An Empirical Model for Predicting Cross-Core Performance Interference on Multicore Processors

Jiacheng Zhao*†
*Institute of Computing Technology, Chinese Academy of Sciences
†University of Chinese Academy of Sciences
Beijing, China
zhaojiacheng@ict.ac.cn

Huimin Cui
SKL Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
cuihm@ict.ac.cn

Jingling Xue
School of Computer Science and Engineering, University of New South Wales
Sydney, NSW 2052, Australia
jingling@cse.unsw.edu.au

Xiaobing Feng
SKL Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
fxb@ict.ac.cn

Youliang Yan
Shannon Laboratory, Huawei Technologies Co., Ltd.
Shenzhen, China
yanyouliang@huawei.com

Wensen Yang
Shannon Laboratory, Huawei Technologies Co., Ltd.
Shenzhen, China
yangwensen@huawei.com

***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.**

This research is driven by two key insights. First, the performance degradation of an application 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 function is piecewise rather than non-piecewise as in prior work, thereby enabling different types of dominant contention factors to be more accurately captured by different subfunctions in its different subdomains. Based on these insights, we propose to adopt a two-phase regression approach to efficiently building a predictor function. Validation using a large number of benchmarks and nine real-world datacenter applications on three different platforms shows that our approach is also precise, with an average error not exceeding 0.4%. When applied to the nine datacenter applications, our approach improves overall resource utilization from 50% to 88% at the cost of 10% QoS degradation.

***Keywords*-cross-core performance interference, memory subsystems, multicore processors, performance analysis, prediction model.**

I. Introduction

Two significant trends are emerging to dominate the landscape of computing today: multicore processors and cloud computing [31]. The microprocessor industry is rapidly moving towards multi/many-core architectures that integrate tens or even hundreds of cores onto a single chip. Meanwhile, much of the world's computing continues to move into the cloud. Due to a synergy of the two trends, multicore processors have been widely adopted in cloud computing.

A datacenter houses large-scale web applications and cloud services. However, the utilization of its computing resources is very low, i.e., around 20% [2], [24]. 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 [12], [24], [36], [37], [42], [45]. Unfortunately, such performance interference impacts negatively and unpredictably the quality of service (QoS) of some user-facing and latency-sensitive applications. As a result, co-location for such applications is disallowed, leaving many cores idle and causing low machine utilization [24]. 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 [4], [17], [19], [23], [25], [26] 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 [24] 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 [11] does not have this limitation but focuses on bandwidth contention only.

In this paper, we introduce an empirical approach to efficiently and precisely predicting the performance degradation suffered by an application due to arbitrarily many co-located applications. Our two key insights are:

- *The performance degradation of an application 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, and*
- *A predictor function is piecewise rather than non-piecewise as in prior work so that different dominant contention factors can be accommodated more accurately with different subfunctions in its different subdomains.*

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 degradation 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 instantiates the abstract model for a given application by co-running it with a small number of training workloads to determine the application-specific coefficients for the functional form obtained earlier. The instantiated model for the given application can then be used to predict its performance degradation with any co-runners. Therefore, 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 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 propose to characterize the performance degradation of an application due to co-location with a piecewise predictor function, which admits different subfunctions depending which contention factors are dominant.
- 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 benchmarks and nine real-world datacenter applications on three platforms.
- Our prediction model can be used in a datacenter to increase its overall resource utilization. For the nine applications tested, the resource utilization has gone up from 50% to 88% at the cost of 10% QoS degradation.

The rest of the paper is organized as follows. Section II motivates this work. Section III presents our two-phase approach. Section IV describes our experimental validation. Section V discusses the related work. Section VI concludes.

II. Motivation

We first present some experimental results to demonstrate performance interference resulting from co-locations (Section II-A). We then discuss our key insights that motivated the design of our prediction model, and subsequently, the development of a two-phase approach to building it efficiently (Section II-B). Finally, we use a benchmark to highlight the intuition behind the precision of our prediction (Section II-C).

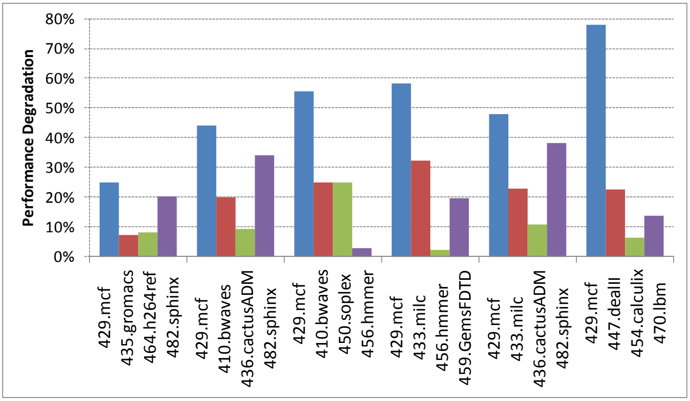

Fig. 1: Performance slowdowns of `429.mcf` when co-running some SPEC2006 benchmarks on a quad-core Xeon.

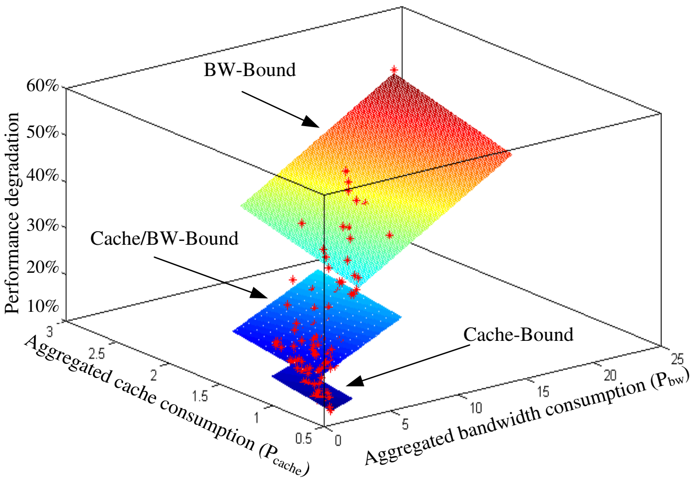

Fig. 2: Existence of a piecewise function relating the performance degradation of `429.mcf` to the aggregate pressures on shared cache and memory bandwidth. The three planes are the plots of the three subfunctions given in (3).

A. Performance Interference

Figure 1 shows the performance interference between some co-running SPEC2006 benchmarks on an Intel quad-core Xeon specified in the first paragraph of Section IV. The horizontal axis shows six groups of co-runners representing six different workloads for `429.mcf`. The vertical axis represents the performance degradation of a benchmark computed by:

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

where $ExeTime_{solo}$ ($ExeTime_{co-run}$) is the execution time in solo (co-running) execution. Note that when co-running with different workloads, a benchmark can experience a wide variation in performance degradation. For example, `429.mcf` suffers from a slowdown ranging from 24.9% to 78%.

B. Our Insights

There are two key insights. 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, e.g., shared cache and memory bandwidth (BW) 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.

Consider two typical scenarios. Given a cache-intensive application C, its performance worsens rapidly if its co-runners compete severely with it for shared cache. However, as the shared cache contention saturates, the contention shifts to memory bandwidth, causing C's performance to degrade more slowly. Given a bandwidth-intensive application B, its performance worsens slowly if its co-runners' bandwidth consumptions are low. However, as the total bandwidth consumption increases, the bandwidth contention grows, causing B's performance to degrade more quickly. In this case, the performance degradation of an application can be predicted by a *piecewise* function consisting of three subfunctions to account for three different dominant contention factors: cache-bound, cache/BW-bound, and BW-bound contention.

C. Our Prediction Model

We use `429.mcf` from SPEC2006 to introduce the key elements involved in building a prediction model for the performance degradation of an application. This sheds some light on the precision behind our prediction. However, the approach discussed here is brute-force and thus impractical for a datacenter housing a large number of applications. In Section III, we introduce our scalable two-phase approach.

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`.

1) 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:

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

2) 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 $((P_{cache}, P_{bw}), PD_w)$.

3) 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 III), we obtain the following piecewise function:

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

where

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

The R-squared value for PD_{mcf} is 0.90, indicating a strong fit. Figure 2 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.

III. Two-Phase Regression Approach

Our insights lead to the design of a precise prediction model. However, a datacenter may house hundreds to thousands of applications with the frequent development and updating of these applications. Repeating the same regression analysis as above 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. Based on this, we introduce a two-phase regression approach, as shown in Figure 3, to build a predictor function efficiently for a given 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 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.

In this paper, we focus on the contention for shared cache and memory bandwidth, as this is the dominant contention in many applications. Our approach is expected to be applicable to other shared resources as well. Section III-A describes how to build an abstract prediction model. Section III-B describes how to instantiate the abstract model for a given application. Section III-C analyzes the efficiency of our approach, which will be experimentally validated in Section IV-C.

A. 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

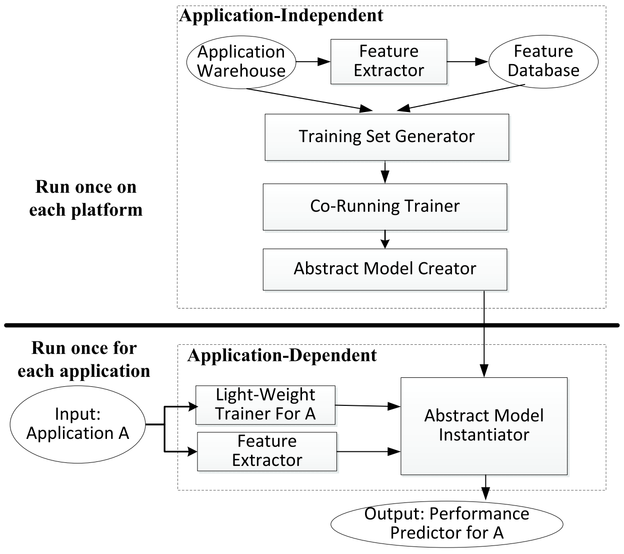

Fig. 3: Two-phase approach for predicting the performance degradation of an application on a computer platform.

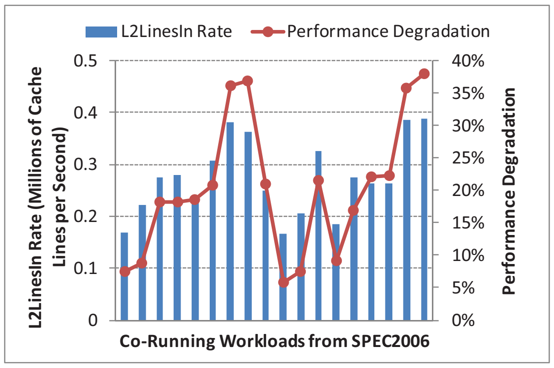

Fig. 4: Performance degradation of `429.mcf` varies with accumulated L2 LinesIn, indicating that the degradation has a good correlation with accumulated L2_LinesIn.

storing these as a feature vector in the "Feature Database" (Section III-A1). 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 III-A2). The "Co-Running Trainer" records the performance slowdowns of all training workloads (Section III-A3). The "Abstract Model Creator" builds an abstract prediction model via regression analysis (Section III-A4).

1) The Feature Extractor: We make use of PMUs to obtain the individual consumptions of shared cache and bandwidth from each application A_i in solo execution and represent the individual pressures as a feature vector of the form $FV(A_i) = (cache_i, bw_i)$. The critical issue here is to identify appropriate PMUs to use. For illustration purposes, we continue to consider the Intel platform described in Section II-A, with private L1 and L2 caches and a shared L3 cache.

To measure the shared cache (L3) consumption, *L2_LinesIn rate* is recommended [35], [37], 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. Figure 4 confirms a good correlation between the accumulated *L2_LinesIn* and the performance degradation of `429.mcf`. Each bar represents the accumulated *L2_LinesIn* for a workload (against the left y-axis) and the (red) line shows the performance degradation of `429.mcf` subject to the corresponding workload (against the right y-axis). The good correlation illustrates that *L2_LinesIn* is a good indicator for shared cache consumption.

An application's bandwidth consumption can be profiled using Intel's PTU (Performance Tuning Utility) [15], 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.

2) 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.

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) 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 I, 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.

4) 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

TABLE I: 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	$W_{A_2,Q}$	$PD_{A_2,W_{A_2,Q}}$
	...	

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.

#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. 5: Syntax of a configuration file.

Configurations: Figure 5 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 resource 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 $(P_{bw}, \textit{equal}(4))$. The user can leverage some empirical knowledge to perform this task. In particular, Tang *et al.* [35] observed that contention for bandwidth has a more dominating effect on performance interference than for other shared resources and Xu *et al.* [42] 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 I 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

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

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.

B. Phase 2: Instantiating the Abstract Model

All the components of this second phase are shown in the bottom part of Figure 3. For a given application A, we instantiate the abstract model obtained earlier by determining its application-specific coefficients. There are two cases. 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 four steps as described below:

- **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 III-A4, we perform a regression analysis to determine A's coefficients, and finally, 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.

C. Why Two Phases Instead of Just One Phase?

Our approach consists of two phases when building a predictor function for an application A. In the first application-independent phase, the piecewise functional relation, i.e., functional form shared by all applications is found. In the second phase, the application-specific coefficients for A are determined. In a brute-force approach that combines the two phases, A must be co-run with a *large* number of training workloads.

By comparing the two approaches, our approach is more efficient while equally being precise (validated in Section IV-C).

1) The Brute-Force Approach: To build a predictor function for A, let us suppose that A is to be co-run with Q workloads to obtain Q data points for regression analysis. Then the dominant cost of building a predictor function for A is:

$$COST_{\mathrm{bf}} = \sum_{i=1}^{Q} wkld_i^{\mathrm{bf}} \tag{7}$$

where $wkld_i^{\mathrm{bf}}$ is the execution time of the i-th workload. The time spent on regression analysis is negligible.

2) The Two-Phase Approach: Let P be the size of our training set and m be the number of cores in a multicore processor. As each application is restricted to appear only in Q workloads, $\frac{Q}{C_{P-1}^{m-1}} \times C_P^m = Q \times P/m$ training runs are required. So the dominant cost of the first phase is:

$$COST_{\mathrm{phase1}} = \sum_{i=1}^{Q \times P/m} wkld_i^{\mathrm{phase1}} \tag{8}$$

where $wkld_i^{\mathrm{phase1}}$ is the execution time of the i-th workload.

To determine the application-specific coefficients for A, $C \times S$ workloads are co-run with A as discussed in Section III-B. So the dominant cost of the second phase is:

$$COST_{\mathrm{phase2}} = \sum_{i=1}^{C \times S} wkld_i^{\mathrm{phase2}} \tag{9}$$

where $wkld_i^{\mathrm{phase2}}$ is the execution time of the i-th workload.

3) Discussion: If we have N applications to predict, both approaches have the following time complexities:

$$\begin{aligned} COST_{\mathrm{brute-force}} &= N \textstyle\sum_{i=1}^{Q} wkld_i^{\mathrm{bf}} \\ COST_{\mathrm{two-phase}} &= \sum_{i=1}^{Q \times P/m} wkld_i^{\mathrm{phase1}} + N \sum_{i=1}^{C \times S} wkld_i^{\mathrm{phase2}} \end{aligned} \tag{10}$$

In our approach, the first phase is performed only once for a platform. (A re-run is warranted only some major software upgrades occur.) When N is large, the underlying cost is amortized. In general, $C \times S \ll Q$ holds. So our approach is much more efficient in practice, as evaluated below.

IV. 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 two platforms are a six-core Intel Xeon E7-4807 and a quad-core AMD Opteron 8374.

A. Methodology and Benchmark

We built a warehouse including the benchmarks from SPEC2000, SPEC2006, LINPACK, MiBench [14], PARSEC [3], and Graph 500 and the nine real-world datacenter programs listed in Table II with a total of 506 applications. All are compiled using "GCC -O3" under Linux (kernel 2.6.18).

In the first phase, we created a training set as Section III-A2. 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 III-A3. With the 30 applications in the training set, we ran $30 \times 200/4(\text{cores}) = 1500$ workloads. In the configuration file given in Figure 5, "pre-processing" is {*none*, *exp(2)*, *log(2)*, *pow(2)*}, "mode" is {*add*, *mul*}, "domain partitioning" is $\{((P_{bw}), \textit{equal}(4)), ((P_{cache}), \textit{equal}(4)), ((P_{cache}, P_{bw}), \textit{equal}(4,4))\}$, 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:

$$PD = \begin{cases} a11P_{cache} + a12P_{bw} + a13 & \text{if } P_{bw} < 3.2 \\ a21P_{cache} + a22P_{bw} + a23 & \text{if } 3.2 \le P_{bw} \le 9.6 \\ a31P_{cache} + a32P_{bw} + a33 & \text{if } P_{bw} > 9.6 \end{cases} \tag{11}$$

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.

Benchmark	Predictor Function	
400.perlbench	$0.108*P_{bw}+0.484*P_{cache}+0.003$	$(P_{bw} < 3.2)$
	$0.115*P_{bw}+0.460*P_{cache}+0.001$	$(3.2 <= P_{bw} <= 9.6)$
	$0.176*P_{bw}+0.336*P_{cache}-0.026$	$(P_{bw} > 9.6)$
401.bzip2	$0.422*P_{bw}+1.337*P_{cache}-0.007$	$(P_{bw} < 3.2)$
	$0.438*P_{bw}+0.714*P_{cache}+0.018$	$(3.2 <= P_{bw} <= 9.6)$
	$0.445*P_{bw}+1.240*P_{cache}-0.046$	$(P_{bw} > 9.6)$
433.milc	--	$(P_{bw} < 3.2)$
	$0.403*P_{bw}+0.752*P_{cache}-0.154$	$(3.2 <= P_{bw} <= 9.6)$
	$0.935*P_{bw}+1.124*P_{cache}-0.719$	$(P_{bw} > 9.6)$
435.gromacs	$0.093*P_{bw}+0.430*P_{cache}-0.015$	$(P_{bw} < 3.2)$
	$0.129*P_{bw}+0.405*P_{cache}-0.028$	$(3.2 <= P_{bw} <= 9.6)$
	$0.154*P_{bw}+0.297*P_{cache}-0.033$	$(P_{bw} > 9.6)$
471.omnetpp	$0.355*P_{bw}+2.044*P_{cache}-0.080$	$(P_{bw} < 3.2)$
	$0.648*P_{bw}+1.280*P_{cache}-0.126$	$(3.2 <= P_{bw} <= 9.6)$
	$0.843*P_{bw}+1.012*P_{cache}-0.222$	$(P_{bw} > 9.6)$
445.gobmk	$0.141*P_{bw}+0.519*P_{cache}-0.006$	$(P_{bw} < 3.2)$
	$0.164*P_{bw}+0.459*P_{cache}-0.015$	$(3.2 <= P_{bw} <= 9.6)$
	$0.175*P_{bw}+0.490*P_{cache}-0.033$	$(P_{bw} > 9.6)$
447.dealII	$0.102*P_{bw}+0.451*P_{cache}$	$(P_{bw} < 3.2)$
	$0.166*P_{bw}+0.491*P_{cache}-0.028$	$(3.2 <= P_{bw} <= 9.6)$
	$0.283*P_{bw}+0.922*P_{cache}-0.187$	$(P_{bw} > 9.6)$
454.calcuix	$0.040*P_{bw}+0.219*P_{cache}-0.006$	$(P_{bw} < 3.2)$
	$0.048*P_{bw}+0.159*P_{cache}-0.005$	$(3.2 <= P_{bw} <= 9.6)$
	$0.066*P_{bw}+0.127*P_{cache}-0.020$	$(P_{bw} > 9.6)$

Fig. 6: Predictor functions for eight SPEC2006 benchmarks.

In the second phase, we follow the four steps described in Section III-B 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. Figure 6 gives the predictor functions for eight SPEC2006 benchmarks. For `433.milc`, the subfunction under $P_{bw} < 3.2$ does not exist, because its own bandwidth consumption is larger than 3.2GB/s.

B. Prediction Precision

We show that our predictors (with some given in Figure 6) are precise for both benchmarks and real-world applications.

1) 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 7 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%.

2) Datacenter Applications: Let us now consider the nine datacenter applications available to us as listed in Table II, 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 8 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%.

TABLE II: Datacenter applications.

Application	**Description**	**QoS Metric**
openssl	A secure sockets layer performance stress test.	user time (seconds)
openclas	A lexical analyzer to segment sentences into words with tags.	queries per second
nlp-mt	A language translator, with a similar functionality as google translate.	user time (seconds)
maxflow	A maximum flow algorithm, widely used in social networks.	user time (seconds)
MR-ANN	An artificial neural network algorithm, used to infer a function from observation, implemented using MapReduce.	throughput
MR-KNN	A K-nearest neighbor algorithm, a type of instance-based learning, implemented using MapReduce.	throughput
MR-kmeans	A K-means clustering algorithm, one of the most commonly-used algorithms in data mining, implemented using MapReduce.	throughput
MR-sort	A sorting algorithm, implemented using MapReduce.	throughput
MR-iindex	An inverted index algorithm used in text searches, with an index data structure used to store a mapping from content to its locations, implemented using MapReduce.	throughput

3) Prediction Error Analysis: While exhibiting a small average prediction error, our approach can make relatively large errors for some applications. In Figure 7, the two worst cases are Workload 108 (for `450.soplex` co-running with `456.hmmer`, `410.bwaves` and `459.GemsFDTD`) with an error of 8.6% and Workload 191 (for `473.astar` co-running with `450.soplex`, `433.milc` and `410.bwaves`) with an error of 7.3%.

There are two main sources of errors. First, our computations for shared resource consumptions are only estimates. We use the PMU for *L2_LinesIn* to estimate the shared cache consumed by an application. While reflecting the number of cache lines fetched from the shared L3 cache to the L2 cache, this metric cannot precisely determine the footprint of an application in the L3 cache, because it is not precisely related to the amount of data reuse in the application. For example, a cache line is fetched twice into the L2 cache due to capacity misses, *L2_LinesIn* counts it twice. However, this indicates only one cache line in the L3 cache.

Second, we use the average consumption for a shared resource by an application during its entire execution to obtain its feature vector. The phase behavior of an application cannot be accurately taken into account. Figure 9 shows the bandwidth consumption of `450.soplex` with a 1-second sampling interval. For an application, when its co-runners reach their peaks (troughs) simultaneously in terms of bandwidth consumption, the performance degradation would be larger (smaller) than predicted. For this main reason, the prediction

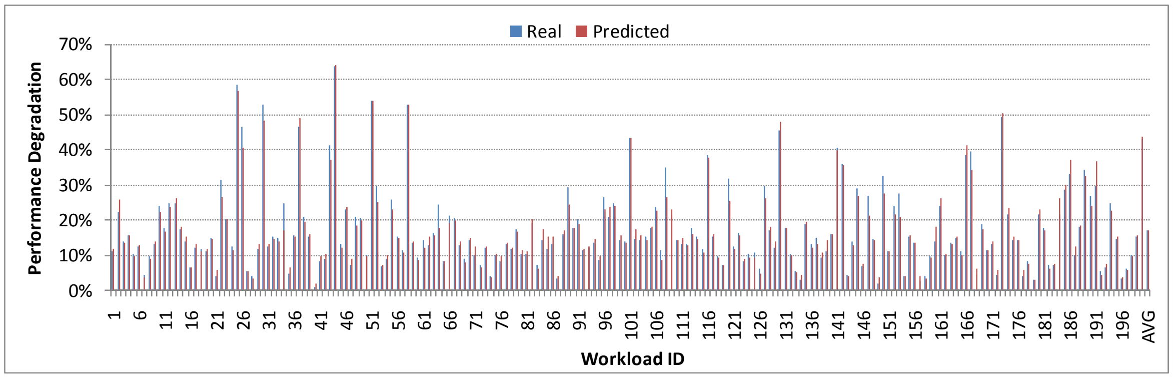

Fig. 7: Prediction precision for 200 randomly generated workloads from 18 SPEC2006 benchmarks. For each workload (with four benchmarks), the real and predicted performance slowdowns of a randomly picked representative are shown.

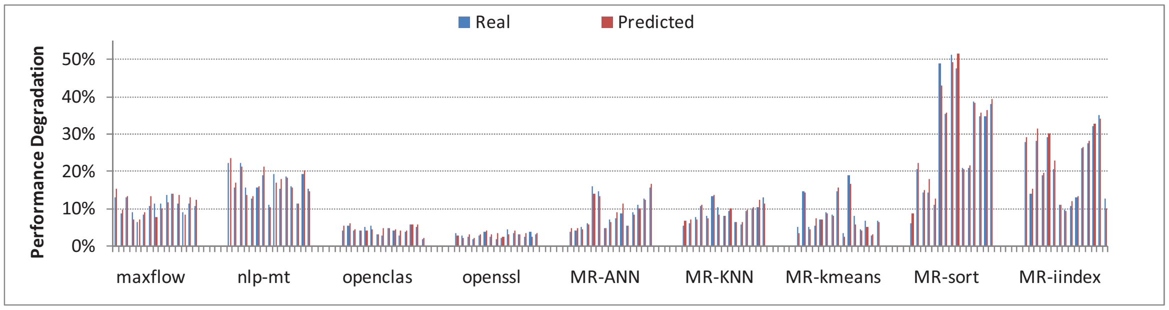

Fig. 8: Prediction precision for the applications listed in Table II, with each co-running in 15 workloads from Table II.

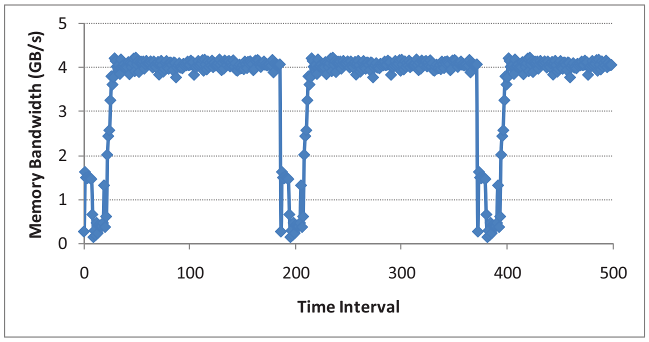

Fig. 9: The phase behavior of the bandwidth consumption for `450.soplex` with a sampling interval of 1 second.

error for `450.soplex` is 8.6% (Workload 108) as shown in Figure 7. Similarly, our approach has the errors mentioned above for Workload 191 in Figure 7 and Workloads 138 and 139 in Figure 10. Some of their co-runners also exhibit phase behavior, including `433.milc` between 3GB/s to 5.4GB/s and `483.xalancbmk` between 0GB/s to 2GB/s.

C. Prediction Efficiency

We provide experimental results to back up the analysis given in Section III-C. We show that our two-phase approach is scalable but the brute-force one-phase approach is not, and in addition, our approach is equally precise.

1) Comparing Efficiency: For our warehouse on Xeon, $m = 4$, $P = 30$, $Q = 200$, $C = 4$ and $S = 3$. A workload takes 15 minutes on average to finish, which is common in practice. According to (10), the costs (in hours) incurred by both approaches in making predictions for N applications are:

$$\begin{aligned} COST_{\text{brute-force}} &= 50N \\ COST_{\text{two-phase}} &= 375 + 3N \end{aligned} \tag{12}$$

$COST_{\text{brute-force}} > COST_{\text{two-phase}}$ when $N \geq 8$. Furthermore, the first phase in our approach takes 375 hours, i.e., about two weeks to build the abstract model for the Xeon platform, but this cost, which is independent of N, can be amortized as our second phase is applied to more and more applications (i.e., as N increases). Our second phase takes three hours for each application. In contrast, the brute-force approach takes more than two days for each application. If $N = 100$, the brute-force approach spends nearly 30 weeks on the N applications while our approach takes only 4 weeks.

2) Comparing Precision: Our two-phase approach is not only scalable but also as precise as the brute-force approach, as shown in Figure 10. We have randomly generated 150 workloads from SPEC2006 to co-run with `471.omnetpp` and compared both in terms of prediction precision. The prediction errors by brute-force range from 0.0% to 10.1% with an average of 0.23%. The prediction errors with two phases range from 0.0% to 11.7% with an average of 0.40%, which is slightly higher than that of the brute-force approach.

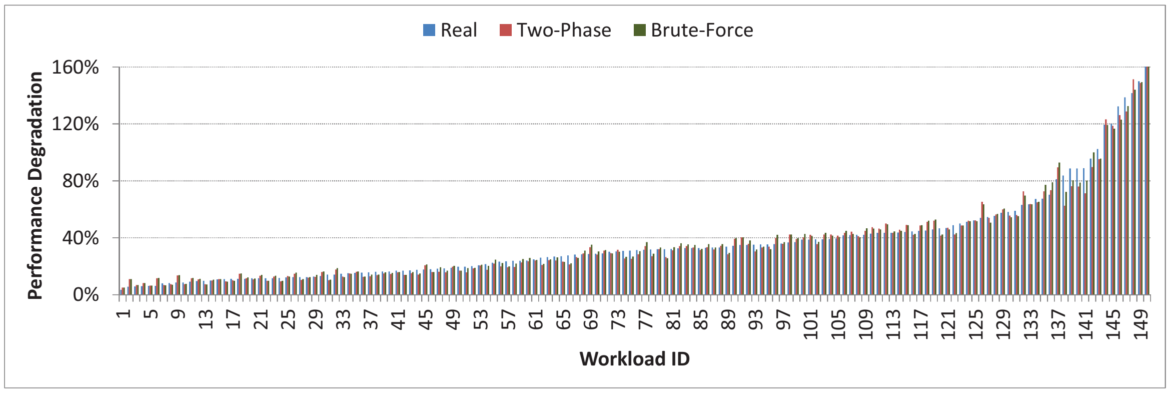

Fig. 10: Prediction precision of the two-phase and brute-force approaches for a SPEC2006 benchmark, `471.omnetpp`, co-running with 150 co-running workloads randomly generated from 18 SPEC2006 benchmarks.

D. Benefits of Piecewise Predictor Functions

We examine the benefits of a piecewise interpolation of the data points for our co-location problem. If a non-piecewise function is used instead, the predictor errors will usually increase, especially in the areas around the breakpoints of the corresponding piecewise predictor function.

Let us consider the 200 workloads in Figure 7. To obtain a non-piecewise predictor function, we use the same data points and configuration file as before except that the domain partitioning option is ignored. The resulting prediction errors range from 0.0% to 15.2% with an average of 3.5%. In contrast, the prediction errors from our piecewise predictor function range from 0.0% to 8.6% with an average of 0.2%.

Even though its average error may be small, a non-piecewise predictor tends to make larger prediction errors near the breakpoints of its corresponding piecewise predictor, called *bad areas*. For these workloads, our piecewise predictor given in (11) has two breakpoints. Two bad areas are observed: $2.2 < P_{bw} < 4.2$ and $8.6 < P_{bw} < 10.6$, with 45 out of 200 data points falling inside. For these data points, the prediction errors of the non-piecewise predictor range from 1.2% to 15.2% with an average of 9.0%. In contrast, the prediction errors of our piecewise predictor are much smaller, ranging from 0.0% to 4.5% with an average of 1.2%. For illustration purposes, both predictor functions are compared using five workloads in the two bad areas in Figure 11.

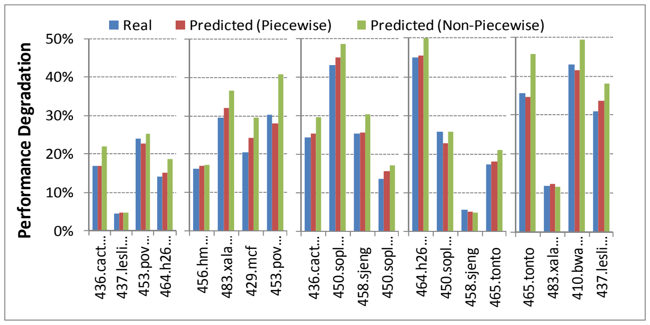

Fig. 11: Comparing piecewise and non-piecewise predictor functions using five workloads selected from "bad areas".

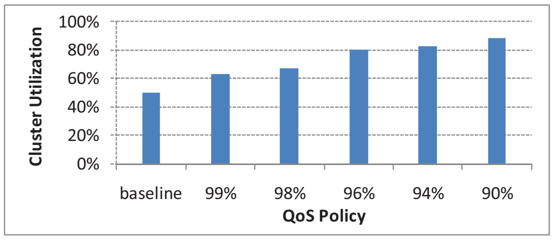

Fig. 12: Cluster utilization improvement when allowing co-locations with `nlp-mt` using our prediction model.

E. Applying Our Model in a Datacenter

In this section, we present an evaluation of applying our prediction model in a datacenter to increase hardware resource utilization. We allow the datacenter applications to have a small amount of QoS degradation, with the tolerable degradation threshold, e.g., 5% or 10%, specified by the user. We use our prediction model obtained in Section IV-A to predict the QoS degradation and allow co-locations when the predicted QoS degradation is within the specified threshold.

We use a real-time interactive language translator application, `nlp-mt`, which is functionally equivalent to *google translate*, as the main application whose QoS should be guaranteed. Our platform is a cluster with 300 Quad-core processors. There are 300 instances of `nlp-mt` with each running on two cores in a processor. There are 600 other application instances, randomly created from the other 8 applications given in Table II, with one instance per core. Our baseline is the resource utilization obtained when co-location is disabled for `nlp-mt`. We evaluate the utilization increases with our prediction model when co-location is enabled.

Figure 12 shows the cluster resource utilization improvement when using our prediction model to allow co-locations with `nlp-mt`. The baseline is 50% when co-location is disabled, in which case, each four-core processor allocates two cores for running `nlp-mt`, thus leaving the other two cores unused. The resource utilization for the cluster is defined

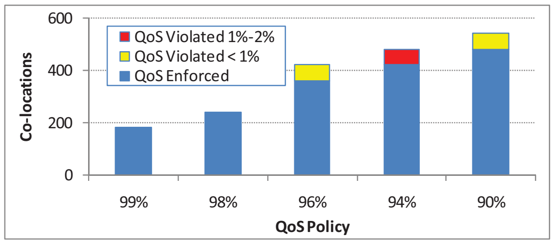

Fig. 13: Number of co-locations with `nlp-mt` under different QoS policies.

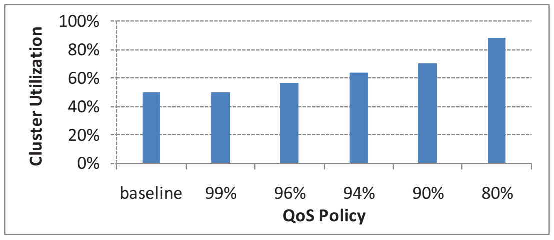

Fig. 14: Cluster utilization improvement when allowing co-locations with `openclas` using our prediction model.

as the aggregate performance of all applications running on the 300 processors, normalized by their performance in solo execution. Suppose A_1, A_2, A_3 and A_4 are co-located on a processor, with their performance degradation ratios being PD_1, PD_2, PD_3 and PD_4, respectively. Then the resource utilization on the processor is $(\sum_{i=1}^{4} \frac{1}{1+PD_i})/4$.

Figure 12 shows that our prediction model can help improve resource utilization. When the QoS policy is set as 99%, `MR-ANN` or `MR-KNN` can co-locate with `nlp-mt`, pushing the resource utilization from 50% to 63%. When the QoS policy is relaxed to 90%, the four cores on some processors are fully occupied, pushing the resource utilization further to 88%. In this case, we have improved the resource utilization from 50% to 88% at the cost of only 10% QoS degradation.

Figure 13 shows that as the QoS policy becomes more relaxed, the number of co-locations steadily increases, reaching 540 when the QoS policy is 90%. The peak is 600 as there are 600 unused cores when co-location is disabled. However, due to prediction errors, some co-locations may slightly violate the specified QoS policy. For example, the 90% QoS policy is violated by around 11% of the 540 co-locations, causing less than 1% extra QoS degradation. Such violations can be tackled by using an error tolerance scheme, as suggested in [24].

Figure 14 shows the cluster utilization improvement when we apply our approach to `openclas`. The experiments setup is the same as in Figure 12 except that `nlp-mt` has been replaced by `openclas`. Similar trends are observed in both cases. However, `openclas` is more sensitive to contention than `nlp-mt`, causing `openclas` to enjoy less utilization improvement under the same QoS policy. Take the 99% QoS policy as an example. While `MR-KNN` or `MR-ANN` can be co-located with `nlp-mt`, no co-location is allowed for `openclas`. As the QoS policy is relaxed, co-locations become possible, pushing the resource utilization from 50% to around 88% at the cost of 20% QoS degradation eventually.

F. Two More Platforms

These are (1) 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, and (2) a 1.86GHz six-core Intel Xeon E7-4807, with a private 128KB L1 D-cache, a private 128KB L1 I-cache, a private 1MB L2 cache, a shared 18MB L3 cache and a memory bandwidth of 25.6GB/s.

With the same warehouse created earlier, we repeat the same steps discussed in Section IV-A to build an abstract model for each new platform in the first phase. The training set obtained has 30 programs for the quad-core AMD Opteron and 60 programs for the six-core Xeon. The abstract model is instantiated similarly for each new program.

For the quad-core AMD Opteron, there are 26 SPEC2006 benchmarks not included in its training set. We randomly generated 200 co-running workloads from these benchmarks and randomly picked one representative for each workload. Figure 15 depicts the real and predicted performance slowdowns for the 200 representatives selected. The prediction errors range from 0.0% to 5.1% with an average of 0.3%.

For the six-core Xeon, there are 20 SPEC2006 benchmarks not included in its training set. Figure 16 is obtained for the six-core Xeon as an analogue of Figure 15. The prediction errors range from 0.0% to 10.2%, with an average of 0.1%.

V. Related Work

There has been a lot of work on addressing the contention for shared resources, especially shared cached, for multicore processors. *Cache partitioning* has been used to mitigate shared cache contention [33], [29], [30], [13], [5], [20], [44], [18], [22], [28], [7], [21], [38], [39]. For hardware solutions, cache resources are allocated to applications based on benefit rather than rate of demand [33], [29], [30], [46]. For software solutions, page coloring is used instead [7], [21], [44].

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 [6], [17], [19], [42], [40], [24], [25]. Furthermore, the scheduler can also adjust the resources allocated to an application to mitigate the contention to shared resources [45], [42], [12], [41], [36], [37].

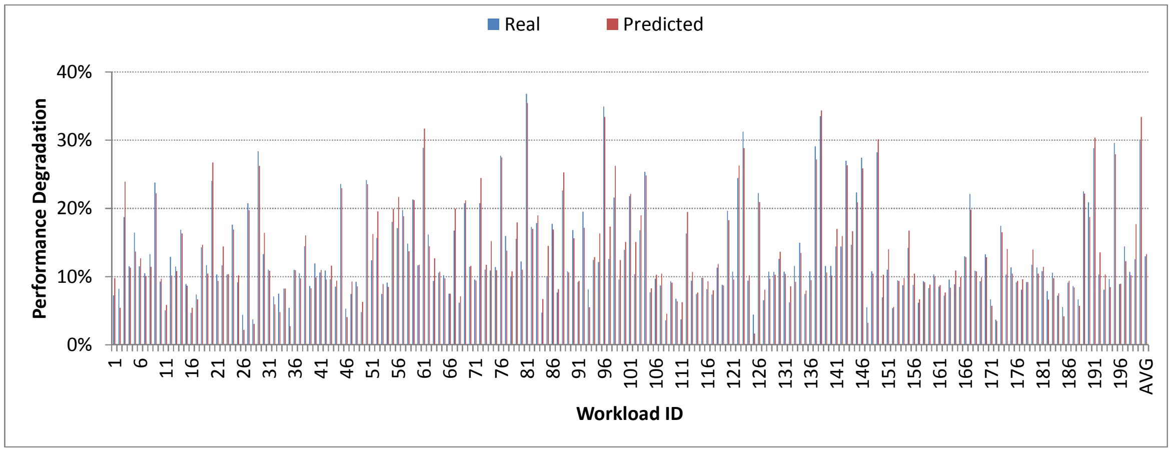

Fig. 15: Prediction precision for 200 randomly generated workloads from 26 SPEC2006 benchmarks on the quad-core AMD Opteron. For each workload, the real and predicted performance slowdowns of a randomly picked representative are shown.

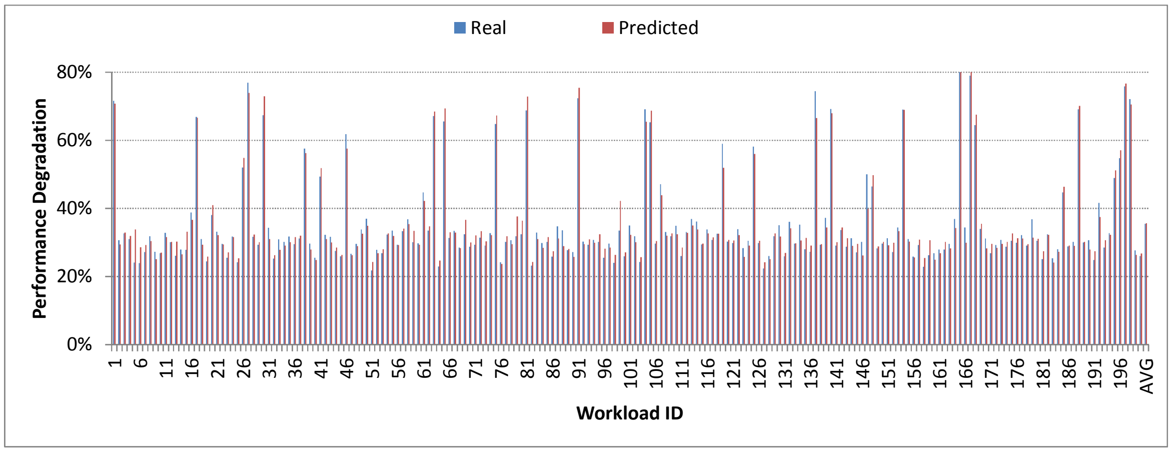

Fig. 16: Prediction precision for 200 randomly generated workloads from 20 SPEC2006 benchmarks on the six-core Xeon. For each workload, the real and predicted performance slowdowns of a randomly picked representative are shown.

There are also extensive studies on understanding and predicting shared cache contention on multicore processors. The best known techniques are Stack Distance Profiles (SDP) [27] and Miss Rate Curves (MRC) [27], [4], 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 [24], [25], [36] 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. Bandit [11] 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 [16], [45]. The performance interference model can be leveraged by the compiler to include some co-runner-aware code transformations and optimizations [1], [36]. Furthermore, some domain-specific optimizations [9], [8], [10] can be applied to some datacenter applications to make them co-locate better.

VI. Conclusion and Future Work

This paper presents 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. By using a piecewise rather than a non-piecewise predictor function, we can accommodate different types of dominant contention factors in its different subdomains, thereby achieving a more accurate prediction.

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 [32], [34], [43] with profiling.

Acknowledgements

This research is supported in part by the National High Technology Research and Development Program of China (2012AA010902), the National Key Basic Research Program of China (2011CB302504), the National Natural Science Foundation of China (61202055, 60925009, 61100011), the Innovation Research Group of NSFC (60921002), and Australian Research Council Grants (DP0987236, DP110104628 and DP130101970). We would like to thank all the reviewers for their valuable comments and suggestions.

References

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

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

[3] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *PACT*, 2008.

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

[5] J. Chang and G. S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *ICS*, 2007.

[6] S. Chen, P. Gibbons, M. Kozuch, V. liaskovitis, A. Ailamaki, G. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. Mowry, and C. Wilkerson. Scheduling threads for constructive cache sharing on CMPs. In *SPAA*, 2007.

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

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

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

[10] H. Cui, J. Xue, L. Wang, Y. Yang, X. Feng, and D. Fan. Extendable pattern-oriented optimization directives. *ACM Transactions on Architecture and Code Optimization*, 9(3):14, 2012.

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

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

[13] F. Guo and Y. Solihin. An analytical model for cache replacement policy performance. In *SIGMETRICS*, 2006.

[14] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *IISWC*, 2001.

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

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

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

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

[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] Y. Liang and T. Mitra. Cache modeling in probabilistic execution time analysis. In *DAC*, 2008.

[21] 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.

[22] F. Liu, X. Jiang, and Y. Solihin. Understanding how off-chip memory bandwidth partitioning in chip multiprocessors affects system performance. In *HPCA*, 2010.

[23] 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.

[24] 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.

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

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

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

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

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

[30] 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.

[31] K. Sankaralingam and R. H. Arpaci-Dusseau. Get the parallelism out of my cloud. In *HotPar*, 2010.

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

[33] 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.

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

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

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

[37] 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.

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

[39] X. Vera, B. Lisper, and J. Xue. Data cache locking for tight timing calculations. *ACM Transactions on Embedded Computing Systems*, 7(1), 2007.

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

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

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

[43] 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.

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

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

[46] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Computing Surveys*, pages 1–31, 2011.