

6.1 Experimental Design

Our CAER prototype supports two applications, one deemed latency-sensitive and the other a throughput-oriented batch application. We use the SPEC2006 benchmark (C/C++ only) and run all programs to completion using their reference inputs. We use the Intel Core i7 (Nehalem) 920 Quad Core architecture to perform our experimentation. This processor has three levels of cache, the first two private to each core, the third shared across all cores. The sizes of the L1 and L2 caches are 16kb and 256kb respectively. The L3 cache is 8mb and inclusive to the L1 and L2. The system used has 4gb of main memory, and runs Linux 2.6.29.

In the experiments shown here, the `lbm` benchmark served as our batch application and was co-located on a neighboring core. The main benchmark is assumed to be the latency-sensitive appli-

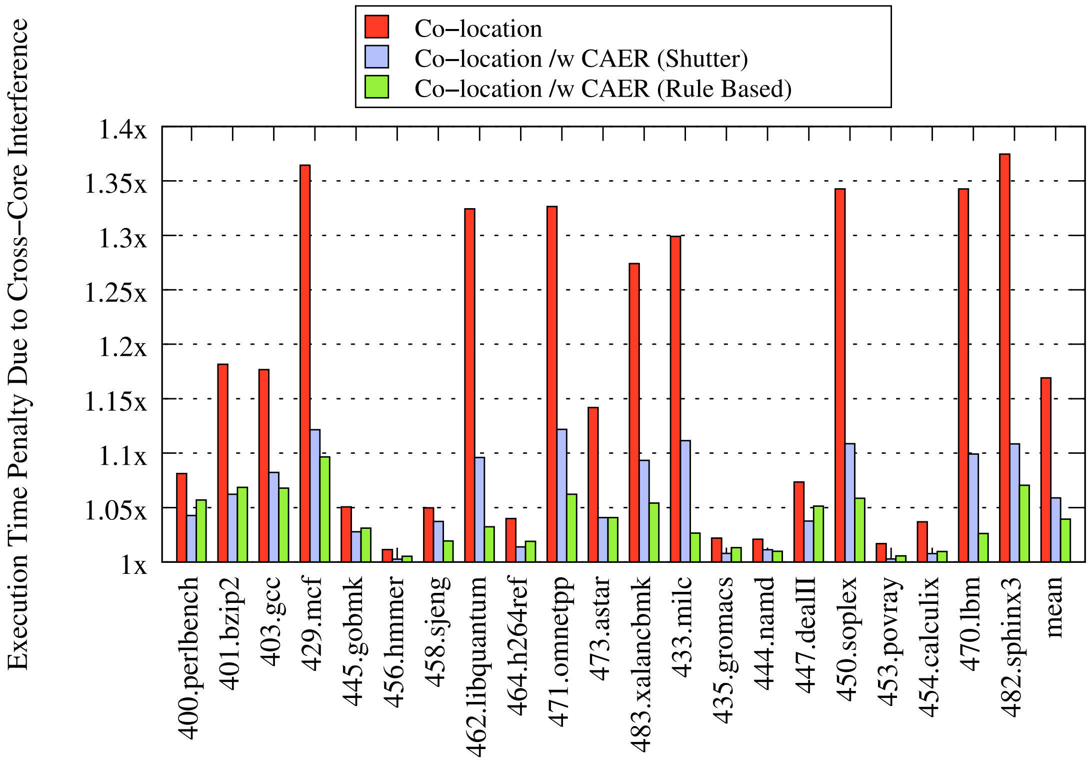


Figure 6. Investigating the reduction in cross-core interference penalty.

cation. `Lbm` was chosen as our batch application because it presents an interesting adversary as it makes heavy usage of the L3 cache. We have performed complete runs using other benchmarks such as `libquantum` and `milc` and produced very similar results. Note that adversaries that make light usage of the L3 cache present more trivial scenarios; contention occurs when two or more applications are making heavy usage of the last level cache. As presented shortly, our experimentation covers cases where the latency-sensitive application make both light and heavy usage of the shared cache.

We have scripted our SPEC runs to launch the latency-sensitive application shortly after the batch is launched. As our applications run, CAER logs the decisions it makes and wall clock execution time of our latency-sensitive application running on CAER M. In the few cases the `lbm` (batch) benchmark completes before the latency-sensitive we automatically and immediately relaunch it and aggregate logs.

6.2 Minimizing Contention and Maximizing Utilization

First we evaluate the reduction in cross-core interference penalty due to contention when running on our CAER environment. In Figure 6 we show the slowdown in execution time due to contention when we co-locate the latency-sensitive and batch applications. The first bars show the cross-core interference penalty when co-locating the native applications directly on multicore chip. The second bars shows the cross-core interference penalty when co-locating the native applications on CAER with the *burst shutter* heuristic. The last bars show this co-location on CAER with the *rule based* approach.

As Figure 6 shows we significantly reduce the cross-core interference penalty for the wide range of SPEC2006 benchmarks. Our *burst shutter* contention detection technique uses the *red-light green-light* response with a response length of 10 periods. The `impact threshold` in Algorithm 1 for the burst shutter detection is set to 5%, meaning if the batch application burst causes a spike of 5% or more in last level cache misses of the latency-sensitive application we assert contention. Using this approach CAER brings the overhead due to contention from 17% down to 6% on average, while gaining close to 60% more utilization of the processor over running the latency-sensitive application alone, which can be seen in Figure 7.

Our *rule based* contention detection technique uses the *soft locking* response and the `usage threshold` found in Algorithm 2 is set to `1500`, meaning we have to see an average of 1500 or more last level cache misses per period (1 ms) to assert heavy usage of the cache. Using this approach CAER brings the overhead due to contention from 17% down to 4% on average, while gaining 58% more utilization of the processor over running the latency-sensitive application alone, as show in Figure 7.

Our *rule based* CAER contention detection approach slightly outperforms our *shutter based* approach on average. However the *shutter based* approach has some desirable characteristics. The *burst shutter* approach is highly tunable to the QoS requirements of the application. The impact threshold determines how much cross-core interference the latency application is willing to withstand; this provides a “knob” which intuitively sets the sensitivity of detection. Here we use “sensitivity” to mean the amount of impact needed to trigger a *c-positive* response. Although the *rule based* approach is also tunable as to how conservative or liberal the definition of “heavy usage” of the cache is, it provides a less intuitive abstraction. As the goal of this evaluation is to demonstrate the ef-

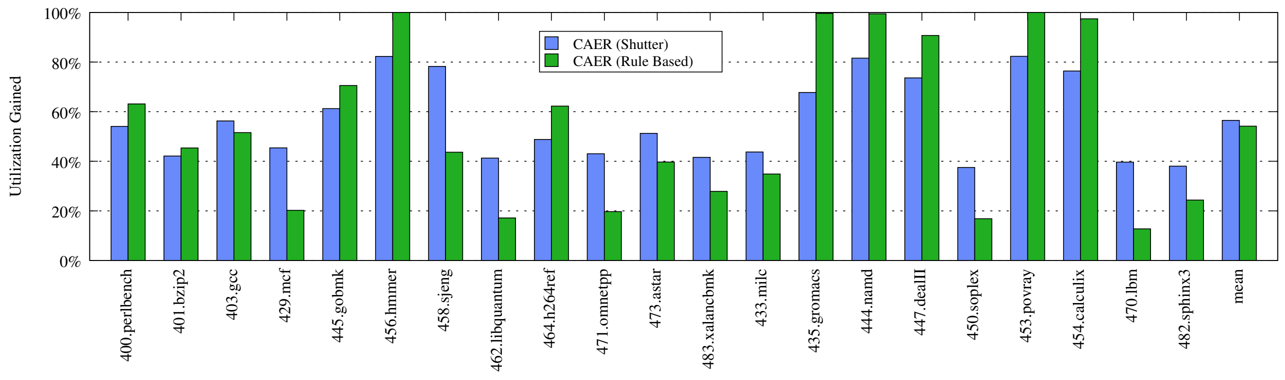


Figure 7. Maximizing Utilization (Higher is Better)

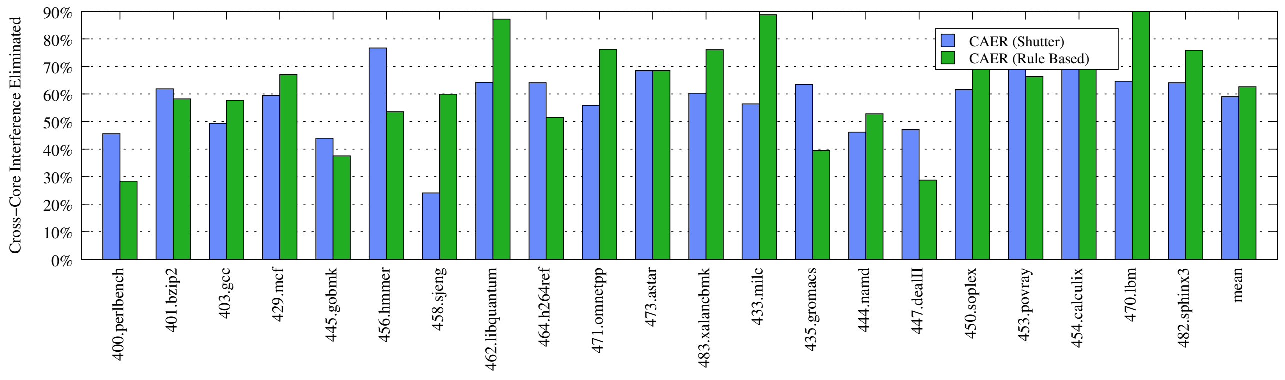


Figure 8. Minimizing Cross-Core Interference (Slowdown Eliminated, Higher is Better)

fectiveness of our CAER runtime environment and its applicability to current multicore architecture, we reserve further investigation of the heuristic tuning space for future work.

Figures 7 and 8 further illustrate CAER's effectiveness. As mentioned before, Figure 7 shows the utilization gained on the multicore processor when co-locating the latency-sensitive and batch applications using CAER. Figure 8 is another way to represent the decrease in cross-core interference penalty shown in Figure 6, showing the percentage of the cross-core interference penalty eliminated. For both of these Figures, higher is better. Running the latency-sensitive application alone will provide 100% cross-core interference elimination but 0% utilization gained. Running the applications together will provide 0% cross-core interference elimination but will have 100% utilization gained. Our goal is to maximize both while running both application on our CAER framework. It is important to note that utilization gained and cross-core interference eliminated are two separate units of measurement, so 50% cross-core interference eliminated for 50% more utilization can be a great result depending on the *cross-core interference sensitivity* of the latency-sensitive application. We explore *cross-core interference sensitivity* in the following section.

6.3 Understanding and Adapting to Cross-Core Interference Sensitivity

The amount of performance impact an application can experience due to contention for shared resources differs from application to application. We call this application characteristic its *cross-core interference sensitivity.* This characteristic can also be determined by the amount of reliance an application puts on a shared resource. Applications whose working set fits in its core-specific private caches are *cross-core interference insensitive*. Applications whose working set uses shared cache, memory, etc, are *cross-core interference sensitive.*

When performing contention detection and response the handling of cross-core interference insensitive and cross-core interference sensitive applications should be different. More concretely, the amount of utilization that is sacrificed to reduce contention of a cross-core interference sensitive application should be higher than the cross-core interference insensitive application. For example, an application a is 50% slower when experiencing contention x, while another application b is 4% slower when contending with x. We say application a is more cross-core interference sensitive than b. To eliminate half of the cross-core interference penalty of a is more valuable than b, meaning the benefit gained, a 20% increase in speed, with a is better than the 1.9% speed up in b. Thus, we should be willing to sacrifice more utilization to eliminate 50% of the cross-core interference penalty for a than b since a is more cross-core interference sensitive.

Lets take `mcf` as application a and `namd` as b. As shown previously in Figure 6 `mcf` suffers a 36% slowdown when contending with `lbm`, `namd` only suffers a 2% performance degradation. Clearly `mcf` is more latency-sensitive than `namd`, therefor a good contention detection and response approach will be able to detect these different cross-core interference sensitivities and sacrifice more utilization for the former case. CAER does exactly this. For `mcf` CAER *burst shutter* approach sacrifices 36% more utilization to accommodate `mcf`'s cross-core interference penalty, and CAER *rule based* sacrifices 80% more utilization.

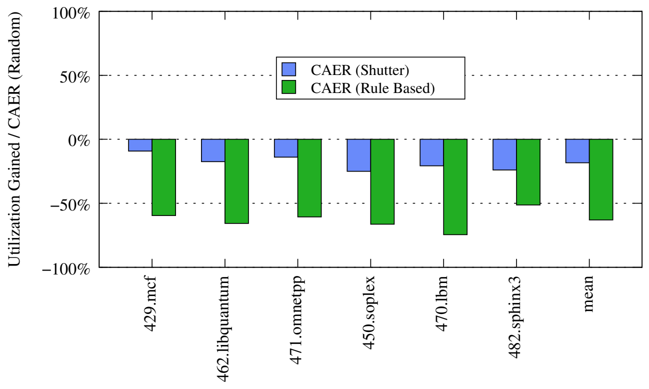


Figure 9. Utilization gained relative to random for 6 most cross-core interference sensitive applications.

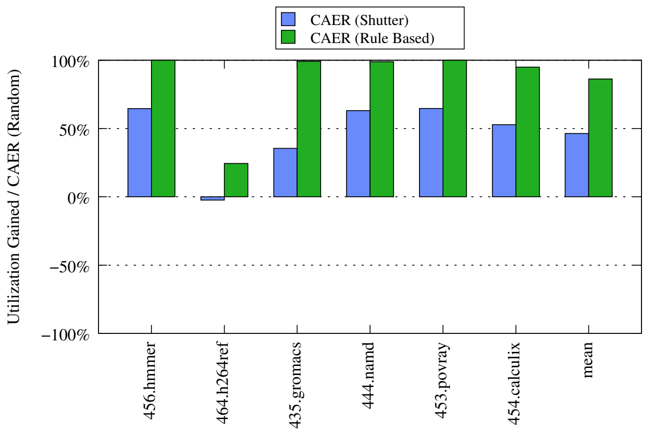


Figure 10. Utilization gained relative to random for 6 least cross-core interference sensitive applications.

6.4 Contention Detection Accuracy

When detecting contention it is possible to have both false positives and false negatives. A false positive occurs when contention is detected where there is none. A false negative occurs when no contention is detected where there is contention. To evaluate a heuristic's ability to accurately detect contention we have developed a baseline random heuristic. This heuristic reports contention with probability P and no contention with probability $1 - P$. In our experiments P equals 0.5. To respond to contention this heuristic uses the *red-light green-light* with a length of 1 period. To illustrate a CAER heuristic's ability to detect contention accurately we use the following

$$A = \frac{U_h}{U_r} - 1 \tag{2}$$

where U_h is the utilization gained from a heuristic h, and U_r is the utilization gain with the random heuristic. Figures 9 and 10 demonstrates the contention detection accuracy of the *burst shutter* and *rule based* heuristics for the six most, and six least cross-core interference sensitive benchmarks respectively. The y-axis corresponds with the calculation of A from the equation. Figure 9 shows that, for cross-core interference sensitive benchmarks, our CAER heuristics sacrifices more utilization than the random technique, indicating that our detection is correctly responding to these applications as high contenders (i.e. cross-core interference sensitive). Figure 10 shows the opposite for cross-core interference insensitive benchmarks. The heuristics gain much more utilization than the random heuristics, indicating we are correctly responding to these workloads as low contenders.

Also note that any inversion in this response to cross-core interference penalty indicates inaccurate contention detection. Gaining more utilization for a cross-core interference sensitive application than the random heuristic represents a false negative (asserting no contention where there is contention). And contrarily, gaining less utilization for cross-core interference insensitive applications represents a false positive (asserting contention where there is none).

7. Related Work

QoS and Fairness techniques have received much research attention [11, 14, 15, 17, 21–23, 26]. These works propose QoS and fairness models, as well as hardware and platform improvement to enable QoS and fairness be enforced. Rafique et al. investigates micro-architectural extensions to support the OS for cache management [24]. There has been a number of works aimed at better understanding and modeling cache contention [2, 3, 8] and job co-scheduling [6, 10, 16]. Other hardware techniques to enable cache management have also received research attention [5, 12, 25, 27]. Suhendra [27] proposes partitioning and locking mechanisms to minimize unpredictable cache contention. Cache reconfiguration [25] has also been proposed as a mechanism to enable cache partitioning. Although these works show promising future directions for hardware and system designers to take when addressing these problems, unfortunately current commodity micro-architectures cannot support these solutions as they do not meet the micro-architectural assumptions made these works.

Another very promising direction based on what is likely to be future hardware capabilities, is to leverage core specific dynamic voltage scaling as is presented by Herdirch, Illikkal, Iyer, et al [11]. Instead of throttling down the execution of an application, this work proposes throttling down the frequency of the core hosting the batch application. This approach also has the added benefit of being energy efficient.

Hardware performance monitoring capabilities have been used heavily for online and adaptive solutions. Azimi et al. used these capabilities to enhance operating system support for multicore systems [1]. Mars et al. leverage performance monitoring hardware to enable online application adaptation via multiversioning [20].

8. Conclusion

Cross core cross-core interference on current multicore processors pose a significant challenge to providing application level quality of service (QoS) guarantees. This problem is especially prevalent with latency-sensitive applications (such as web-search) in the web services and data center domains. The commonly used solution to this problem is to disallow the co-location of latency-sensitive and batch applications on a single chip. This approach leaves much of the processing capabilities in multicore systems underutilized, and is a contributing factor to the low utilization in today's data centers (typically 15% or less [18]). In this work we have presented the **Contention Aware Execution Runtime (CAER)** environment, the first of its kind to our knowledge. The goals of CAER is to minimize the cross-core interference penalty of application co-location on multicore processors, while maximizing the utilization on the processor. In addition, CAER can be applied on today's commodity hardware.

By allowing co-location with CAER, as opposed to disallowing co-location, we are able to increase the utilization of the multicore

CPU by 58% on average. Meanwhile CAER brings the overhead due to allowing co-location from 17% down to just 4% on average. CAER can be used in today's data centers and serve to increase utilization, resulting in savings in energy and cost.

References

[1] R. Azimi, D. K. Tam, L. Soares, and M. Stumm. Enhancing operating system support for multicore processors by using hardware performance monitoring. *SIGOPS Oper. Syst. Rev.*, 43(2):56–65, 2009.

[2] G. E. Blelloch and P. B. Gibbons. Effectively sharing a cache among threads. In *SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 235–244, New York, NY, USA, 2004. ACM.

[3] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 340–351, Washington, DC, USA, 2005. IEEE Computer Society.

[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):1–26, 2008.

[5] J. Chang and G. S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*, pages 242–252, New York, NY, USA, 2007. ACM.

[6] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson. Scheduling threads for constructive cache sharing on cmps. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 105–115, New York, NY, USA, 2007. ACM.

[7] L. Cherkasova, Y. Fu, W. Tang, and A. Vahdat. Measuring and characterizing end-to-end internet service performance. *ACM Trans. Internet Technol.*, 3(4):347–391, 2003.

[8] C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 245–257, New York, NY, USA, 2003. ACM.

[9] S. Eranian. Perfmon2. http://perfmon2.sourceforge.net/.

[10] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum. Performance of multithreaded chip multiprocessors and implications for operating system design. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 26–26, Berkeley, CA, USA, 2005. USENIX Association.

[11] A. Herdrich, R. Illikkal, R. Iyer, D. Newell, V. Chadha, and J. Moses. Rate-based qos techniques for cache/memory in cmp platforms. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, pages 479–488, New York, NY, USA, 2009. ACM.

[12] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler. A nuca substrate for flexible cmp cache sharing. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 31–40, New York, NY, USA, 2005. ACM.

[13] Intel Corporation. *IA-32 Application Developer's Architecture Guide*. Intel Corporation, Santa Clara, CA, USA, 2009.

[14] R. Iyer. Cqos: a framework for enabling qos in shared caches of cmp platforms. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pages 257–266, New York, NY, USA, 2004. ACM.

[15] 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. In *SIGMETRICS '07: Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 25–36, New York, NY, USA, 2007. ACM.

[16] Y. Jiang, X. Shen, J. Chen, and R. Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 220–229, New York, NY, USA, 2008. ACM.

[17] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 111–122, Washington, DC, USA, 2004. IEEE Computer Society.

[18] S. Lohr. Demand for data puts engineers in spotlight. *The New York Times*, 2008. Published June 17th.

[19] K. London, J. Dongarra, S. Moore, P. Mucci, K. Seymour, and T. Spencer. End-user tools for application performance analysis using hardware counters. In *14th Conference on Parallel and Distributed Computing Systems*, August 2001.

[20] J. Mars and R. Hundt. Scenario based optimization: A framework for statically enabling online optimizations. In *CGO '09: Proceedings of the 2009 International Symposium on Code Generation and Optimization*, pages 169–179, Washington, DC, USA, 2009. IEEE Computer Society.

[21] M. Moreto, F. J. Cazorla, A. Ramirez, R. Sakellariou, and M. Valero. Flexdcp: a qos framework for cmp architectures. *SIGOPS Oper. Syst. Rev.*, 43(2):86–96, 2009.

[22] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair queuing memory systems. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 208–222, Washington, DC, USA, 2006. IEEE Computer Society.

[23] K. J. Nesbit, J. Laudon, and J. E. Smith. Virtual private caches. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 57–68, New York, NY, USA, 2007. ACM.

[24] N. Rafique, W.-T. Lim, and M. Thottethodi. Architectural support for operating system-driven cmp cache management. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 2–12, New York, NY, USA, 2006. ACM.

[25] R. Reddy and P. Petrov. Eliminating inter-process cache interference through cache reconfigurability for real-time and low-power embedded multi-tasking systems. In *CASES '07: Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 198–207, New York, NY, USA, 2007. ACM.

[26] L. Soares, D. Tam, and M. Stumm. Reducing the harmful effects of last-level cache polluters with an os-level, software-only pollute buffer. In *MICRO '08: Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 258–269, Washington, DC, USA, 2008. IEEE Computer Society.

[27] V. Suhendra and T. Mitra. Exploring locking & partitioning for predictable shared caches on multi-cores. In *DAC '08: Proceedings of the 45th annual Design Automation Conference*, pages 300–303, New York, NY, USA, 2008. ACM.