Predicting Cross-Core Performance Interference on Multicore Processors with Regression Analysis

Jiacheng Zhao, Huimin Cui, Jingling Xue, *Senior Member, IEEE*, Xiaobing Feng

Abstract—Despite their widespread adoption in cloud computing, multicore processors are heavily under-utilized in terms of computing resources. To avoid the potential for negative and unpredictable interference, co-location of a latency-sensitive application with others on the same multicore processor is disallowed, leaving many cores idle and causing low machine utilization. To enable co-location while providing QoS guarantees, it is challenging but important to predict performance interference between co-located applications. We observed that the performance degradation of an application can be represented as a piecewise predictor function of the aggregate pressures on shared resources from all cores. Based on this observation, we propose to adopt regression analysis to build a predictor function for an application. Furthermore, the prediction model thus obtained for an application is able to characterize its contentiousness and sensitivity. Validation using a large number of single-threaded and multi-threaded benchmarks and nine real-world datacenter applications on two different platforms shows that our approach is also precise, with an average error not exceeding 0.4%.

Index Terms—cross-core performance interference, memory subsystems, multicore processors, performance analysis, prediction model.

1 Introduction

As the microprocessor industry is rapidly moving towards multi/many-core architectures and much of the world's computing is continuously moving into the cloud, multicore processors have been widely adopted to serve as the mainstream processors in datacenters, which house large-scale web applications and cloud services.

However, for modern datacenters, the utilization of computing resources is very low, i.e., around 20% [3], [25]. Researchers have observed that when multiple applications are co-located on the same multicore processor, contention for shared resources in the memory subsystem can cause severe cross-core performance interference [11], [25], [42], [43], [49], [56], [55], [54], [50]. When the datacenter houses some user-facing and latency-sensitive applications which have specific quality of service (QoS) requirements, such as web search, these applications might be interfered by co-location, causing negative and unpredictable QoS violation. As a result, co-location for such applications is disallowed, leaving many cores idle and causing low machine utilization [25]. To enable co-location while providing QoS guarantees, it is challenging but important to predict the performance interference between co-located applications.

Despite extensive efforts on mitigating the performance interference due to resource contention on multicore processors, not much work is directly applicable to the datacenter co-location problem. The majority of existing solutions [5], [16], [19], [24], [26], [27] classify *qualitatively* how aggressive an application is for shared resources and make co-location decisions accordingly. To predict *quantitatively* the amount of the performance degradation suffered by an application due to co-location, brute-force profiling is frequently used but impractical for a datacenter housing N applications (with C_N^m co-locations on an m-core processor), where N can be 1000+. To alleviate this problem, Bubble-Up [25] characterizes the sensitivity and aggressiveness of an application by co-running it with a stress-testing application (called the *bubble*). However, it is limited to predicting the performance interference between two co-running applications only. Bandit [9] does not have this limitation but focuses on bandwidth contention only. SMiTe [54] aims to predict performance interference between SMT co-locations.

In our recent research, we have empirically observed that the performance degradation of an application can be represented as a *piecewise* predictor function of the *aggregate pressures* on shared resources from all cores, regardless of which applications are co-running and what their individual pressures are. In particular, the piecewise function indicates that different dominant contention factors can be accommodated more accurately with different subfunctions in its different subdomains.

Based on the above observation, we introduce an empirical approach to efficiently and precisely predicting the performance degradation suffered by an application due to arbitrarily many co-located applications. To build a precise predictor function efficiently, we proceed in two phases. The key lies in decoupling the construction of the piecewise functional relation itself from that of its coefficients. The first phase uses training workloads to build an abstract model, which defines the functional form used to relate the performance degrada-

tion of *any* application to the aggregate pressures on shared resources from all cores, with its coefficients undetermined. This phase is platform-dependent but application-independent. The second phase determines the application-specific coefficients for the functional form obtained earlier. Thus, the more costly first phase can be amortized by all applications in a datacenter.

This paper makes the following contributions:

- We present empirical evidence for the existence of a piecewise functional relation between the performance degradation of an application and the aggregate pressures on shared resources from all cores, regardless of what co-running applications and their individual pressures are.
- We introduce a two-phase regression approach to building a predictor function. Our approach is efficient because the first phase is performed only once for a platform and the second phase can be done in $O(1)$. Our approach is also precise as it has an average error not exceeding 0.4%, validated using a large number of single- and multi-threaded benchmarks as well as nine real-world datacenter applications on two platforms.
- We extend our prediction model to characterize the inter-thread contention for multi-threaded applications in terms of accumulated CPU cycles consumed by all threads. With the model we can predict the increased CPU cycles caused by inter-thread contention, which directly affects the scalability of multi-threaded applications. Evaluations using representative PARSEC benchmarks show that our approach is precise to predict the consumed CPU cycles from all threads, with an average error of 0.56%.
- We evaluate our prediction model in terms of its prediction precision and compare the performance interference on representative Intel and AMD multi-core processors. Our experimental results also show that inclusive caches can cause severe performance interference for some applications.

The rest of the paper is organized as follows. Section 2 motivates this work. Section 3 presents our regression approach. Section 4 describes our approach to characterize inter-thread contention for multi-threaded applications. Section 5 presents our experimental validation. Section 6 discusses the related work. Section 7 concludes.

2 Motivation

We have two key insights from our empirical observations. First, the performance degradation suffered by an application due to co-location can be represented as a predictor function of the aggregate pressures on shared resources from all cores, regardless of which applications are co-running and what their individual pressures are. Second, a predictor is piecewise in order to capture different dominant contention factors more accurately with different subfunctions. In this section, we use `429.mcf` from SPEC2006 to introduce the key elements involved in building a prediction model.

In this paper, the performance degradation of a benchmark is computed by:

$$PD = (ExeTime_{co-run} - ExeTime_{solo})/ExeTime_{solo}$$

where $ExeTime_{solo}$ ($ExeTime_{co-run}$) is its execution time in solo (co-running) execution.

The platform we use for our motivation example is an Intel quad-core Xeon E5506 with 32KB L1 DCache, 32KB L1 ICache, 256KB L2 cache, a 4MB shared L3 cache and a memory bandwidth of 12.8GB/s. We focus on two shared resources, shared cache and memory bandwidth. We generated randomly 200 workloads (with three applications per workload) from SPEC2006 to co-run with `429.mcf`. For each workload, we calculate the aggregate pressure for each shared resource and seek for a functional relation with the performance degradation of `429.mcf`.

Measuring Aggregate Pressures: For a given workload, let the three co-runners of `429.mcf`, denoted A_{mcf}, be A_1, A_2 and A_3. First, we use PMUs to collect each benchmark's pressure on (i.e., consumption of) each shared resource in solo execution. Let $cache_i$ (bw_i) be the individual pressure on shared cache (bandwidth) from A_i, where $i \in \{1, 2, 3, mcf\}$. We then combine the individual pressures on a resource to obtain the aggregate pressure on the same resource:

$$\begin{array}{rcl} P_{cache} & = & cache_{mcf} + \Sigma_{i=1}^{3} cache_i \\ P_{bw} & = & bw_{mcf} + \Sigma_{i=1}^{3} bw_i \end{array} \quad (1)$$

Collecting Data Points: For each workload w, the performance degradation of `429.mcf` is recorded as PD_w and the aggregate pressures P_{cache} and P_{bw} are found by (1), giving rise to one data point, denoted $((P_{cache}, P_{bw}), PD_w)$.

Finding the Functional Relation: With 200 randomly generated workloads to co-run with `429.mcf`, we obtain 200 data points for `429.mcf`. The Xeon platform used for this experiment has a bandwidth of 12.8GB/s. With [0, 12.8GB/s] being partitioned into three bandwidth bands (as described in Section 3.2), we obtain the following piecewise function:

$$PD_{mcf} = \begin{cases} PD_{Cache-Bound} & \text{if } P_{bw} < 3.2 \\ PD_{Cache/BW-Bound} & \text{if } 3.2 \leq P_{bw} \leq 9.6 \\ PD_{BW-Bound} & \text{if } P_{bw} > 9.6 \end{cases} \quad (2)$$

where

$$\begin{array}{rcl} PD_{Cache-Bound} & = & 0.485P_{bw} + 0.183P_{cache} - 0.138 \\ PD_{Cache/BW-Bound} & = & 0.706P_{bw} + 1.725P_{cache} - 0.220 \\ PD_{BW-Bound} & = & 0.907P_{bw} + 3.087P_{cache} - 0.561 \end{array} \quad (3)$$

The R-squared value for PD_{mcf} is 0.90, indicating a strong fit. Figure 1 plots the three subfunctions of PD_{mcf} given in (3) that capture three different types of dominant contention factors. The performance degradation of `429.mcf` varies at different rates in the three corresponding subdomains. This sheds some light on the precision behind our prediction.

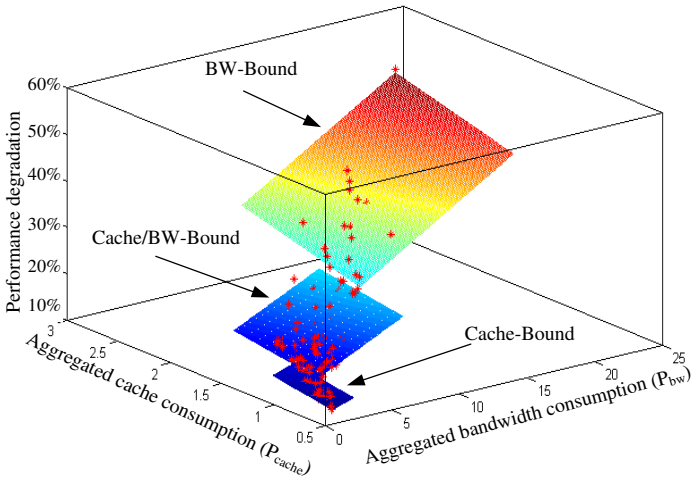


Fig. 1: Existence of a piecewise function relating the performance degradation of `429.mcf` to the aggregate pressures on shared cache and memory bandwidth. The three planes are the plots of the three subfunctions given in (3).

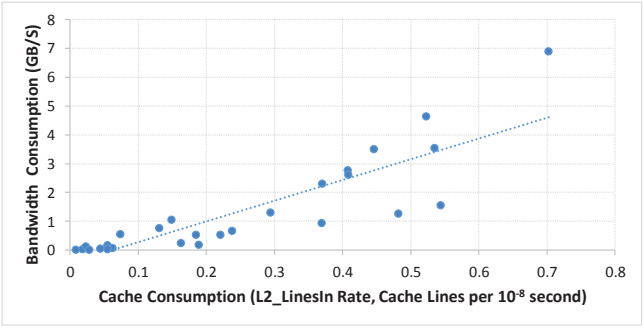


Fig. 2: Correlation between cache and bandwidth consumptions, with each data point standing for one application from a set of 29 applications in SPEC2006.

3 Our Regression Approach

Our insights lead to the design of a brute-force regression approach to determine the functional relation between the co-running applications and the performance degradation. However, the brute-force approach would be impractical for a datacenter housing a large number of applications. Therefore, we introduce a scalable two-phase approach to solving the datacenter co-location problem. In this section, we first discuss how to characterize an application into a vector for regression analysis and then present the two-phase approach.

3.1 Extracting Feature Vectors

As discussed in Section 2, we characterize an application using its individual consumptions of (pressures on) shared resources in solo execution. We do so by making use of PMUs. By now, we focus on two shared resources, shared cache and memory bandwidth. The critical issue here is to identify appropriate PMUs to use. For illustration purposes, we continue to consider the Intel platform described in Section 2, with private L1 and L2 caches and a shared L3 cache.

- *Shared cache consumption.* To measure the shared cache (L3) consumption, *L2_LinesIn rate* is recommended [41], [43], [55], which gives the number of cache lines brought from the L3 cache for a given period of time, including passive accesses triggered by cache misses and proactive prefetching.
- *Shared memory bandwidth consumption.* An application's bandwidth consumption can be profiled using Intel's PTU (Performance Tuning Utility) [13], which offers the hardware event counters for memory system performance analysis. PTU can report the *System Memory Throughput* periodically. In our experiments, we use the average throughput of an application as its bandwidth consumption.

Therefore, an application A is characterized into a feature vector of the form $FV(A) = (cache_i, bw_i)$. We observed that $cache_i$ and bw_i are not independent variables as they are related by the same memory hierarchy. Figure 2 depicts their correlation for the 29 SPEC2006 benchmarks. A benchmark is represented by a point, with the horizontal axis representing its cache consumption and the vertical axis for its bandwidth consumption.

From Figure 2, a strong and positive correlation can be observed. The Pearson product-moment correlation coefficient can be up to 0.867, meaning that multicollinearity for cache and bandwidth consumptions exists. In multiple regression, multicollinearity can make the estimation of regression coefficients unreliable [10], [18], especially for our prediction outside the training set used. In order to construct our prediction model using regression analysis and eliminate potential multicollinearity, we conduct principal component analysis (PCA) as recommended by previous work [10], [18]. PCA uses an orthogonal transformation to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables called principal components.

3.2 Two-Phase Regression Approach

For the datacenter's co-location problem, we should do more than the brute-force approach, because a datacenter may house hundreds to thousands of applications with the frequent development and updating of these applications. Repeating the same regression analysis as stated in Section 2 for each application is impractical. Fortunately, we have made an important observation about a predictor function used for an application on a given computer platform: its coefficients are application-specific but the actual functional relation, i.e., form itself is not. We introduce a two-phase regression approach, as shown in Figure 3, to build a predictor function efficiently for an application.

The first phase is platform-dependent but application-independent. This phase builds an abstract prediction model, i.e., a piecewise function to be shared by all

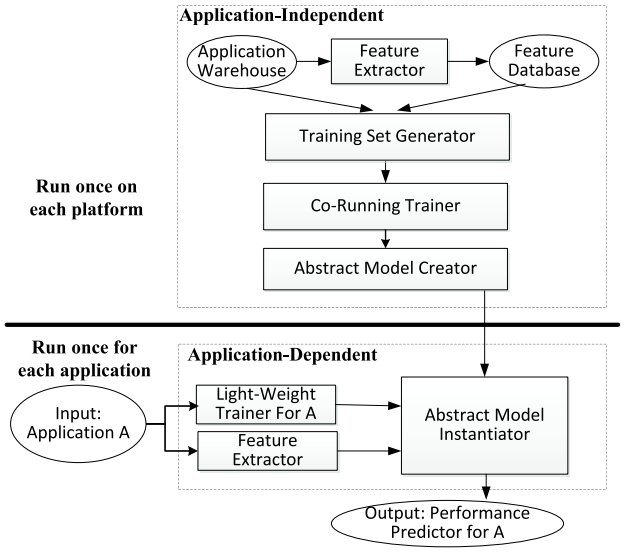


Fig. 3: Two-phase approach for predicting the performance degradation of an application.

applications in a datacenter, with its coefficients undetermined. The second phase instantiates the abstract model for a given application to determine its application-specific coefficients. By decoupling the construction of the two components, the more costly first phase is performed only once for a platform, with its cost being amortized by all applications in a datacenter.

Section 3.2.1 describes how to build an abstract prediction model. Section 3.2.2 describes how to instantiate the abstract model for a given application. The analysis and evaluation of efficiency of our approach can be found in Appendix B.

3.2.1 Phase 1: Building an Abstract Model

All the components of this first phase are shown in the top part of Figure 3. The "Application Warehouse" contains all applications routinely run in a datacenter. The "Feature Extractor" is responsible for obtaining each application's individual consumptions of (pressures on) shared resources and storing these as a feature vector in the "Feature Database" (Section 3.1). The feature vectors are used to compute the aggregate pressures of co-runners on shared resources and allow the "Training Set Generator" to generate training workloads (Section 3.2.1.1). The "Co-Running Trainer" records the performance slowdowns of all training workloads (Section 3.2.1.2). The "Abstract Model Creator" builds an abstract prediction model via regression analysis (Section 3.2.1.3).

3.2.1.1 **The Training Set Generator:** A datacenter warehouse can contain a large number of applications, say, 1000+. It is impractical to use them all as training workloads. We create a training set so that the warehouse is evenly sampled based on the feature vectors of all applications stored in the "Feature Database". This ensures that different degrees of contention for shared resources are all represented by the training workloads.

TABLE 1: An interference table for a m-core processor, with each co-running workload $W_{A_i,j}$ containing $m-1$ co-runners.

Application	Co-Runners	$A_i's$ Performance Degradation
A_1	$W_{A_1,1}$	$PD_{A_1,W_{A_1,1}}$
	...	
A_1	$W_{A_1,Q}$	$PD_{A_1,W_{A_1,Q}}$
A_2	$W_{A_2,1}$	$PD_{A_2,W_{A_2,1}}$
	...	
A_2	$W_{A_2,Q}$	$PD_{A_2,W_{A_2,Q}}$
	...	

We proceed in three steps. First, we define an n-dimensional feature space, with one dimension representing the consumption of each distinct shared resource. In this paper, the feature space has two dimensions. Each application is mapped into the feature space using its feature vector. Second, we partition the feature space into $N_{cache} \times N_{bw}$ grids, where N_{cache} and N_{bw} are user-supplied values. The applications falling into the same grid can be regarded as having a similar resource consumption. Finally, we sample one point from each non-empty grid by adding it to the training set.

3.2.1.2 **The Co-Running Trainer:** Given an m-core processor, the co-running trainer randomly generates a set of workloads of size m from our training set, launches these workloads one by one, and records every application's degradation.

When generating co-running workloads, there is no need to enumerate all possible co-locations from the training set, which can be too expensive. Instead, we guarantee that every application appears in Q different workloads (with no duplicates) to provide enough data points for our regression analysis, where Q is a user-supplied parameter.

The training data are organized into an *interference table*, shown in Table 1, where each row has three columns: an application A_i, its set $W_{A_i,j}$ of $m-1$ co-runners, and the performance degradation $PD_{A_i,W_{A_i,j}}$ of A_i in this j-th workload. For each workload, m new rows are added to the table, one for each application in the workload.

3.2.1.3 **The Abstract Model Creator:** Given the interference table for training applications and their feature vectors, we (1) compute the aggregate consumptions of shared resources, (2) identify the subdomains for the piecewise predictor function being created, and (3) determine their corresponding subfunctions but leave their coefficients undetermined. To accomplish these tasks, we provide an interface for the user to define a model search space via a configuration file. Our model creator will automatically search this space to find an optimal solution by performing a regression analysis.

Configurations: Figure 4 shows the syntax of a configuration file. Each option admits multiple values so that many different configurations can be tried to find the best solution.

- *Aggregation.* The "pre-processing" option allows the user to specify how to process the individual re-

#Aggregation
  #Pre-Processing: none/exp(p)/log(p)/pow(p)
  #Mode: add/mul
#Domain Partitioning: (shared-resource$_1$, condition$_1$), ...
#Function: linear/polynomial(p)/*user-defined*

Fig. 4: Syntax of a configuration file.

source consumptions recorded in a feature vector before they are aggregated. There are presently four choices: *none*, *exp*(p), *log*(p) and *pow*(p), which transform v into v, p^v, $\log_p v$ and v^p, respectively. The default is *none*.

The "mode" option specifies an arithmetic operator used for combining the pre-processed resource consumptions of co-runners for a shared resource into their aggregate pressure on the same shared resource. There are presently two choices: *add* and *mul*, with *add* set as the default.

- *Domain Partitioning.* This option allows the user to define the subdomains of a predictor function in terms of shared resource pressures. For each (*shared-resource*$_i$, *condition*$_i$), *shared-resource*$_i$ is a list of shared resources, which is (P_{cache}), (P_{bw}) or (P_{cache}, P_{bw}), and *condition*$_i$ is a conditional expression in terms of variables in *shared-resource*$_i$. As a shorthand, *equal*$(n_1, n_2, \dots)$ is a condition indicating that the j-th resource in *shared-resource*$_i$ is partitioned into n_j equal bands. One example is (P_{bw}, *equal*(4)). The user can leverage some empirical knowledge to perform this task. In particular, Tang *et al.* [41] observed that contention for bandwidth has a more dominating effect on performance interference than for other shared resources and Xu *et al.* [49] observed that performance degradation worsens when bandwidth consumption approaches the system peak bandwidth. Some arbitrary user-defined conditions are also admitted.
- *Function.* This option specifies the functional form used for interpolation. The default is *linear*, i.e., *polynomial(1)*, indicating a functional form of $a1 \times P_{cache} + a2 \times P_{bw} + a3$. Some *user-defined* functions are also allowed.

Regression Analysis: We find the best piecewise predictor function as follows. We try all possible configurations and pick the one with the largest average R-squared value, indicating the best fit possible. Given a data set consisting of n observed values x_i each of which has an associated predicted value $\hat{x_i}$, let $\bar{x} = \sum_{i=1}^{n} x_i/n$ be the average of the observed values. The R-squared value is $1 - \sum_{i=1}^{n}((x_i - \hat{x_i})^2/(x_i - \bar{x})^2)$.

In a configuration file, there are four options, with each specifying a set of values. Collectively, they define a set $\mathcal{C}$ of configurations as their Cartesian product to be searched for. Let $T = \{A_1, \dots, A_P\}$ be the training set of size P. For every application $A_i \in T$, let R_i be the set of Q rows in the interference table given in Table 1 that contains the performance slowdowns for A_i. For every $(c, A_i) \in \mathcal{C} \times T$, let $D(c, A_i)$ be the set of Q data points created from R_i, one from each row of R_i, as follows. For a row $(A_i, W_{A_i,j}, PD_{A_i,W_{A_i,j}})$ in R_i, the feature vectors for its associated applications are available in the "Feature Database". The data point obtained from the row is $((P_{cache}, P_{bw}), PD_{A_i,W_{A_i,j}})$, where P_{cache} and P_{bw} are the aggregated pressures on shared cache and bandwidth computed according to the configuration c.

Let $f(c, A_i)$ be the interpolating function over $D(c, A_i)$. Let $Fitness(c, A_i)$ be the R-squared value of $f(c, A_i)$. Let $AVG\_Fitness(c)$ be the average of the R-squared values for all applications in the training set T under configuration c:

$$AVG\_Fitness(c) \quad = \quad \sum_{i=1}^{P} Fitness(c, A_i)/P \qquad (4)$$

The best prediction is made under configuration c_{opt} if

$$AVG\_Fitness(c_{opt}) \quad \geq \quad \max_{c' \in \mathcal{C}} AVG\_Fitness(c') \quad (5)$$

We also record the best predictor functions found for all applications in the training set under the best configuration:

$$best\_funs \quad = \quad \{f(c_{opt}, A) | A \in T\} \qquad (6)$$

We have also tried a few more sophisticated interpolation methods. However, the one described above is fast and precise for the co-location problem addressed here, as evaluated later.

3.2.2 Phase 2: Instantiating the Abstract Model

All the components of this second phase are shown in the bottom part of Figure 3. For an application A, we instantiate the abstract model obtained earlier by determining its application-specific coefficients. If A is in the training set, we are done, because its coefficients are already recorded in (6) in the first phase. Otherwise, we proceed in the following four steps:

- **Step 1. Determining the Feature Vector for** A. This is done only if it is not in the "Feature Database".
- **Step 2. Generating Co-Running Workloads.** We build a set CS of workloads from the training set to co-run with A. CS contains C points for each subdomain, where C is set by the user based on the functional form found. So $|CS|$ is $C \times S$, where S is the number of subdomains. To ensure that C points are sampled evenly from the training set in each subdomain, we generate all C_P^{m-1} possible co-locations from the training set, where P is the size of the training set and m is the number of cores. We map these workloads into a two-dimensional space, one for P_{cache} and one for P_{bw}. Finally, we partition each subdomain evenly into C strips/grids. Then one point is sampled from each strip/grid. If some strips/grids are empty, the partitioning is refined until C points are sampled.
- **Step 3. Creating the Interference Table for** A. For each workload in CS, we co-run the $m - 1$

applications in the workload with A, record the performance degradation of A, and finally, create its interference table.

- **Step 4. Determining the Coefficients for A.** With A's interference table and the abstract model obtained earlier in Section 3.2.1, we perform a regression analysis to determine A's coefficients and obtain the instantiation of the abstract model for A.

This second phase takes $O(1)$ as $C \times S$ is small relative to the number of workloads run in the first phase.

4 Using the Model To Characterize Inter-Thread Contention For Multi-Threaded Applications

Besides predicting the performance interference between simultaneously running applications, our model can further be used to characterize the inter-thread contention for multi-threaded applications. Inter-thread contention is a significant issue affecting the scalability of multi-threaded applications.

It is well-known that for multi-threaded applications, scalability is a significant issue. Some existing works focus on providing performance metrics regarding to scalability for multi-threaded applications on multi/many-core systems [1], [22]. Besides, some tools are built to analyze the scalability of applications [12]. Furthermore, some concurrent data structures and synchronization schemes have been proposed to improve scalability for multi-threaded applications [32], [29]. In addition, researchers also investigate sensible resource allocation for multi-threaded programs [21], [40]. To predict bandwidth demands of multi-threaded applications, DraMon has been developed [46].

Earlier researches on the scalability of multi-threaded applications have demonstrated that two factors would negatively affect an application's scalability: thread synchronization and inter-thread resource contention. In this paper, we only focus on the inter-thread resource contention. Similar with the resource contention between simultaneously running applications, memory subsystem is the major source of inter-thread contention for multi-threaded applications [46]. In this section, we first show how shared resource contention would affect the scalability of multi-threaded applications on CMPs (Section 4.1), and then leverage our performance interference model to quantitatively characterize inter-thread contention for shared resources (Section 4.2).

4.1 Inter-Thread Resource Contention

For a multi-threaded application, the execution time is usually used as the performance metric, as suggested by [1], [22]. Its execution time can be divided into two parts: CPU execution time and CPU waiting time. The CPU execution time represents the accumulated CPU cycles consumed by all threads for executing the instructions, and the CPU waiting time represents the cost for thread synchronization. In this paper, we only focus on the CPU execution time, and the scalability issues caused by thread synchronization is beyond the scope of this paper. Therefore, we use the accumulated CPU cycles as our performance metric, and we will discuss the relation between the CPU cycles and the execution time in Section 5.

```
#pragma omp parallel for
for( j = 0; j < STREAM_ARRAY_SIZE; j++)
        c[j] = a[j] + b[j];
```

Fig. 5: Synthesized STREAM kernel ρ.

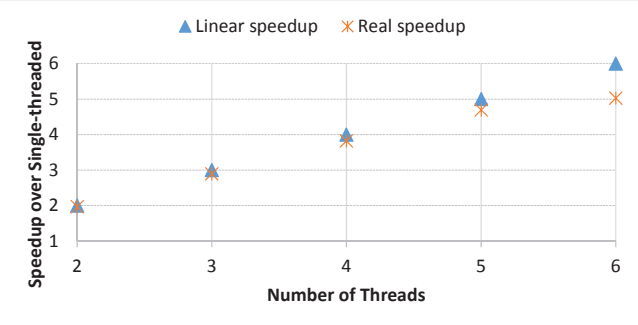


Fig. 6: Scalability of ρ on a six-core platform.

We use STREAM [31] to synthesize a benchmark to demonstrate how the memory subsystem contention affects the scalability of a multi-threaded application. We denote the benchmark as ρ, which is shown in Figure 5 and is synthesized following two principles. First, ρ includes one loop body which is parallelized using OpenMP. Second, the working set of ρ is evenly partitioned across all threads, and each thread accesses its own partition without interleaving or overlapping. In particular, ρ does not include lock operations or data sharing between threads. Only one global barrier is introduced after the parallel loop, and its cost can be ignored due to the long running loop body. Therefore, ρ's scalability is only affected by the inter-thread resource contention.

We use a six-core Intel E5645 platform with a memory bandwidth of 10.67GB/s (Details in Section 5) for our discussion. Figure 6 shows our experimental results for ρ's scalability on the platform, with the horizontal axis representing the number of threads and the vertical axis representing the speedup over one thread. The orange crosses show the scalability of ρ, meanwhile, the ideal linear speedup is shown for comparison by the blue triangle spots. From the figure, we can observe that the gap between the real and the ideal speedups will increase as the number of threads increases. As the design principle of the benchmark ρ, there is no lock synchronization and the cost of the global barrier can be ignored, so the scalability is only affected by the inter-thread resource contention. In particular, inter-thread contention delays memory accesses and makes each thread consume more cycles waiting for memory accesses.

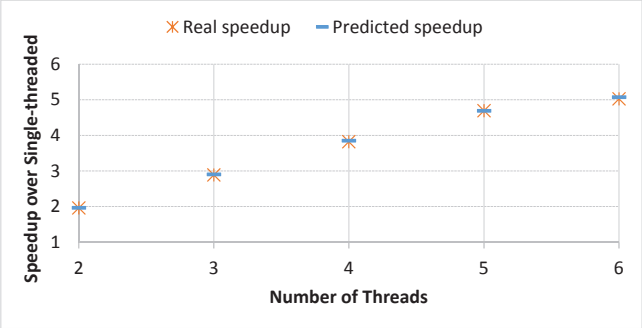


Fig. 7: Speedup computed using execution time and CPU cycles of ρ.

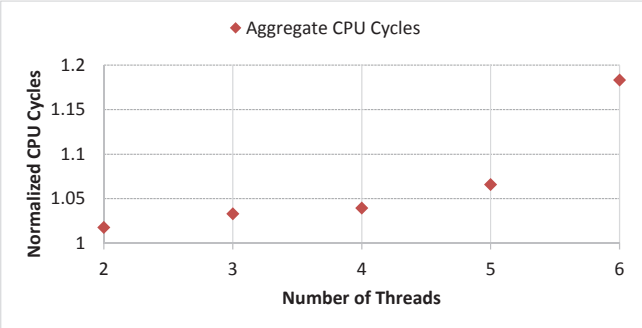


Fig. 8: Increment of aggregate CPU cycles of ρ.

To demonstrate that the accumulated CPU cycles is a reasonable metric when CPU waiting time is ignored, we compute the speedup using the CPU cycles, and compare it with the real speedup which is computed using execution time. Note that when only one thread is used, the CPU cycles equal to the execution time. Figure 7 shows the result for ρ, with the orange crosses representing the speedup computed using the execution time, and the blue lines representing the speedup computed using the profiled CPU cycles. In particular, the consumed CPU cycles are profiled using Intel VTune, which accumulates the CPU cycles from all threads. According to Figure 7, speedup computed using CPU cycles matches the real speedup very well, which confirms that CPU cycles is a reasonable metric when CPU waiting time is ignored.

Figure 8 shows ρ's accumulated CPU cycles from all threads, which is normalized to one thread. It shows that ρ consumes more CPU cycles as the number of threads increases. In particular, when 6 threads are used, the consumed CPU cycles will increase by nearly 20%. Since the metric of consumed CPU cycles does not include any cost of synchronization, the 20% slowdown is caused by inter-thread contention. In particular, a n-threaded application can be regarded as one single-threaded application contending with $n-1$ single-threaded co-runners. Therefore, we use our model to characterize such inter-thread contention.

4.2 Predicting Inter-Thread Contention

We ignore all sequential regions in the multi-threaded application and only focus on the parallel regions. As for predicting the performance interference between multiple applications, we keep assuming that the number of threads is equal or less than the number of cores. Furthermore, this paper focus only on the applications with the thread synchronization scheme being locks or barriers, because the execution time of such applications can be clearly divided into CPU execution time and CPU waiting time. Applications using other synchronization schemes, e.g., atomic operations, are beyond the scope of this paper. First, we create the performance prediction model for an application with the number of threads set to 1. Second, we regard an application with n threads as n co-running single-threaded applications, and predict the increased CPU cycles, as discussed below.

Given a multi-threaded application A, which includes m parallel regions, let A_j^i represents the j-th parallel region executed with i threads. We rewrite the formula to compute performance degradation as:

$$PD = \frac{(CPUCycles_{co-run} - CPUCycles_{solo})}{CPUCycles_{solo}}$$

where $CPUCycles_{solo}$ ($CPUCycles_{co-run}$) is the accumulated CPU cycles of A in solo (co-running) execution. We use CPU cycles instead of execution time to exclude the thread synchronization cost for multi-threaded application and focus on the CPU execution time. We create a prediction model for each parallel region $A_j^1(j = 1, .., m)$ using the approach in Section 3.2. Let $f_j(j = 1, ...m)$ be the predictor function for the parallel region A_j, and let $cache_j^1$ and $bw_j^1(j = 1, ..., m)$ be the pressure of A_j^1 on shared cache and bandwidth, respectively.

Now we use the model f_j to predict accumulated CPU cycles of $A_j^n(j = 1, ..., m)$, where n represents the number of threads. When a parallel region is executed with n threads, three issues will affect the consumed CPU cycles. First, the number of instructions will increase due to the cost of thread creation and termination. For long-running parallel regions, the cost of thread creation and termination is ignorable. In particular, for PARSEC[4], the overhead is less than 2% for most applications. Second, data sharing between threads will make positive effects and slightly reduce the CPU cycles. This issue is also ignorable as discussed in [52]. Third, inter-thread resource contention will cause performance interference and consume more CPU cycles for execution, and it dominates the increased CPU cycles.

Therefore, the accumulated CPU cycles can be predicted as:

$$CPUCycles_j^n = CPUCycles_j^1 * (1 + f_j(P_{cache}^1 * n, P_{bw}^1 * n) \quad (7)$$

where $CPUCycles_j^1$ is the consumed CPU cycles when the region is executed with single thread, f_j is the predictor function of the region obtained above, P_{cache}^1 and

P^1_{bw} are the pressure on the shared cache and bandwidth when the region is executed with single thread respectively, and $P^1_{cache} * n$ and $P^1_{bw} * n$ are the accumulated pressure of the n threads.

5 Evaluation

We demonstrate using a large number of benchmarks and nine real-world applications available to us that our approach can build a precise prediction model for an application efficiently. The main platform used is an Intel 2.13GHz quad-core Xeon E5506 with a private 32KB L1 D-cache, a private 32KB L1 I-cache, a private 256KB L2 cache, a shared 4MB L3 cache and a memory bandwidth of 12.8GB/s (with only two channels populated). The other platform is a octa-core Intel Xeon E7-8830.

Section 5.1 describes our evaluation methodology and the set of benchmarks used. Section 5.2 evaluates the precision of our approach. Section 5.3 reports briefly our results for larger Intel octa-core platform. Section 5.4 presents our results on predicting the scalability of multi-threaded applications. Section 5.5 analyzes the effect of different cache designs on performance interference.

5.1 Methodology and Benchmarks

We build a warehouse including the benchmarks from SPEC2000, SPEC2006, LINPACK, PARSEC [4], and Graph 500 and the nine real-world datacenter programs listed in Appendix D with a total of 506 applications. All are compiled using "GCC -O3" under Linux (kernel 2.6.18).

5.1.1 Applying Phase 1

In the first phase, we created a training set as Section 3.2.1.1. With $N_{cache}=6$ and $N_{bw}=10$, there are 30 applications selected since half of the $N_{cache} \times N_{bw} = 60$ grids are empty. During training, we collected $Q=200$ data points for each application as per Section 3.2.1.2. With the 30 applications in the training set, we ran $30 \times 200/4(\text{cores})=1500$ workloads. In the configuration file given in 4, "pre-processing" is {*none*, *exp(2)*, *log(2)*, *pow(2)*}, "mode" is {*add*, *mul*}, "domain partitioning" is $\{((P_{bw}), equal(4)), ((P_{cache}), equal(4)), ((P_{cache}, P_{bw}), equal(4,4))\}$, and "function" is {*linear*, *polynomial(2)*}. Thus, we conducted regression analysis for a total of $4 \times 2 \times 3 \times 2 = 48$ configurations, which took under 15 minutes using MATLAB, to build the abstract model. Finally, the best piecewise function found is:

$$PD = \begin{cases} a11P_{cache} + a12P_{bw} + a13 & \text{if } P_{bw} < 3.2 \\ a21P_{cache} + a22P_{bw} + a23 & \text{if } 3.2 \le P_{bw} \le 9.6 \\ a31P_{cache} + a32P_{bw} + a33 & \text{if } P_{bw} > 9.6 \end{cases} \quad (8)$$

The domain for bandwidth was initially partitioned into four equal bands. However, the subfunctions in the middle two bands are merged because they are identical.

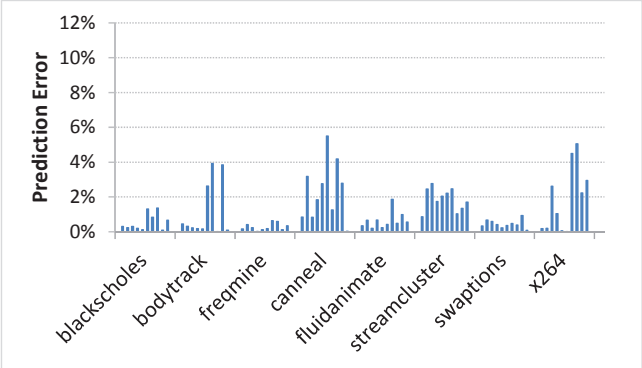


Fig. 10: Prediction precision for eight PARSEC benchmarks, with each co-running with 10 workloads generated from SPEC20006 and PARSEC benchmarks.

5.1.2 Applying Phase 2

In the second phase, we follow the four steps described in Section 3.2.2 to instantiate the above abstract model for a new application A. As the function is linear, we sample four workloads (or points) from each of the three subdomains to obtain $4 \times 3 = 12$ workloads, create an interference table and determine its application-specific coefficients. We can then use the instantiated model to predict its performance degradation when co-located. And the predictor functions for eight SPEC2006 benchmarks can be found in Appendix A. For `433.milc`, the subfunction under $P_{bw} < 3.2$ does not exist, because its own bandwidth consumption is larger than 3.2GB/s.

5.2 Prediction Precision

We show that our predictors (with some given in Appendix A) are precise for both benchmarks and real-world applications.

5.2.1 Single-Threaded SPEC Benchmarks

We focus on the 18 SPEC2006 benchmarks that are not included in the training set. We randomly generated 200 co-running workloads from these 18 benchmarks and randomly picked one representative for each workload. For the 200 representative workloads selected, Figure 9 depicts the real and predicted performance slowdowns for each benchmark. In most cases, the predicted performance degradation is close to the real one, with the prediction errors ranging from 0.0% to 8.6% with an average of 0.2%.

5.2.2 Multi-Threaded PARSEC Benchmarks

We also evaluate our approach using eight multi-threaded PARSEC benchmarks. Each benchmark is configured to have two threads, with one thread per dedicated core. In addition, the thread-to-core mappings are determined statically. We randomly generated 10 workloads from SPEC2006 and PARSEC benchmarks to co-run with each PARSEC benchmark, and the prediction

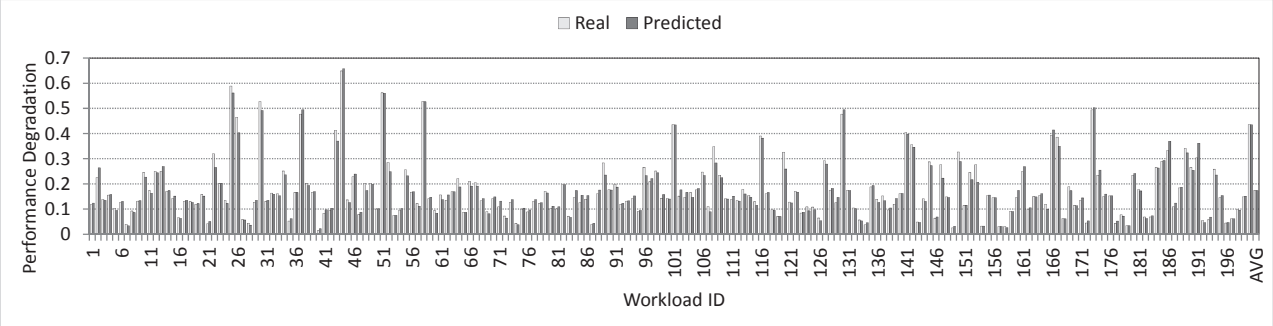


Fig. 9: Prediction precision for 200 randomly generated workloads from 18 SPEC2006 benchmarks. For each workload (with four benchmarks), the real and predicted performance slowdowns of a randomly picked representative are shown.

errors are shown in Figure 10. For each PARSEC benchmark depicted along the x-axis, each bar corresponds to one workload, representing the absolute value of the predicted value minus the real value. The absolute values of prediction errors are less than 2% in most cases, ranging from 0% to 5.5% with an average of 1.1%.

5.2.3 Datacenter Applications

Let us consider the nine datacenter applications available to us whose detailed descriptions are present in Appendix D, with five oriented to process large data and implemented using MapReduce. For each application, we randomly generated 15 workloads from these applications to co-run with it. Figure 11 depicts both the real and predicted performance slowdowns for each application. In most cases, the predicted performance degradation is close to the real one. The prediction errors range from 0.0% to 5% with an average of only 0.3%.

5.3 One More Platform

We discuss briefly our results on one more platform: a 2.13GHz octa-core Intel Xeon E7-8830, with a private 32KB L1 D-cache, a private 32KB L1 I-cache, a private 256KB L2 cache, a shared 24MB L3 cache and a memory bandwidth of 32GB/s.

With the same warehouse created earlier, we repeat the same steps discussed in Section 5.1 to build an abstract model for the new platform in the first phase. The training set obtained has 90 programs for the octa-core Xeon. The abstract model is instantiated similarly for each new program.

For the octa-core Xeon, there are 24 SPEC2006 benchmarks not included in its training set. We randomly generated 100 co-running workloads and randomly picked two representatives for each workload. Figure 12 depicts the prediction error distribution for the 200 representatives selected. The horizontal axis is prediction error while the vertical axis is cumulative distribution function. The prediction errors range from 0.0% to 5.54%, with an average of 0.79% while about 70% of representatives exhibit a prediction error less than 1%.

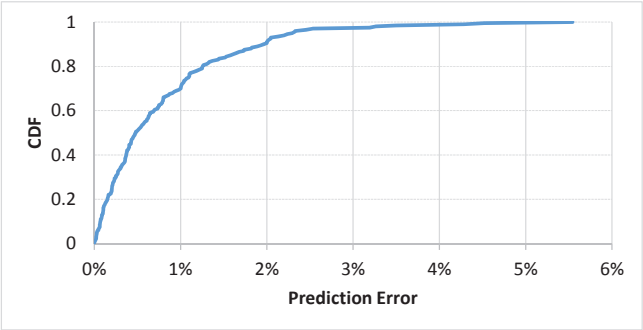


Fig. 12: Prediction error distribution for 100 randomly generated workloads from 24 SPEC2006 benchmarks on the octa-core Intel Xeon. For each workload, the prediction errors of two randomly picked representatives are included.

5.4 Evaluating Inter-Thread Contention for Multi-Threaded Applications

In this section, we first evaluate our model from the perspective of the prediction accuracy for accumulated CPU cycles, and then we take the cost of thread synchronization into consideration, demonstrate the predicted speedups varying with the number of threads using our inter-thread contention model (Section 5.4.1) and briefly discuss the data sharing of multi-threaded applications (Section 5.4.2).

The main platform for evaluating the multi-threaded applications is a 2.4GHz Intel six-core E5645 processor with a private 32KB L1 D-cache, a private 32KB L1 I-cache, a private 256KB L2 cache, a shared 12MB L3 cache and a memory bandwidth of 10.67GB/s. The reason that we did not select the Intel platform with 8 cores in Section 5.3 is that it has very large shared cache (24MB) and PARSEC benchmarks can not exhibit inter-thread contention at all. Take `facesim` for instance, the accumulated CPU cycles increase only 1.9% when we scale it to occupy all available cores of the octa-core Intel platform.

We use PARSEC[4] to evaluate our model for predicting inter-thread contention. We select three

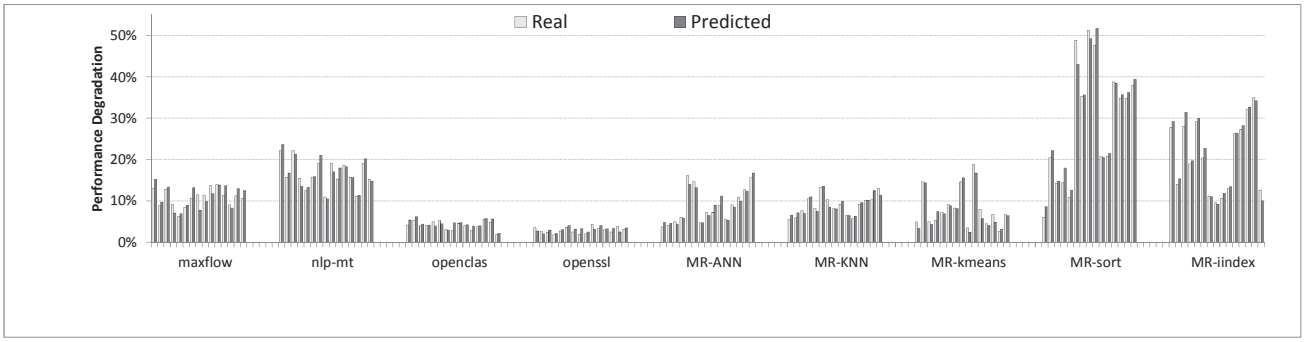


Fig. 11: Prediction precision for the applications listed in Appendix D, with each co-running in 15 workloads from Appendix D.

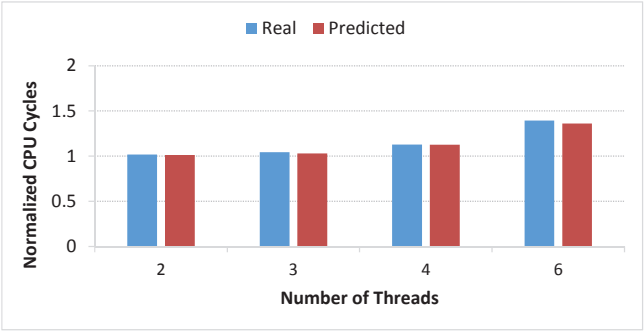


Fig. 13: Prediction precision of accumulated CPU cycles for `facesim`.

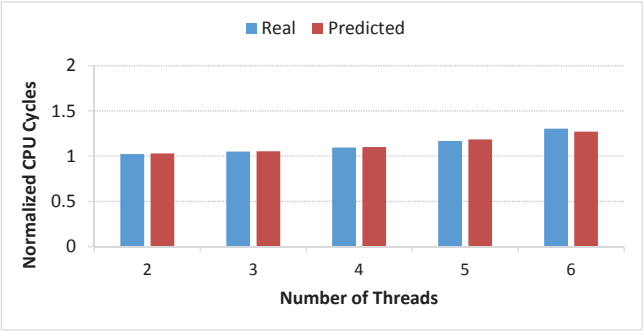


Fig. 14: Prediction precision of accumulated CPU cycles for `streamcluster`.

representative benchmarks for evaluation: `facesim`, `streamcluster`, `x264`. The first two are selected because they are memory-intensive and would introduce inter-thread contention when multiple threads are launched [46]. `x264` is selected as a representative pipelined application. All benchmarks are compiled using default `gcc-pthreads` configuration provided by PARSEC.

5.4.1 Prediction Accuracy of CPU Cycles

In this section, we present our evaluation results of CPU cycles and speedup for the three benchmarks, i.e., `facesim` and `streamcluster` for memory-intensive, and `x264` for pipelined application. We focus only on the parallel regions and ignore all sequential regions.

Memory-Intensive Applications. Figure 13 and Figure 14 depicts the prediction precision of accumulated CPU cycles for `facesim` and `streamcluster` respectively, with the horizontal axis representing the number of threads and the vertical axis representing the accumulated CPU cycles normalized to single-threaded execution. The blue bars represent real CPU cycles while the red bars are predicted values using our inter-thread contention model. For `facesim`, the results for 5 threads are not shown because the benchmark itself does not enable to launch 5 threads. Figure 13 and Figure 14 show that our predicted CPU cycles are very close to real ones.

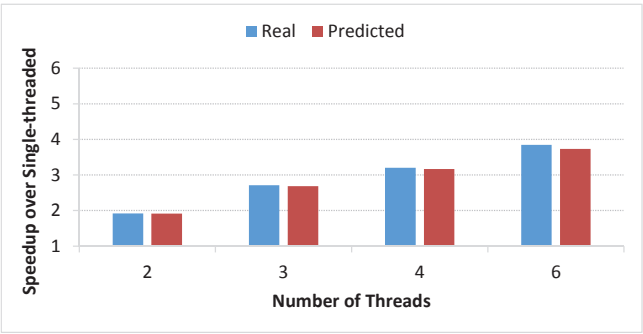


Fig. 15: Prediction precision of speedup over single-threaded for `facesim`.

Specifically, for `facesim` the average prediction error is only 1.3%. For `streamcluster`, the average prediction error is 1.2%.

For non-pipelined memory-intensive applications, our model achieve good accuracy when predicting accumulated CPU cycles. And we further use our model to predict the real execution time and predict the real scalability.

Figure 15 and Figure 16 present the predicted speedups in terms of execution time for `facesim` and `streamcluster`, with the horizontal axis representing the number of threads and the vertical axis representing

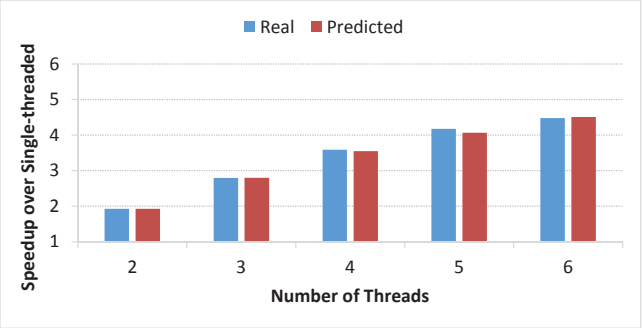


Fig. 16: Prediction precision of speedup over single-threaded for `streamcluster`.

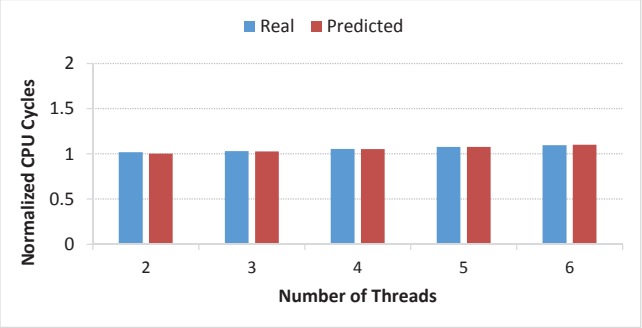


Fig. 17: Prediction precision of accumulated CPU cycles for `x264`.

the speedup in terms of execution time. The blue and red bars represent real and predicted values, respectively.

In particular, we compute the execution time of a multi-threaded application using the formula:

$$ExeTime = \frac{CPUTime + SyncTime}{n} \quad (9)$$

where

$$CPUTime = \frac{CPUCycles}{freq} \quad (10)$$

n is number of threads, $CPUCycles$ is accumulated CPU cycles predicted using our model, $freq$ is the frequency of the processor, and $SyncTime$ is time consumed for synchronization and is obtained via profiling. As shown by Figure 15 and Figure 16, our model can be used to precisely predict the scalability, with the average error of 1.3% for `facesim` and 0.9% for `streamcluster`.

In summary, since inter-thread contention can be accurately captured, our model can precisely predict the scalability for multi-threaded applications.

Pipelined Applications. Figure 17 shows the prediction precision of accumulated CPU cycles for `x264`, with an average error of 0.56%. Compare with the above two memory-intensive applications, `x264` suffers less from inter-thread contention. In particular, when the number of threads is set to 6, the CPU cycles for `facesim` and `streamcluster` are increased by 39% and 30% respectively, while only 10% for `x264`.

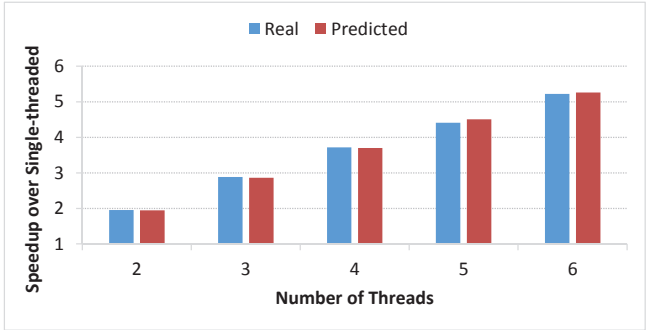


Fig. 18: Prediction precision of speedup over single-threaded for `x264`.

Figure 18 further depicts the prediction accuracy of speedup for `x264` in terms of the execution time, with an average error of 0.91%.

5.4.2 Discussion on Data Sharing

In this section, we briefly discuss data sharing and present some experimental results to demonstrate that on modern machines data sharing is not a performance-critical issue, using the PARSEC benchmarks and the six-core Intel platform.

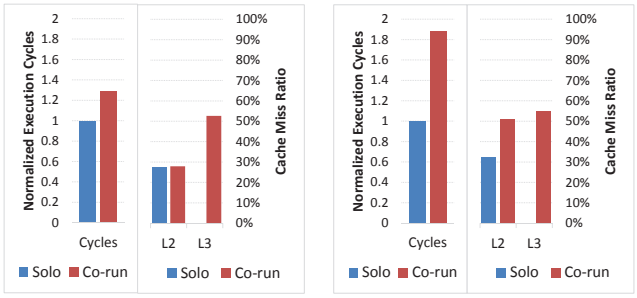
Data sharing can affect multi-threaded applications' performance both positively and negatively. On the one hand, data sharing can make applications use shared cache collaboratively, e.g. prefetch data constructively [52]. On the other hand, it introduces cache coherence traffic, causing increased cache access latency and reduced bandwidth to caches [33].

Thus, we focus on all the read requests to LLC, and compute the percentage of the "shared" cache line accesses (a shared cache line means that the line is located in more than one core's private caches) [52]. A larger value indicates higher data sharing. The values for three representative benchmarks `facesim`, `streamcluster`, `x264` are 0.9%, 4.5%, 0.5%, respectively when 6 threads are running. The results illustrate that effect of data sharing can be ignored.

5.5 Effects of Inclusive and Non-Inclusive Cache Designs on Performance Interference

Intel and AMD processor families have distinct cache sharing policies, inclusive and non-inclusive, respectively. We analyze how different cache designs affect the performance interference for co-located applications.

For processors with an inclusive cache hierarchy, inclusion victims are introduced which may exacerbate cache contention. Consider a multicore processor with an inclusive cache. When applications A and B are co-located on the same processor (different cores), there may be shared cache contention between A and B. Thus, for each application, say A, some data of its working set may be evicted from the shared cache before they are fully used. To maintain inclusion, these data have to



(a) AMD (Non-Inclusive Cache) (b) Intel (Inclusive Cache)

Fig. 19: Normalized increases of execution cycles, private L2 and shared L3 cache miss ratios for INTERFEREE when co-run with FLUSHER on two different platforms.

be purged from A's private cache. The resulting cache misses are known as *inclusion victims* [14].

5.5.1 Experimental Design

We use two quad-core platforms, the Intel platform used in Sections 5.1 and a 2.20GHz quad-core AMD Opteron 8374, with a private 64KB L1 D-cache, a private 64KB L1 I-cache, a private 512KB L2 cache, a shared 6MB L3 cache and a memory bandwidth of 12.8GB/s. In particular, the Intel platform has an inclusive LLC [33] and the AMD platform has an non-inclusive LLC [33]. We demonstrate how these two different cache sharing policies will affect performance interference with a synthesized STREAM kernel [31]. We refer to this kernel as INTERFEREE and analyze its performance degradation and behavior variations when it is co-located. INTERFEREE is designed to periodically generate private L2 cache miss and issue shared L3 cache access requests in solo execution. Thus, when co-located with the L3 cache being contended, INTERFEREE is expected to suffer from the inclusion-victim problem on the Intel platform. Therefore, we have designed the INTERFEREE with the following principles:

- Its working set (as an array) exceeds per-core's private cache size but not the shared cache size. This ensures that the data that are missed in the L2 cache can be found in the L3 cache. As a result, the working set of size $1.25\times$(L1's size + L2's size) is used for both the Intel and AMD platforms.
- The (working set) array is randomly accessed in order to periodically generate L2 cache misses. Meanwhile, the random access invalidates the hardware prefetcher. As a result, the cache misses represent the dominant factor for performance variations.
- The above access pattern is repeated for sufficiently many times to guarantee that the kernel runs for a long time and thus exhibits stable behaviors.

To generate cache contention with INTERFEREE, we have synthesized a cache flusher, called FLUSHER, as the co-runner, which continuously accesses a large array in a streaming mode. We co-run INTERFEREE with FLUSHER on the two platforms and analyze the behaviors of INTERFEREE with the help of PMUs.

5.5.2 Analysis

Figure 19 depicts the experimental results on the Intel and AMD platforms. For each platform, there are three groups of bars representing the normalized execution cycles, L2 cache miss ratios and L3 cache miss ratios obtained when INTERFEREE runs alone and together with FLUSHER. The execution cycles are normalized to solo execution (against the left y-axis). The L2 and L3 cache miss ratios are given against the right y-axis.

On the AMD platform, when INTERFEREE is co-running with FLUSHER, the L3 cache miss ratio increases dramatically, from 0% to 52.6%, due to shared cache contention caused by the cache flushing behavior of FLUSHER. Meanwhile, the L2 cache miss ratio remains unchanged, since the cache hierarchy is non-inclusive. Thus, the data in the L2 cache are not affected by the increased L3 cache misses, but the number of execution cycles has increased due to the increased L2 cache miss penalty.

In contrast, the situation is different on the Intel platform. As shown in Figure 19(b), the L3 cache miss ratio also increases dramatically, from 0% to 54.7%. However, due to inclusion victims, the L2 cache miss ratio also exhibits a significant increase, from 32.3% to 51.1%. As a result, the performance degradation of INTERFEREE is more significant on the Intel platform.

Our results show that inclusive caches can exacerbate cache contention for co-running applications. However, this does not mean that Intel processors will cause more severe performance interference than AMD processors. The performance interference for an application is affected by a number of architectural issues, e.g., cache associativity, cache replacement strategy, hardware prefetcher, memory controller. In this experiment, we have designed INTERFEREE to expose the effects of cache sharing policies while minimizing the influences of other issues on performance interference.

6 RELATED WORK

There has been a lot of work on addressing the contention for shared resources, especially shared cache, for multicore processors. *Cache partitioning* has been used to mitigate shared cache contention [38], [35], [36], [53], [17], [34], [6], [23], [44], [45]. For hardware solutions, cache resources are allocated to applications based on benefit rather than rate of demand [38], [35], [36], [56]. For software solutions, page coloring is used instead [6], [23], [53].

In the case of the contention for other shared resources such as memory bandwidth and on-chip interconnect, *contention-aware scheduling* represents a useful approach to mitigate the contention. By default, these shared resources are application-unaware, causing performance interference between co-running applications. The main intuition behind contention-aware scheduling is to classify *qualitatively* all applications into two categories depending on whether they consume shared resources

aggressively or not. With this classification, the scheduler can mix the applications from the two categories to mitigate resource contention when deciding which applications should be grouped together to run simultaneously on the same multicore processor [16], [19], [49], [47], [25], [26]. Furthermore, the scheduler can also adjust the resources allocated to an application to mitigate the contention for shared resources [56], [49], [11], [48], [42], [43].

There are also extensive studies on understanding and predicting shared cache contention on multicore processors. The best known techniques are Stack Distance Profiles (SDP) [30] and Miss Rate Curves (MRC) [30], [5], which shed light on an application's reuse behavior with its cache sharing.

These earlier techniques cannot be used directly to solve the datacenter co-location problem, which requires the ability of predicting *quantitatively* the performance interference between co-running applications on a multicore processor. Recently, Bubble-Up [25], [26], [42] predicts the performance degradation that results from contention for the shared memory subsystem. This is the closest related to our work. However, Bubble-Up is limited to predicting interference between two applications. SMiTe [54] is also proposed to predict performance interference caused by SMT co-location. Bandit [9] does not have this limitation but focuses only on bandwidth contention. In contrast, our two-phase approach applies to arbitrary co-running applications competing for multiple shared resources. In addition, this paper represents the first to use a piecewise predictor function.

Finally, once performance interference is predicted, applications can be mapped to multicores under some scheduling policies and optimization goals [15], [56]. Bubble-flux [50] probes and schedules low-priority applications dynamically to guarantee the QoS of high-priority applications and improve the overall utilization of datacenters. The performance interference model can be leveraged by the compiler to include some co-runner-aware code transformations and optimizations [2], [42]. To enable low overhead online code transformation, Protean Code[20] is also developed. Furthermore, some domain-specific optimizations [8], [7] can be applied to some datacenter applications to make them co-locate better. Besides, as modern datacenters are more considered as heterogeneous, identify and utilize the performance opportunity behind the heterogeneity is also been researched[28].

7 Conclusion and Future Work

In this paper, we present a two-phase regression approach to predicting the performance interference between multiple applications co-running on a multicore processor. We have experimentally validated the existence of a piecewise function between the aggregate shared resource consumptions and the performance degradation of an application when co-located. By proceeding in two phases rather than one, we can obtain a predictor function scalably. Furthermore, the prediction model obtained for an application is able to characterize its contentiousness and sensitivity as well.

In future work, we plan to generalize our model so that more multi-threaded programs can be handled. Presently, our model works for a multi-threaded program if the program has a fixed number of threads, statically grouped, so that the threads in a group always run together on a processor, with at most one thread per core. In addition, we also plan to study how to consider positive interactions between applications, e.g., those caused by applications sharing the same OS utility. Finally, it would be interesting to investigate how to predict performance interference by combining static program analysis, particularly pointer analysis [37], [39], [51] with profiling.

Acknowledgements

This research is supported in part by the National High Technology Research and Development Program of China (2012AA010902, 2015AA011505), the National Natural Science Foundation of China (61202055, 61221062, 61303053, 61432016, 61402445), the National Basic Research Program of China (2011CB302504), and Australian Research Council (ARC) Grants (DP110104628 and DP130101970). We would like to thank all the reviewers for their valuable comments and suggestions.

References

[1] Alaa R Alameldeen and David A Wood. Ipc considered harmful for multiprocessor workloads. *IEEE Micro*, 26(4):8–17, 2006.

[2] B. Bao and C. Ding. Defensive loop tiling for shared cache. In *CGO*, 2013.

[3] L. A. Barroso and U. Holzle. The case for energy-proportional computing. In *IEEE Computer*, 2007.

[4] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[5] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *HPCA*, 2005.

[6] S. Cho and L. Jin. Managing distributed, shared L2 caches through OS-level page allocation. In *MICRO*, 2006.

[7] H. Cui, L. Wang, J. Xue, Y. Yang, and X. Feng. Automatic library generation for BLAS3 on GPUs. In *IPDPS*, 2011.

[8] H. Cui, J. Xue, L. Wang, Y. Yang, X. Feng, and D. Fan. Extendable pattern-oriented optimization directives. In *CGO*, 2011.

[9] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten. Bandwidth Bandit: Quantitative characterization of memory contention. In *CGO*, 2013.

[10] Donald E Farrar and Robert R Glauber. Multicollinearity in regression analysis: The problem revisited. *The Review of Economics and Statistics*, 49(1):92–107, 1967.

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

[12] Yuxiong He, Charles E Leiserson, and William M Leiserson. The cilkview scalability analyzer. In *SPAA'10*, 2010.

[13] Intel. Intel performance tuning utility. http://software.intel.com/en-us/articles/intel-performance-tuning-utility.

[14] Aamer Jaleel, Eric Borch, Malini Bhandaru, SC Steely, and Joel Emer. Achieving non-inclusive cache performance with inclusive caches: Temporal locality aware (tla) cache management policies. In *Micro-43*, pages 151–162. IEEE, 2010.

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

[16] Y. Jiang, K. Tian, and X. Shen. Combining locality analysis with online proactive job co-scheduling in chip multiprocessors. In *HiPEAC*, 2010.

[17] S. Jim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *PACT*, 2004.

[18] Ian T Jolliffe. Principal components in regression analysis. *Principal Component Analysis*, pages 167–198, 2002.

[19] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using OS observations to improve performance in multicore systems. In *Micro*, 2008.

[20] Michael Laurenzano, Yunqi Zhang, Lingjia Tang, and Jason Mars. Protean code: Achieving near-free online code transformations for warehouse scale computers. In *Micro-47*. ACM, 2014.

[21] Janghaeng Lee, Haicheng Wu, Madhumitha Ravichandran, and Nathan Clark. Thread tailor: Dynamically weaving threads together for efficient, adaptive parallel applications. In *ISCA '10*, pages 270–279. ACM, 2010.

[22] Kevin M Lepak, Harold W Cain, and Mikko H Lipasti. Redeeming ipc as a performance metric for multithreaded programs. In *PACT 2003*, pages 232–243. IEEE, 2003.

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

[24] J. Machina and A. Sodan. Predicting cache needs and cache sensitivity for applications in cloud computing on CMP servers with configurable caches. In *IPDPS*, 2009.

[25] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations. In *MICRO*, 2011.

[26] J. Mars, L. Tang, and M. L. Soffa. Directly characterizing cross-core interference through contention synthesis. In *HiPEAC*, 2011.

[27] J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. Contention aware execution: online contention detection and response. In *CGO*, 2010.

[28] Jason Mars and Lingjia Tang. Whare-map: Heterogeneity in "homogeneous" warehouse-scale computers. In *ISCA*, 2013.

[29] José F. Martínez and Josep Torrellas. Speculative synchronization: Applying thread-level speculation to explicitly parallel applications. In *ASPLOS X*, pages 18–29. ACM, 2002.

[30] R. L. Matterson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. In *IBM Systems Journal 9*, 1970.

[31] John D McCalpin. Stream: Sustainable memory bandwidth in high performance computers, 1995.

[32] Maged M Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE TPDS*, 15(6):491–504, 2004.

[33] Daniel Molka, Daniel Hackenberg, Robert Schöne, and Matthias S. Müller. Memory performance and cache coherency effects on an intel nehalem multiprocessor system. In *PACT*, 2009.

[34] K. Nesbit, M. Moreto, F. Cazorla, A. Ramirez, M. Valero, and J. Smith. Multicore resource management. In *MICRO*, 2008.

[35] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A case for MLP-aware cache replacement. In *ISCA*, 2006.

[36] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO*, 2006.

[37] L. Shang, X. Xie, and J. Xue. On-demand dynamic summary-based points-to analysis. In *CGO*, pages 264–274, 2012.

[38] S. Srikantaiah, R. Das, A. K. Mishra, C. R. Das, and M. Kandemir. A case for integrated processor-cache partitioning in chip multiprocessors. In *SC*, 2009.

[39] Y. Sui, Y. Li, and J. Xue. Query-directed adaptive heap cloning for optimizing compilers. In *CGO*, pages 1–11, 2013.

[40] M. Aater Suleman, Moinuddin K. Qureshi, and Yale N. Patt. Feedback-driven threading: Power-efficient and high-performance execution of multi-threaded workloads on cmps. In *ASPLOS*, 2008.

[41] L. Tang, J. Mars, and M. L. Soffa. Contentiousness vs. sensitivity: improving contention aware runtime systems on multicore architectures. In *EXADAPT*, 2011.

[42] L. Tang, J. Mars, and M. L. Soffa. Compiling For Niceness Mitigating Contention for QoS in Warehouse Scale Computers. In *CGO*, 2012.

[43] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. The Impact of Memory Subsystem Resource Sharing on Datacenter Applications. In *ISCA*, 2011.

[44] X. Vera, B. Lisper, and J. Xue. Data caches in multitasking hard real-time systems. In *RTSS*, pages 154–165, 2003.

[45] X. Vera, B. Lisper, and J. Xue. Data cache locking for tight timing calculations. *ACM TECS*, 7(1), 2007.

[46] Wei Wang, Tanima Dey, Jack W Davidson, and Mary Lou Soffa. Dramon: Predicting memory bandwidth usage of multi-threaded programs with high accuracy and low overhead. In *HPCA*, 2014.

[47] Y. Xie and G. Loh. PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches. In *ISCA*, 2009.

[48] D. Xu, C. Wu, P. Yew, J. Li, and Z. Wang. Providing Fairness on Shared-Memory Multiprocessors via Process Scheduling. In *SIGMETRICS*, 2012.

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

[50] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *ISCA '13*, 2013.

[51] H. Yu, J. Xue, W. Huo, X. Feng, and Z. Zhang. Level by level: making flow- and context-sensitive pointer analysis scalable for millions of lines of code. In *CGO*, pages 218–229, 2010.

[52] Eddy Z. Zhang, Yunlian Jiang, and Xipeng Shen. Does cache sharing on modern cmp matter to the performance of contemporary multithreaded programs? In *PPoPP '10*. ACM, 2010.

[53] X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multicore cache management. In *EuroSys*, 2009.

[54] Yunqi Zhang, Michael Laurenzano, Jason Mars, and Lingjia Tang. Smite: Precise qos prediction on real system smt processors to improve utilization in warehouse scale computers. In *MICRO-47*, New York, NY, USA, 2014. ACM.

[55] Jiacheng Zhao, Huimin Cui, Jingling Xue, Xiaobing Feng, Youliang Yan, and Wensen Yang. An empirical model for predicting cross-core performance interference on multicore processors. In *PACT*, pages 201–212. IEEE Press, 2013.

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

Jiacheng Zhao received the BS degree in computer science from Tianjin University, China, in 2012. He is currently working toward the PhD degree from the Institute of Computing Technology, Chinese Academy of Sciences. His research interests include parallel computing and parallel compiling.

Huimin Cui received the BS degree and MS degrees in computer science from Tsinghua University, China in 2001 and 2004, respectively. She received the PhD degree from the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, China, in 2012. She is currently an associate professor at ICT, CAS. Her research interests include compiler optimizations, programming languages and programming environments.

Jingling Xue received the BSc and MSc degrees in Computer Science from Tsinghua University, China in 1984 and 1987, respectively. He received the PhD degree in Computer Science from Edinburgh, United Kingdom, in 1992. He is currently a Professor in the School of Computer Science and Engineering at the University of New South Wales. His current research interests include programming languages, compiler optimisations, program analysis, high-performance computing and embedded systems. He is a senior member of IEEE and a member of ACM.

Xiaobing Feng received the BS degree in computer science from Tianjin University, China, in 1992. He received the MS degree in computer science from Peking University, China, in 1996. He received the PhD degree from the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, China, in 1999. He is currently a professor at ICT, CAS. His research interests include compiler optimizations and binary translation.