Directly Characterizing Cross Core Interference Through Contention Synthesis

Jason Mars
Dept. of Computer Science
University of Virginia
jom5x@cs.virginia.edu

Lingjia Tang
Dept. of Computer Science
University of Virginia
lt8f@cs.virginia.edu

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

ABSTRACT

In this paper, we present a *direct* methodology and framework for the measurement and characterization of an application's *cross-core interference sensitivity* on multicore microarchitectures. While prior works use *indirect* indicators, such as last level cache miss rate, to infer an application's cross-core interference sensitivity, our approach is *direct*, in that it characterizes the application's cross-core interference sensitivity using the performance impact due to actual contention. Our methodology and framework, the **C**ross-core **i**nterference **P**rofiling **E**nvironment, or **CiPE**, is composed of a lightweight runtime environment on which a host application runs, along with a carefully designed *contention synthesis* engine that executes on a neighboring core. CiPE manipulates the co-running contention synthesis engine, while monitoring and analyzing the resulting dynamic impact on the host application.

CiPE is able to characterize the cross-core interference sensitivity of the entire application, its individual phases, or source level code regions. To demonstrate the effectiveness of CiPE, we use CiPE characterizations to address two pressing problems. First, we use CiPE characterizations to perform *contention conscious* batch scheduling that minimizes cross-core interference, resulting in a 12% performance improvment on average when applied to the SPEC2006 benchmark suite, and beyond 20% in the case of `mcf` and `omnetpp`. Second, we use CiPE to design a performance analysis tool that is capable identifying contentious bottlenecks in application code.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*parallel programming*; D.3.4 [**Programming Languages**]: Processors—*run-time environments, compilers, optimization, debuggers*; D.4.1 [**Operating Systems**]: Process Management—*scheduling*; D.4.8 [**Operating Systems**]: Performance—*measurements, monitors*

General Terms

Performance, Measurement, Algorithms, Experimentation

Keywords

cross-core interference, profiling framework, program understanding, contention, multicore, dynamic analysis, execution runtimes

1. INTRODUCTION

As multicore architectures become ever more prolific, techniques to characterize and understand application behavior when executing along with co-running applications on neighboring cores remain few. One of the most critical program characteristics in this domain is that of cross-core interference. An application suffers *cross-core interference* when its performance is negatively impacted by contending for shared resources with another application process or thread concurrently executing on a neighboring core.

While prior work has demonstrated the importance of characterizing an application's cross-core interference sensitivity, current best known techniques use *indirect* methods [27, 13, 2, 29, 6]. An indirect analysis is one that infers an application's cross core interference sensitivity. An example of an indirect analysis is the usage of an application's last level cache missrate to predict its cross-core interference sensitivity [13]. A *direct* analysis on the other hand, is one that characterizes the impact on application performance when contention occurs in comparison to when no contention is present. In this paper, we present the **C**ross-core **i**nterference **P**rofiling **E**nvironment, **CiPE**, the first *direct* methodology and framework for the characterization of an application's sensitivity to cross-core performance interference.

The key insight and observation motivating the design of our cross-core profiling methodology is the fact that contention for shared cache and memory resources is an intrinsically dynamic property of the application's memory behavior, coupled with the intricacies of the particular microarchitectural and memory subsystem design. Therefore, we employ a direct, empirical, online characterization approach.

As an application executes on our CiPE environment, a carefully designed *contention synthesis engine* is spawned on a neighboring core to run alongside the application. This contention synthesis engine is dynamically manipulated by CiPE, and the resulting impact on the host application is analyzed.

To demonstrate the effectiveness of CiPE, we apply CiPE to two real problems on current state-of-the-art general purpose multicore architectures. These problems include scheduling to minimize cross-core interference, and identifying contentious bottlenecks in application source code.

The understanding of an application's cross-core interference sensitivity enables *contention conscious* application co-scheduling. Applications that place a higher demand on shared memory resources can be co-located with applications that place a lower demand on shared memory to gain better overall performance and throughput. In this work we show that contention conscious scheduling significantly reduces cross-core interference and improves application throughput. Using our approach, we were able to improve the performance of the contention sensitive SPEC2006 benchmarks by 12% on average and up to 24% in the case of `mcf`.

In addition to contention conscious scheduling, we use our cross-core interference sensitivity profiling information to design a performance analysis and debugging tool that identifies contentious code regions in application code. To the best of our knowledge, this is the first performance analysis and debugging tool capable of identifying contentious code regions by measuring actual contention. To accomplish this profiling technique, we take phase level CiPE profiles with the highest contention sensitivity rating and correlate this sensitivity back to source code. As a result we are able to identify sources of contention sensitivity in the original application source code. The ability to detect the most contention sensitive application phases and regions of source code allows the user to determine application bottlenecks.

The contributions of this work are:

- A general *direct* measurement technique and metric for quantifying *cross-core interference sensitivity*, and the design of CiPE, a framework that employs this measurement technique along with *contention synthesis* to characterize application cross-core interference sensitivity.

- The design of a contention synthesis engine, including a thorough evaluation of multiple contention synthesis methods and investigation of whether various forms of contentious behavior produce differing characterizations.

- The design and implementation of both: 1) a *contention conscious* scheduling approach using the CiPE characterization methodology and 2) a performance analysis approach that identifies regions of code that exhibit high sensitivity to cross-core interference on current microarchitectures.

Next, in Section 2 we demonstrate the problem of cross-core interference. We then present an overview of our framework in Section 3. Section 4 discusses our cross-core interference characterization methodology. In Section 5, we discuss the design of our contention synthesis engine. We evaluate our methodology and framework in Section 6, discuss related work in Section 7 and finally conclude in Section 8.

2. CROSS-CORE INTERFERENCE

The memory subsystem on current commodity multicore architectures is shared among multiple processing cores. Two

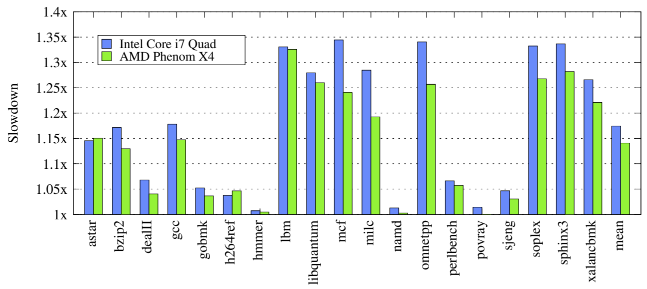

Figure 1: Performance impact due to contention from co-location with LBM.

representative examples of the state of the art multicore chip designs are the Intel Core i7 Quad Core chip and AMD's Phenom X4 Quad Core. Intel's Core i7 has four processing cores sharing a large 8mb L3 cache. AMD's Phenom X4 also has four cores and shares a 6mb L3 cache. These chips were designed to accommodate 4 simultaneous streams of execution. However, as we can see through experimentation, their shared caches and memory subsystem often cannot efficiently accommodate even 2 co-running processes.

Figure 1 illustrates the potential *cross-core interference* that can occur when multiple co-running applications are executing on the Core i7 and Phenom X4 architectures described above. In this experiment we study the cross-core performance interference suffered by each of the SPEC2006 benchmarks when co-running with `lbm`, one of the SPEC2006 benchmarks known be an especially heavy user of the on-chip memory subsystem. Figure 1 shows the slowdown of each benchmark due to the cross-core interference from `lbm`. Each application was executed to completion on their `ref` inputs. On the y-axis we show the execution time of the application while co-running with `lbm` normalized to the execution-time of the application running alone on the system. The first bar in Figure 1 presents this data for the Core i7 architecture and the second bar for the Phenom X4. As this graph shows, there are severe performance degradations due to cross-core interference on a large number of Spec benchmarks. The large last level on-chip caches of these two architectures do little to accommodate many of these co-running applications. On a number of benchmarks including `lbm`, `mcf`, `omnetpp`, and `sphinx`, this degradation approaches 35%.

In addition to the general performance degradation, this *sensitivity* to cross-core interference is particularly undesirable for real time and latency sensitive application domains. In the latency sensitive domain of web search, for instance, cross core interference can cause unexpected slowdowns, negatively impacting the QoS on a search query. A commonly used solution in industry is to simply disallow the co-location of latency sensitive applications with others on a single machine, resulting is lowered utilization and higher energy cost [14].

Note that not all applications are effected by the contention properties of their co-runners. Applications such as `hmmer`, `namd`, and `povray` appear to be immune to `lbm`'s cross core interference, demonstrating that cross-core interference sensitivity varies substantially across applications.

It is clear from Figures 1 that knowledge of an application's sensitivity to cross-core performance interference is

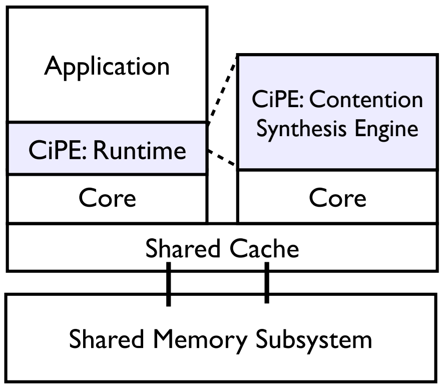

Figure 2: The CiPE Framework

critical to understanding the dynamic interaction and the resulting performance implications of co-running applications on current commodity multicore architecture. In this work, we aim to understand this sensitivity at three levels: entire applications, their individual phases, and their source level code regions.

3. OVERVIEW

In this section, we describe CiPE and present an overview of our approach.

3.1 Description and Insight

CiPE is a profiling analysis approach capable of characterizing an applications *cross-core interference sensitivity.* To perform this characterization, an application must be run only once on our CiPE framework. This characterization produces application-level and phase-level information that can be subsequently used for a range of purposes such as contention-conscious scheduling, performance analysis and debugging, and a host of other uses.

It is important to remember, however, that since the design of the underlying architecture and memory subsystem determine the potential for cross-core interference, each CiPE profile represents the application's cross-core interference sensitivity on the underlying architecture on which the profile was collected. For example a multicore chip with core-private L1 caches big enough to contain the working set of `lbm` could run multiple instances of `lbm` with no cross-core interference. On this architecture `lbm` is not *sensitive* to *cross-core interference.* This is not the case on other chips whose core-private caches cannot accommodate `lbm`, such as the Core i7 or Phenom X4. This insight about the nature of cross-core interference further motivates having a *direct* approach like CiPE. While the profiles produced by CiPE are representative of a particular underlying architecture and those that are similar, the characterization methodology, and CiPE itself, is portable from chip generation to chip generation.

3.2 Profiling Environment

Figure 2 provides an overview of **CiPE** running on a multicore architecture with two separate cores sharing an on-chip cache and memory subsystem. The shaded boxes represent our CiPE profiling framework, which is composed of the CiPE runtime and a *contention synthesis engine* (CSE). As shown on the left side of Figure 2, the host application is monitored throughout its execution by the CiPE runtime. Before the execution of the host application, the CiPE runtime spawns the CSE on a neighboring core, as shown to the right of the figure. This CSE shares the cache and memory subsystem of the host application. As the application executes, the CSE aggressively accesses memory causing as much cross-core interference as possible. The CiPE runtime manipulates the execution of the CSE allowing bursts of execution to occur by turning the CSE on and off. Slowdowns in the application's instruction retirement rate that result from this bursty execution are monitored using the *hardware performance monitoring* (HPM) information [8] and are used to characterize its sensitivity. This intermittent control of the CSE and monitoring of the HPM are achieved using a *periodic probing* approach [15]. A timer interrupt is used to periodically execute the monitoring and profiling directives. Periodic probing has shown to be a very low overhead approach for the dynamic monitoring and analysis of applications. In the next section, we present the analysis used to measure the *cross-core interference sensitivity*, and Section 5 presents the design of the contention synthesis engine.

Algorithm 1: CiPE Core Algorithm

```
Description: This main loop is executed throughout the
        lifetime of the host application.
Initialize_CSE();
CSE_Off();
$CSE\_Satus \leftarrow dormant$;
while application running do
    Let_App_Run(probe_time);
    if CSE_Status equals active then
        $CSE\_active\_ir \leftarrow$ Read_PMU(instructions_retired);
        Characterize_CIS(CSE_dormant_ir, CSE_active_ir);
        Record_Profile();
        $CSE\_Satus \leftarrow dormant$;
        CSE_Off();
    end
    else if CSE_Status equals dormant then
        $CSE\_dormant\_ir \leftarrow$ Read_PMU(instructions_retired);
        $CSE\_Satus \leftarrow active$;
        CSE_On();
    end
end
```

The core algorithm of our CiPE runtime is presented in Algorithm 1. In this algorithm, *CSE_Status* (line 3) is a flag used to denote whether the CSE engine is executing (active) or sleeping (dormant). *CSE_active_ir* (line 7) and *CSE_dormant_ir* (line 14) records the value of the *instructions_retired* performance counter available in most current microarchitectures. The periodic probing interval is set with *probe_time*. `CSEOn()` and `CSEOff()` (lines 11 and 16) turn the contention synthesis engine on and off (described in Section 5). `Characterize_CIS()` calculates a cross-core interference sensitivity score (described in Section 4). Our CiPE Algorithm runs continuously for the duration of the host application's execution.

As the algorithm shows, the performance monitors are read at every *probe_time* interval. A sample of the *instructions_retired* is collected when the CSE is *active*, and another is collected when the CSE is *dormant.* Both samples are passed as input to the cross-core sensitivity characterization routine `Characterize_CIS()`, and finally recorded by `Record_Profile()`.

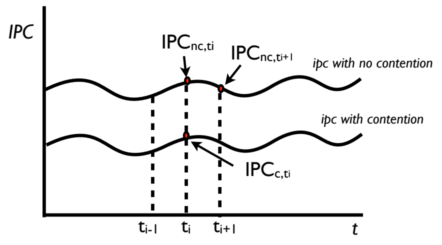

Figure 3: IPC Curves

4. QUANTIFYING SENSITIVITY

In this section, we present the metrics and measurements used to quantify *cross-core interference sensitivity* (CIS). This analytical model is used for the `Characterize_CIS()` function presented in Algorithm 1.

4.1 Defining Sensitivity (CIS)

We define an application's *cross-core interference sensitivity* in terms of performance degradation. Our *direct* metric to characterize an application's cross-core interference sensitivity is the normalized difference between an application's IPC (instruction per cycle) in the presence of contention and its IPC when it is running alone. CIS can be studied at a fine granularity continuously during an application's execution. The resulting CIS report would reveal dynamic phases of the application's sensitivity characteristics. We use the following formula to define CIS at any time point t_i during an execution:

$$CIS_{t_i} = \frac{IPC_{no\_c,t_i} - IPC_{c,t_i}}{IPC_{no\_c,t_i}} \tag{1}$$

where IPC_{no_c,t_i} is the application's IPC with no contention at time t_i, and IPC_{c,t_i} is the application's IPC in the presence of contention at time t_i. The intuition of the formula is shown in Figure 3. Here we present an illustrative diagram of an application's two IPC curves of its two executions, with and without contention respectively. At time t_i, CIS is represented by the distance between two points at t_i on two IPC curves, IPC_{no_c,t_i} and IPC_{c,t_i}, then normalized by the IPC with no contention IPC_{no_c,t_i}. Compared to using other metrics such as cache misses to indirectly infer the application's sensitivity, CIS directly measures sensitivity using the percentage of and application's performance (IPC) that is lost due to cross-core interference. Notice that executions with and without contention may vary in time to finish so one IPC curve may need to be normalized for the sake of the calculation.

CIS can also be studied at an application level to characterize the application's general intrinsic sensitivity to cross-core interference. This can be viewed as the average distance between two curves, and can be graphically interpreted as the area between the two curves normalized to the area under the curve for IPC with no contention as shown in Figure 3 and the formula:

$$\begin{aligned} CIS_{avg} &= \frac{\int_{t_s}^{t_e} IPC_{no\_c} - \int_{t_s}^{t_e} IPC_c}{\int_{t_s}^{t_e} IPC_{no\_c} \times (t_e - t_s)} \quad (2) \\ &= (IPC_{no\_c} - IPC_c)_{avg} \quad (3) \end{aligned}$$

where t_s and t_e is the start and end point of the time period we are characterizing. This can also be calculated simply as the average CIS throughout the execution, as shown in the formula.

4.2 CIS Sampling Methodology

Given the above CIS definition, our CiPE system provides a novel approach to calculating, monitoring and profiling CIS through execution. The CiPE system invokes the contention synthesis engine to inject cross-core interference at regular frequent intervals, which facilitates direct measurement and profiling of contention's impact on an application's performance. To measure the difference between IPCs with and without contention, CiPE system uses an bursty sampling methodology. CiPE system first turns off the synthesis engine for a sample interval, collects the host application's IPC without contention, then in the immediately-following sample interval, turns on the synthesis engine to collect the application's IPC under the synthesized contention. CiPE uses these two adjacent samples to approximate Formula 1 by approximating IPC_{no_c,t_i} using $IPC_{no\_c,t_{i+1}}$, as shown in the following formula:

$$CIS_{t_i} \approx \frac{IPC_{no\_c,t_{i+1}} - IPC_{c,t_i}}{IPC_{no\_c,t_{i+1}}} \tag{4}$$

where $t_{i+1} - t_i$ is the sample interval. Notice that the interval from t_{i-1} to t_i is when CiPE has the synthesis engine on to generate contention, and at t_i, IPC_{c,t_i} is collected. From t_i to t_{i+1} the synthesis engine is off and $IPC_{no\_c,t_{i+1}}$ is sampled at t_{i+1}. We call the characterization using this formula, the CIS score.

The above formula is used to generate CIS score at every sample interval along the application's execution. Also, we can define the application's average CIS score using the following formula to calculate the average of all CIS scores during the execution to approximate Formula 2:

$$CIS_{avg} \approx \frac{\sum_{i=0}^{n} CIS_{t_i}}{n} \tag{5}$$

where n is the number of samples.

5. CONTENTION SYNTHESIS

In this section, we discuss the challenge and task of synthesizing contention.

5.1 Challenge of Contention Synthesis

The type of data access pattern and the way that data is mapped into the cache is very important to consider when constructing the CSE. Structures such as hardware cache prefetchers and victim caches can avert poor and contentious cache behavior even when the working set of the application is very large. The features and functionality of these hardware techniques are difficult to anticipate as vendors keep these details closely guarded.

With these advances in microarchitectural design, simply accessing a large amount of data does not necessarily cause high pressure on cache and memory performance. For example, access patterns that exhibit a large amount of spatial or temporal locality can easily be prefetched into the earlier and later levels of cache, and prefetch buffers can be used. An important question that arises is, on sophisticated modern architectures, whether an application's sensitivity to contention depends on the manner in which the contention is synthesized.

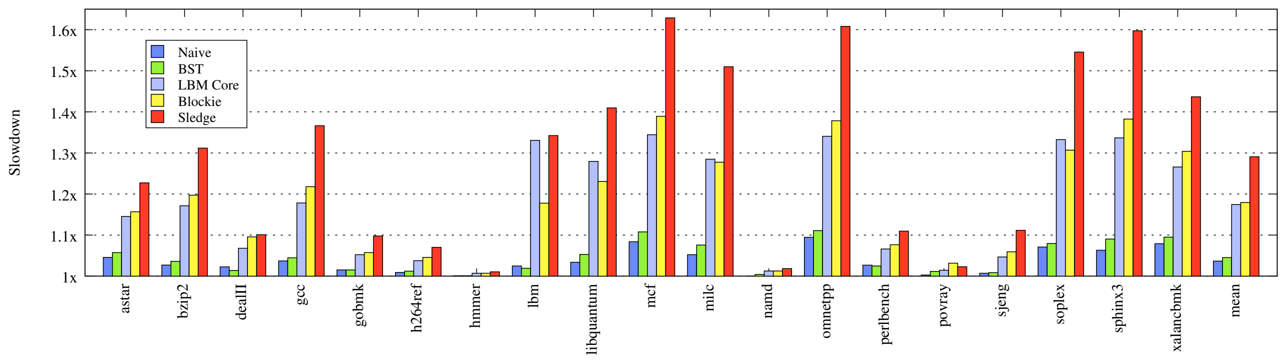

Figure 4: Slowdown caused by contention synthesis on Intel Core i7.

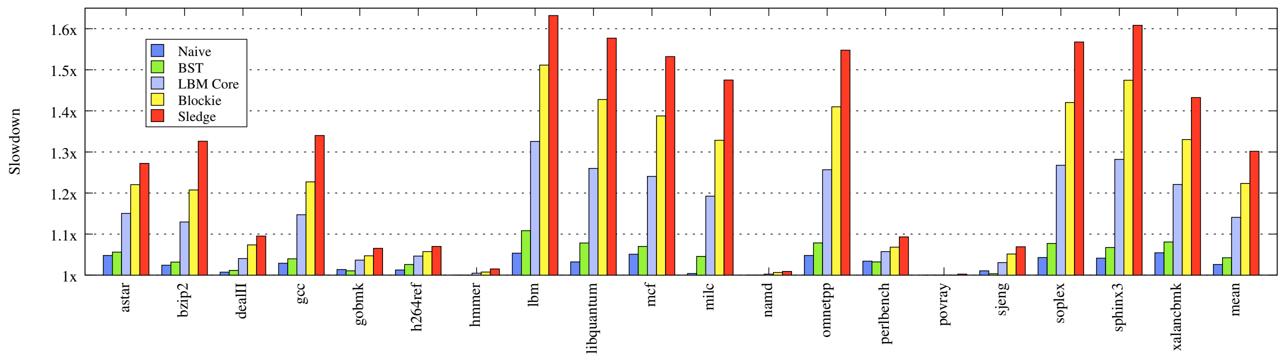

Figure 5: Slowdown caused by contention synthesis on AMD Phenom X4.

5.2 Designing Contention Synthesis

To address this question, we constructed a four diverse execution kernels that mimic common access patterns, and investigated the contentious impact of these varying workloads. The first of these designs, `Naive`, simply accesses a large array of memory (larger than the last level cache) performing both loads and store at random memory locations.

The second design for the CSE, `BST`, consists of the random construction and traversal of a binary search tree. Each tree node consisted of an `id` and a `payload`, and a custom traversal function is used. The payload consisted of a number of random bytes (128 in our design) to have the node map into its own cache line. The contentious kernel of this approach consisted of a specialized traversal function that performs a random depth first search through the tree touching and changing the data alone the way.

The third design, `Blockie`, is a 3D data movement microbenchmark consisting of a number of large 3D arrays of `double` precision values that represent solid virtual cubes. The contentious kernel of this CSE is the transposition of cells of each cube into the space of another cube. The cells of one cube are continuously copied to another.

The forth design, `Sledge`, was designed by reverse engineering and investigating `lbm` to learn its contentious core nature. We call this design "The Sledgehammer." This name is motivated by the fact that the behavior of this design resembles touching an element in a multidimensional array, and modifying a number of sparsely surrounding elements. The final version of this CSE first allocates two large arrays and enters its contentious kernel which copies data back and forth with this sledgehammer pattern.

More details and the complete algorithms for these four kernels can be found in our prior work [16].

5.3 Evaluating Contention Synthesis Designs

5.3.1 *Goals of Experiment*

We seek to answer two questions with our evaluation of contention synthesis designs. The first is whether there is a drastic difference between the interactions of different applications to the different contention synthesis designs. We hypothesize that contention is agnostic to the nature of memory access. We seek to evaluate this very question. The other goal of this evaluation is to learn whether there exists a synthesis engine that consistently generates more contention than all others, and if so, identify it.

5.3.2 *Experiments with 4 Designs*

Figures 4 and 5 show the performance impact of co-running each of the contention synthesis designs with each of the SPEC2006 benchmarks (C/C++ only), run to completion on `ref` inputs. Figure 4 shows the results when performing this co-location on Intel's Core i7 Quad architecture, and Figure 5 shows these results on AMD's Phenom X4 Quad. The bars show the slowdown when co-located with naive random access (naive), binary search tree (BST), the lbm benchmark (LBM Core), the 3d block data movement (Blockie), and our sledgehammer technique (Sledge), in that order. The `lbm` benchmark is used as a baseline to compare the synthetic engines. It is clear from the graphs that the `Naive` and `BST` approaches produce the smallest amount of

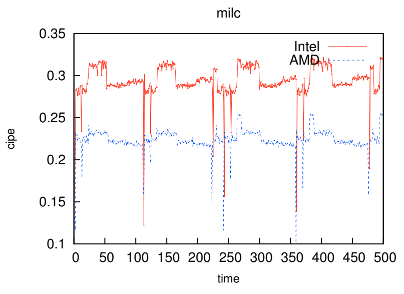

Figure 6: milc

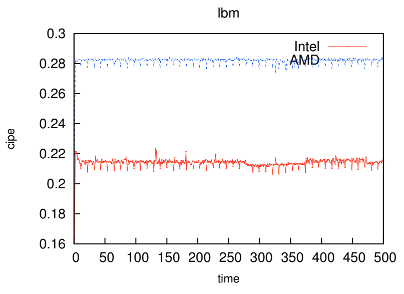

Figure 7: lbm

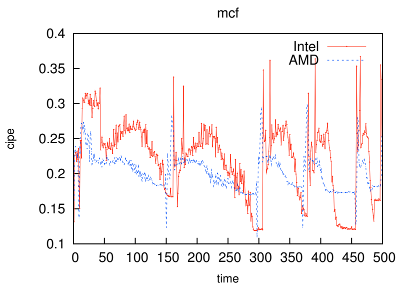

Figure 8: mcf

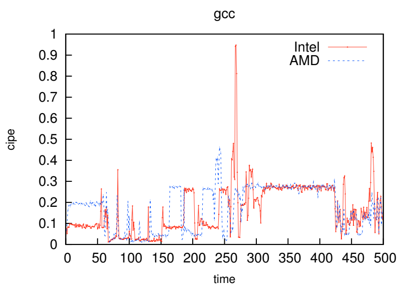

Figure 9: gcc

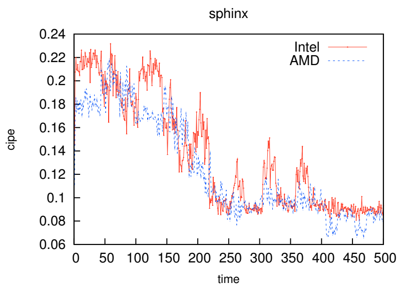

Figure 10: sphinx

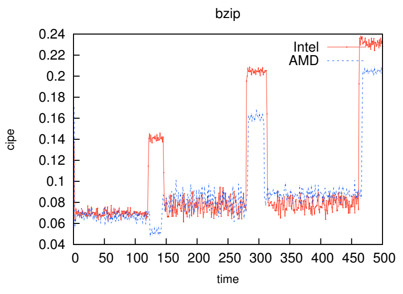

Figure 11: bzip

contention. However note that they do an adequate job of indicating the applications that are most sensitive to cross-core interference. The contention produced by `Naive` and `BST` is low as there is computation performed between single memory accesses. `Blockie` and `Sledge` touch large amounts of data in a single pass and with less computation. Note that our `Blockie` and `Sledge` techniques are more effective than using the most contentious of the SPEC benchmarks.

Across the two architectures the general trend is similar, although we do see some differences. We see that applications that tend to be sensitive to contention tend to be uniformly so across these two representative architectures. We also see that the varying contention synthesis designs rank similarly on both architectures. This general trend supports our hypothesis that contention is agnostic across this class of commodity multicore architectures.

Although the general trend is the same, there are some clear differences. For example, the benchmark most sensitive to cross-core interference on the two architectures differs. On Intel's architecture `mcf` shows the most significant degradation in performance, while on AMD's architecture `lbm` has the most significant degradation. These variations are due to the idiosyncrasies of the microarchitectural design.

The key observation is the effectiveness of the contention synthesis designs are mostly uniform across the different benchmark workloads. This trend supports our hypothesis that in addition to being generally agnostic across this class of commodity multicore architectures, it is also agnostic across the varying workloads and memory access patterns present in SPEC.

For our CiPE framework we finalized the design of our main CSE with a implementation based on Sledge as it most vividly illustrates contention.

6. EVALUATION

In this section we first present the results of our CIS analysis. We then demonstrate the practicality and usefulness of CiPE by using it to address two problems. The first problem is the selection of *contention-conscious* co-schedules for a batch of jobs to dynamically minimize cross-core performance interference and maximize overall throughput and performance. The second problem is to locate regions of code that, when executed dynamically, are highly sensitive to cross-core performance interference. We address this problem by designing a novel performance analysis and debugging tool using CiPE.

6.1 Evaluating CIS Analysis

Figures 6 to 11 show phase-level CIS scores calculated using our CiPE system for a representative selection of the SPEC 2006 benchmarks. CIS scores are calculated using samples collected at 1ms interval along the application's complete execution on `ref` input.

For each benchmark, the two lines in the graph indicate the CIS scores on the two different architectures Intel Core i7 and AMD Phenom X4. Both lines represent a complete execution and is smoothed to 500 data points. We selected representative key benchmarks from the SPEC2006 suite. The higher the CIS score is, the more sensitive the application is to cross-core interference. One important observation is that there are interesting clear phases in both some of the highly sensitive benchmarks (`milc`, `mcf`, `sphinx`) and in relatively insensitive benchmarks (`bzip`). There are also benchmarks that do not exhibit clear phases including both sensitive benchmarks (`lbm`) and insensitive benchmarks.

Profiling and discovering phase level characteristics of sensitivity is valuable for dynamic co-scheduling, whether the scheduling is done through a runtime system or OS. For example, as shown in Figure 10, `sphinx` is highly sensitive

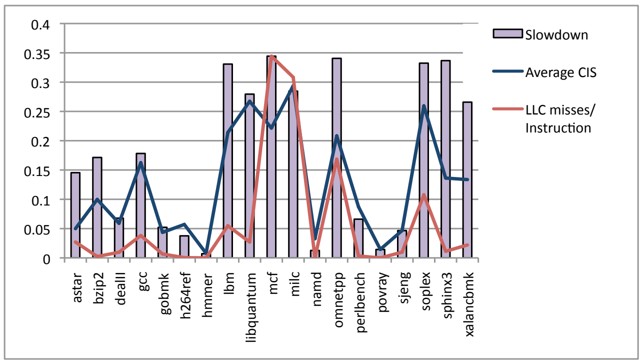

Figure 12: CIS score and LLC misses compared to slowdown when contending with LBM (Core i7)

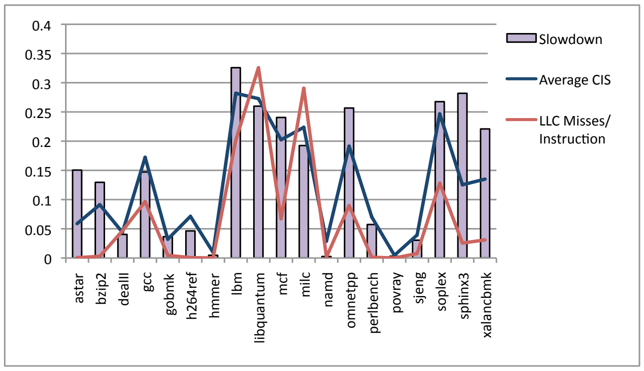

Figure 13: CIS score and LLC misses compared to slowdown when contending with LBM (Phenom X4)

during the first half of the execution but later its sensitivity drops. Thus an intelligent dynamic scheduler equipped with phase information can foresee peaks of contention sensitivity and schedule the application wisely according to the phase. In addition, as shown later in this section, we have designed a performance analysis and debugging tool that associates these phases to source code regions. Using this, users can identify code regions that are highly sensitive and optimize accordingly. Figures 12 and 13 show the average CIS scores calculated using Formula 5 for all C/C++ benchmarks in SPEC2006, compared against the performance degradation when each benchmark is co-running with `lbm`, on both Intel Core i7 and AMD Phenom X4. We also compare our CIS approach with using average last level cache (LLC) miss rates, as this approach is currently believed to be one of the best known indicators of contention sensitivity [29, 13]. In Figures 12 and 13 we present the cache miss rate using a line graph for each benchmark.

Our results show that generally, an application's average CIS score has a strong correlation with its performance degradation (e.g., a lower CIS scores indicate smaller degradations and vice versa). Although in a few cases our CIS scores is less representative of the actual degradation (**`sphinx`**, **`xalan`** and **`astar`**), we see that in general our CIS scores match the actual performance degradation much more closely than cache miss rates. These three benchmarks have more sporadic phases that we believe increased the inaccuracy on its average. However, notice that even in these cases, the CIS score significantly outperforms using last level cache miss rates. In addition, studying the phase level CIS scores for these types of applications would give more insight about their dynamic sensitivity.

6.2 Contention Conscious Scheduling

When two applications are co-scheduled on current commodity multicore architectures in a *contention oblivious* fashion, cross-core performance interference occurs, and ultimately system utilization and throughput suffer. This problem is especially worrisome in the data-center and cluster computing domains [3]. A CiPE based *contention-conscious* scheduling approach is especially well suited in these domains as the set of applications running on these systems are known, and system application scheduling plans and policies can be created offline. For example, Google, Microsoft, and Yahoo have a known set of applications that run in their data-centers, including Search, Maps, Mail, Video etc. An understanding of each application's sensitivity to cross-core interference can prove critical to improving throughput, responsiveness and even power and energy.

For our experimental setup we use the following scheduling model. We have a single batch of jobs to execute and two processing cores available. Jobs are selected to run concurrently with another job on the neighboring core. If any core becomes free, a job from the job queue is selected to run. For our experiment our queue consists of the 19 SPEC2006 benchmarks (C/C++). Each benchmark is run to completion on their **`ref`** inputs. We have performed this experimentation on both Intel's Core i7 and AMD's Phenom X4 multicore architectures.

As a baseline we show the effects of *cache oblivious* scheduling. Our *contention oblivious* scheduling is a random schedule where contentious applications are naively co-scheduled. Our CiPE based *contention conscious* scheduling heuristically places applications with high sensitivity to cross-core interference with applications with a low sensitivity. Every time a job completes, the scheduler selects a job from the queue with either the highest or lowest CIS score. If the currently running job is sensitive, a job with the lowest CIS score is selected. If the currently running job is insensitive the job with the highest CIS score is selected.

Figures 14 and 15 show the results of our experimentation. Figure 14 shows a significant reduction in the performance degradation that occurs due to cross-core performance interference. The bars show the execution time of each application while being co-scheduled over running alone on both the Core i7 and the Phenom X4. For each benchmark, the first and second bars show performance degradation when scheduled in a contention oblivious fashion and the third and forth bars show the degradation when scheduled based on CIS scores. As shown in the figure, for most benchmarks, the performance degradation dropped significantly using our CiPE-based approach.

Figure 15 summarizes the overall improvement in throughput when using CiPE to select *contention-conscious* colocations. Using our approach we were able to improve the performance of the contention sensitive SPEC2006 benchmarks by 12%, on average and up to 24% in the case of `mcf`. The remaining insensitive jobs only suffered a 1% performance impact from being co-scheduled with sensitive jobs.

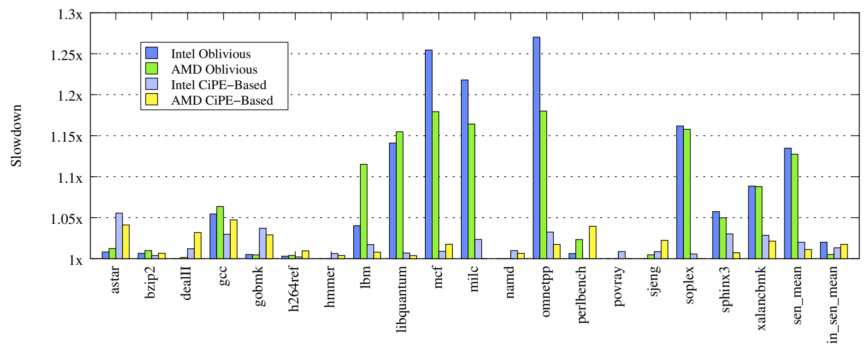

Figure 14: Contention Oblivious vs CiPE-based scheduling (lower is better)

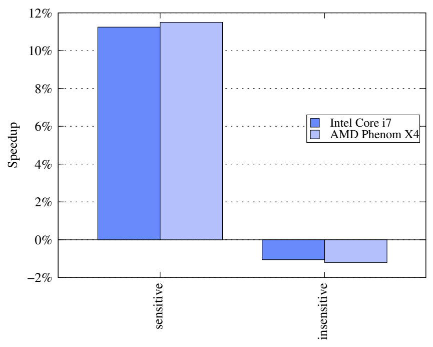

Figure 15: Performance Impact due to Co-Scheduling (higher is better)

```
CIS Score: 0.21033
Rank: 1 Occupies: 68% File: lbm.c Lines: 187 − 206
Rank: 2 Occupies: 32% File: lbm.c Lines: 257 − 168
```

Figure 16: lbm Profile Sample

```
CIS Score: 0.28453
Rank: 1 Occupies: 51% File: su3_proj.c Lines: 42 − 47
Rank: 2 Occupies: 49% File: s_m_a_mat.c Lines: 18 − 19
```

Figure 17: milc Profile Sample

6.3 Identifying Sensitive Code Regions

We also applied our CiPE framework to identify contentious code regions. Software developers and system analyzers can use this information for performance analysis, performance debugging, and to detect the most contention sensitive application phases and identify regions of source code responsible for this contention. We have developed a prototype performance debugger using CiPE.

Our performance debugger functions as a post processor of the profiles generated by our CiPE profiling environment. For each CIS sample generated by the CiPE profiling framework while executing an application, a record of the number of instructions executed since the previous sample is recorded. This record represents a region of executed instructions. This dynamic instruction trace can then easily be linked back to source level code using the standard `elf` debugging information.

Figures 16 and 17 show examples of the output of our CiPE performance debugging tool. In each figure the debuggers output from a single CIS sample is shown. As our debugger replays execution, a stream of these samples are printed to the screen or to a log file. For each CIS sample, our performance debugger points to the source-level basic blocks which were responsible for that sample's CIS score, and their dynamic coverage relative to each other. The number of blocks shown is a parameter set by the user; two or three is often covers more than 99% of the sample interval. Figures 18, 19, and 20 show the corresponding SPEC2006 code snippets. As these figures show, for `lbm` and `milc`, the most contentious regions of code are composed of dense array and memory operations.

```
186   SWEEP_START( 0, 0, 0, 0, 0, SIZE_Z )
187     if( TEST_FLAG_SWEEP( srcGrid, OBSTACLE )) {
188       DST_C ( dstGrid ) = SRC_C ( srcGrid );
189       DST_S ( dstGrid ) = SRC_N ( srcGrid );
190       DST_N ( dstGrid ) = SRC_S ( srcGrid );
191       DST_W ( dstGrid ) = SRC_E ( srcGrid );
192       DST_E ( dstGrid ) = SRC_W ( srcGrid );
193       DST_B ( dstGrid ) = SRC_T ( srcGrid );
194       DST_T ( dstGrid ) = SRC_B ( srcGrid );
195       DST_SW( dstGrid ) = SRC_NE( srcGrid );
196       DST_SE( dstGrid ) = SRC_NW( srcGrid );
197       DST_NW( dstGrid ) = SRC_SE( srcGrid );
198       DST_NE( dstGrid ) = SRC_SW( srcGrid );
199       DST_SB( dstGrid ) = SRC_NT( srcGrid );
200       DST_ST( dstGrid ) = SRC_NB( srcGrid );
201       DST_NB( dstGrid ) = SRC_ST( srcGrid );
202       DST_NT( dstGrid ) = SRC_SB( srcGrid );
203       DST_WB( dstGrid ) = SRC_ET( srcGrid );
204       DST_WT( dstGrid ) = SRC_EB( srcGrid );
205       DST_EB( dstGrid ) = SRC_WT( srcGrid );
206       DST_ET( dstGrid ) = SRC_WB( srcGrid );
207       continue;
208     }
```

Figure 18: lbm.c

```
38 void su3_projector( su3_vector *a,
   su3_vector *b, su3_matrix *c ){
39 register int i,j;
40 register double tmp,tmp2;
41     for(i=0;i<3;i++)for(j=0;j<3;j++){
42   tmp2 = a->c[i].real * b->c[j].real;
43   tmp = a->c[i].imag * b->c[j].imag;
44   c->e[i][j].real = tmp + tmp2;
45   tmp2 = a->c[i].real * b->c[j].imag;
46   tmp = a->c[i].imag * b->c[j].real;
47   c->e[i][j].imag = tmp - tmp2;
48     }
49 }
```

Figure 19: su3_proj.c

```
16 register int i,j;
17     for(i=0;i<3;i++)for(j=0;j<3;j++){
18 c->e[i][j].real=a->e[i][j].real+s*b->e[i][j].real;
19 c->e[i][j].imag=a->e[i][j].imag+s*b->e[i][j].imag;
20     }
```

Figure 20: s_m_a_mat.c

Our general CIS Analysis can be used for other performance debugger designs. For example, instead of replaying CiPE profiles, branch information can be efficiently extracted online using structures such as Intel's last branch record (LBR), and then linked back to the source level during the CiPE profile generation. Enabling these such modifications are matters of engineering. However, once the CiPE profiles are gathered, our post processing replay debugger provides the same functionality and suffers only 20% overhead over native execution.

7. RELATED WORK

In this paper we present a profiling and characterization framework for program sensitivity to cross-core interference on modern CMP architecture. Related to our work is a cache monitoring system for shared caches [28], which proposes novel hardware designs to facilitate better understanding of how applications are interacting and contending when running together. Similar to our work, the system is then used for profiling and program behavior characterization. However, in contrast to our methodology, this work requires hardware extensions and thus is evaluated using simulations. Our methodology and framework is applicable to current commodity multicore architectures. In addition, our framework is not limited to cache contention but any contention in the memory system that would impact performance, and can be produced by our CSE. For our co-scheduling scheme, the works by Knauerhase et al. [13] and Zhuravlev et al. [29] are related. These works both use last level cache misses to perform *contention-conscious* scheduling. They exploit the observation that a program's cache miss behavior tends to be good indicators of contentiousness to dynamically pair jobs with heavy cache usage with jobs with lower cache usage to reduce contention. Although CiPE requires a profiling phase, using CIS score has shown to be a be a stronger indicator than last level cache misses. Our system is not a replacement of these prior works, but our profiling results are complementary to dynamic scheduling because being able to characterizing a program's sensitivity and discovering phases from profiling can help design more sophisticated and intelligent dynamic schedulers.

In recent years, cache contention has received much research attention. Most works focus on exploring the design space of cache and memory proposing novel hardware solutions or managing policies to alleviate the contention problem. Hardware techniques and related algorithms to enable cache management such as cache partitioning and memory scheduler are proposed [25, 12, 22, 19]. Other solutions have been developed to guarantee fairness and QoS [17, 20, 10, 23]. Related to novel cache designs and architectural support, analytical models to predict the impact of cache sharing are also proposed by Chandra et al. [2]. In addition to new hardware cache management, other approaches manage the shared cache through the OS [24, 4]. Instead of novel hardware or software solution to managing shared caches, our solution focuses on the other side of the problem, namely the application's inherent sensitivity to interference on existing modern microarchitecture. Contention conscious scheduling schemes that guarantee fairness and increase QoS for co-running applications or multithreaded application have been proposed [13, 7, 1, 18]. Fedorova et al. used cache model prediction to enhance the OS scheduler to provide performance isolation [7]. There are also theoretical studies that investigate approximation algorithms to optimally schedule co-running jobs on CMPs [11]. Much work has been done for constructing general frameworks for memory profiling of applications [9, 21, 5], profiling techniques and methods to use such profiling to improve performance or help develop better compilers and optimizations [9, 26].

8. CONCLUSION

In this work, we present CiPE, a methodology and framework for the measurement and characterization of an application's cross-core interference sensitivity on current multicore micro-architectures. This framework is composed of a lightweight runtime environment on which a host application runs, along with a carefully designed *contention synthesis* engine that executes on a neighboring core. We have explored and evaluated four contention synthesis mechanisms and presented a general characterization methodology for application *cross-core interference sensitivity* (CIS Score). We have presented the design and implementation of this characterization methodology and framework, and demonstrate its application to the SPEC2006 benchmark suite on two real-world multicore architectures characterizing entire applications, their individual phases, and also source level code regions. We have also used our CiPE methodology to perform *contention conscious scheduling* that minimizes cross-core interference and significantly improves application performance and throughput. Using our CiPE-based scheduling, we improve the performance of contention sensitive SPEC2006 benchmarks by 12% on average, and beyond 24% in the case of `mcf` and `omnetpp`.

9. REFERENCES

[1] M. Banikazemi, D. Poff, and B. Abali. Pam: a novel performance/power aware meta-scheduler for multi-core systems. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[2] 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.

[3] 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. *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, 2006.

[4] S. Cho and L. Jin. Managing distributed, shared l2 caches through os-level page allocation. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 455–468, Washington, DC, USA, 2006. IEEE Computer Society.

[5] S. Eranian. What can performance counters do for memory subsystem analysis? In *MSPC '08: Proceedings of the 2008 ACM SIGPLAN workshop on Memory systems performance and correctness*, pages 26–30, New York, NY, USA, 2008. ACM.

[6] A. Fedorova, S. Blagodurov, and S. Zhuravlev. Managing contention for shared resources on multicore processors. *Communications of the ACM*, 53(2), 2010.

[7] A. Fedorova, M. Seltzer, and M. D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *PACT '07: Proceedings of the 16th International Conference on Parallel*

Architecture and Compilation Techniques, pages 25–38, Washington, DC, USA, 2007. IEEE Computer Society.

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

[9] M. Itzkowitz, B. J. N. Wylie, C. Aoki, and N. Kosche. Memory profiling using hardware counters. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 17, Washington, DC, USA, 2003. IEEE Computer Society.

[10] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. Qos policies and architecture for cache/memory in cmp platforms. 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.

[11] 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.

[12] 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.

[13] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using OS Observations to Improve Performance in Multicore Systems. *IEEE Micro*, 28(3), 2008.

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

[15] 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.

[16] J. Mars and M. Soffa. Synthesizing contention. *WBIA '09: Proceedings of the Workshop on Binary Instrumentation and Applications*, Dec 2009.

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

[18] A. Merkel, J. Stoess, and F. Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. *EuroSys '10: Proceedings of the 5th European conference on Computer systems*, Apr 2010.

[19] 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.

[20] 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.

[21] A. Pesterev, N. Zeldovich, and R. Morris. Locating cache performance bottlenecks using data profiling. *EuroSys '10: Proceedings of the 5th European conference on Computer systems*, Apr 2010.

[22] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 423–432, Washington, DC, USA, 2006. IEEE Computer Society.

[23] 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.

[24] 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.

[25] G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *HPCA '02: Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, page 117, Washington, DC, USA, 2002. IEEE Computer Society.

[26] Q. Wu, A. Pyatakov, A. Spiridonov, E. Raman, D. W. Clark, and D. I. August. Exposing memory access regularities using object-relative memory profiling. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 315, Washington, DC, USA, 2004. IEEE Computer Society.

[27] Y. Xie and G. H. Loh. Dynamic Classification of Program Memory Behaviors in CMPs. In *The 2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects*, 2008.

[28] L. Zhao, R. Iyer, R. Illikkal, J. Moses, S. Makineni, and D. Newell. Cachescouts: Fine-grain monitoring of shared caches in cmp platforms. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 339–352, Washington, DC, USA, 2007. IEEE Computer Society.

[29] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010.