ADAPT: A Framework for Coscheduling Multithreaded Programs

KISHORE KUMAR PUSUKURI, RAJIV GUPTA, and LAXMI N. BHUYAN, University of California, Riverside

Since multicore systems offer greater performance via parallelism, future computing is progressing towards use of multicore machines with large number of cores. However, the performance of emerging multithreaded programs often does not scale to fully utilize the available cores. Therefore, simultaneously running multiple multithreaded applications becomes inevitable to fully exploit the computing potential of such machines. However, maximizing the performance and throughput on multicore machines in the presence of multiple multithreaded programs is a challenge for the OS. We have observed that the state-of-the-art contention management algorithms fail to effectively coschedule multithreaded programs on multicore machines. To address the above challenge, we present ADAPT, a scheduling framework that continuously monitors the resource usage of multithreaded programs and adaptively coschedules them such that they interfere with each other's performance as little as possible. In addition, ADAPT selects appropriate memory allocation and scheduling policies according to the workload characteristics. We have implemented ADAPT on a 64-core Supermicro server running Solaris 11 and evaluated it using 26 multithreaded programs including the TATP database application, SPECjbb2005, and programs from Phoenix, PARSEC, and SPEC OMP suites. The experimental results show that ADAPT substantially improves total turnaround time and system utilization relative to the default Solaris 11 scheduler.

Categories and Subject Descriptors: D.4.1 [Process Management]: Scheduling, Threads; D.4.8 [Performance]: Monitors, Measurements

General Terms: Performance, Measurement, Algorithms

Additional Key Words and Phrases: Multicore, lock contention, cache miss-rate, fairness, coscheduling

ACM Reference Format:

Pusukuri, K. K., Gupta, R., and Bhuyan, L. N. 2013. ADAPT: A framework for coscheduling multithreaded programs. *ACM Trans. Architect. Code Optim.* 9, 4, Article 45 (January 2013), 24 pages.
DOI = 10.1145/2400682.2400704 <http://doi.acm.org/10.1145/2400682.2400704>

1. INTRODUCTION

The advent of multicore machines provides an attractive opportunity for achieving high performance for a wide variety of multithreaded programs. The performance of a multithreaded program running on a multicore machine often does not scale with the number of cores. Therefore, to fully exploit the computing potential of a machine with a large number of cores, it becomes inevitable that we simultaneously run multiple multithreaded programs. However, effective coscheduling of multithreaded programs on such machines is a challenge because of their complex architecture [Boyd-Wickizer et al. 2009; Peter et al. 2010]. For effective coscheduling of multithreaded programs, the

This research is supported in part by NSF grants CCF-0963996, CNS-0810906, CCF-0905509, and CSR-0912850 to the University of California, Riverside, CA.

Authors' addresses: K. K. Pusukuri (corresponding author), R. Gupta, and L. N. Bhuyan, Computer Science Department, University of California, Riverside, CA; email: kishoreguptaos@gmail.com.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2013 ACM 1544-3566/2013/01-ART45 \$15.00

DOI 10.1145/2400682.2400704 <http://doi.acm.org/10.1145/2400682.2400704>

OS must understand their resource usage characteristics and then adaptively allocate cores as well as select appropriate memory allocation and scheduling policies.

To address the above challenge, contention management techniques have been proposed [Blagodurov et al. 2011; Zhuravlev et al. 2010; Bhadauria and McKee 2010; Pusukuri et al. 2011c; Knauerhase et al. 2008]. These techniques are primarily guided by the cache usage characteristics of the programs, such as the last-level cache miss-rate, and are aimed at coscheduling of multiple *single-threaded* programs [Zhuravlev et al. 2010; Pusukuri et al. 2011c; Knauerhase et al. 2008] or for coscheduling threads of a *single multithreaded* program [Blagodurov et al. 2011]. For effective coscheduling of multiple multithreaded programs the cache usage characteristics alone are not enough. We demonstrate that it is necessary to also consider *lock contention* and *thread latency* to guide coscheduling decisions. Therefore existing techniques are not effective in coscheduling multithreaded programs. Existing techniques have some additional limitations. The contention management techniques in Blagodurov et al. [2011], Zhuravlev et al. [2010], Bhadauria and McKee [2010], Pusukuri et al. [2011c], and Knauerhase et al. [2008] are applicable to machines with a small number of cores and their evaluations were carried out using one thread per core configuration. While this configuration gives best performance for machines with four or eight cores, this is not true for machines with a larger number of cores [Pusukuri et al. 2011a]. Furthermore, the performance metrics used in some of the existing techniques [Pusukuri et al. 2011c] are not appropriate for coscheduling multithreaded programs [Eyerhan and Eeckhout 2008]. Finally, in addition to allocating cores, the OS must also adaptively select process scheduling and memory allocation policies according to the resource usage characteristics of the programs. However, existing techniques only consider allocation of cores.

To address the above challenges, we develop ADAPT, a framework for effective coscheduling of multithreaded programs on machines with large number of cores. ADAPT uses supervised learning techniques for predicting the effects of interference between programs on their performance and adaptively coschedules programs that interfere with each other's performance the least. Using simple performance monitoring utilities available on a modern OS, it adaptively allocates cores as well as assigns appropriate memory allocation and scheduling policies according to the resource usage characteristics of multithreaded programs. We have implemented ADAPT on a 64-core machine running Solaris 11 and evaluated it using 26 programs including the TATP database application [TATP 2003], SPECjbb2005 [SPECjbb 2005], and programs from PARSEC [Bienia et al. 2008], SPEC OMP [SPECOMP 2001], and Phoenix [Yoo et al. 2009] suites. Our experiments show that ADAPT achieves up to 44% improvement in turnaround time and it improves throughput of TATP and JBB by 23.7% and 18.4% relative to the default Solaris 11 scheduler. The overhead of ADAPT is negligible and it requires no changes to the application source-code or the OS kernel. Furthermore, while existing techniques [Zhuravlev et al. 2010; Blagodurov et al. 2011; Bhadauria and McKee 2010; Knauerhase et al. 2008] are based on fixed heuristics, ADAPT dynamically learns appropriate contention factors and their effect on the performance of programs on any target architecture. Therefore, we believe ADAPT will be able to evolve with changes in processor architecture and computing environment. The major contributions of this work are as follows.

- We demonstrate that coscheduling decisions must not be exclusively based upon the cache usage behavior, but rather lock contention and thread latency must also be considered.
- We develop statistical models based on supervised learning for identifying the effects of interference between programs on their performance.

(a) Our 64-core machine.

No.	Configuration	#Cores to A	#Cores to B
1	All-cores	64	64
2	Processor-set	32	32
3	Processor-set	24	40
4	Processor-set	40	24
5	Processor-set	16	48
6	Processor-set	48	16

(b) Cores-configurations.

Fig. 1. The machine has four 16-core CPUs that are interconnected via HyperTransport. The above table shows the number of cores allocated to two programs A and B in different cores configurations.

—We develop ADAPT with simple utilities available on Solaris 11. ADAPT improves the turnaround time by 21% on average and by a maximum of 44% relative to the default Solaris 11 scheduler.

The remainder of this article is organized as follows. Section 2 describes the motivation of this work and Section 3 presents the development of the ADAPT framework in detail. Section 4 describes the experimental setup and Section 5 presents the evaluation of ADAPT using a wide variety of multithreaded programs. Related work and conclusions are given in Sections 6 and 7.

2. INADEQUACIES OF EXISTING ALGORITHMS

In this section we demonstrate why existing contention management techniques that are based only upon the last-level cache miss-ratio are inadequate for coscheduling multithreaded programs. For this purpose, we conducted experiments on a 64-core machine running Solaris 11, involving coscheduling of four multithreaded programs (facesim (FS), bodytrack (BT), quake (EQ), and applu (AP)) taken from the PARSEC and SPEC OMP suites. We ran the programs using OPT number of threads, where OPT is the minimum number of threads that gives maximum performance during a solo run of a multithreaded program on our 64 core machine. As shown in Figure 1(a), this machine has four 16-core CPUs, that is, it has a total of 64-cores. To capture the distance between different CPUs and memories, a new abstraction called *locality group* (lgroup) has been introduced in Solaris. The lgroups are organized in a hierarchy that represents the latency topology of the machine [McDougall and Mauro 2006].

The existing contention management techniques for single-threaded workloads [Zhuravlev et al. 2010; Pusukuri et al. 2011c] or threads of a single multithreaded program [Blagodurov et al. 2011] minimize resource contention by scheduling memory intensive threads on different CPUs or different processor sets. A processor set is a pool of cores such that if we assign a multithreaded program to a processor set, then during load balancing the OS restricts the migration of threads across the cores within the processor set. A program is treated as being memory intensive if its last-level cache miss-ratio is high; otherwise it is considered to be CPU intensive. Other application characteristics such as *lock contention* and *thread latency* are not considered by the existing contention management techniques. Here, lock contention is the percentage of elapsed time that a program spends waiting for user locks, condition variables, etc. and thread latency is the percentage of elapsed time a program spends waiting for CPU resources, i.e., although a thread is ready to run, it is not scheduled on any core.

Table I. Coscheduling of Memory-Intensive Programs EQ and FS

Configuration	MPA	TTT	T_{EQ}	T_{FS}
All-cores	0.78	381	141	240
Processor-set	0.69	409	158	251
DINO	0.72	436	174	262

All-cores provides high performance. T_{EQ} and T_{FS} are the average running times of EQ and FS. $TTT = (T_{EQ} + T_{FS})$.

For effective coscheduling of multithreaded programs on a machine with large number of cores, we need to run multithreaded programs on processor sets such that memory hierarchy interference between them is minimized. Since number of processor set configurations can be numerous, we identify a subset of them that are the most suitable for coscheduling. We assume that at least one CPU (16 cores) is needed to obtain good performance for a multithreaded program on our 4 CPU machine. We derived this assumption by running 26 multithreaded programs with varying number of threads. Based upon this assumption, we have chosen five processor set configurations shown in Figure 1(b).

In our study, in each of the coscheduled runs, we run two multithreaded programs concurrently in two configurations: (1) all-cores configuration; and (2) processor set configuration. In all-cores configuration we run both the programs on all the 64 cores while in processor set configuration we run each program on a separate processor set to minimize interference between the programs. By default, the OS scheduler runs the programs in all-cores configuration. Here we chose the processor set configuration that gives the best performance in terms of total turn-around time (TTT) from among the five processor set configurations shown in Figure 1(b). Whenever possible, we avoid placing cores belonging to CPU pairs that have the maximum memory access latency gap in the same processor set.

2.1. Cache Miss-Ratio vs Lock Contention vs Thread Latency

In the first coscheduling run, we used two memory-intensive multithreaded programs: facesim (FS) from PARSEC and equake (EQ) from SPEC OMP. The best processor set configuration for this pair of programs is (32, 32); each program runs in a processor set of 32 cores. In all-cores configuration both FS and EQ share all the 64 cores. We also evaluated the existing contention management technique DINO [Blagodurov et al. 2011], a NUMA version of DI [Zhuravlev et al. 2010], for coscheduling FS and EQ. The key idea of DINO [Blagodurov et al. 2011] is to monitor cache miss-ratio of each thread and then coschedule threads that exhibit least interference with respect to memory hierarchy. It consequently reduces overall last-level cache misses per accesses (MPA)¹ and improves performance. EQ is fairly memory intensive with solo MPA of 0.79 in comparison to FS which has a solo MPA of 0.46. Moreover, different threads belonging to each program experience nearly the same MPA. Therefore, DINO separates memory-intensive EQ threads from each other by running them along with FS threads; here we tried all possible combinations of FS and EQ threads on all possible processor set configurations and chose the one that gives the best performance.

As we can see from Table I, all-cores configuration produces a high MPA in comparison to both processor set configuration and DINO. This is due to high memory

¹We also considered cache misses per instruction (MPI), but did not observe significant difference in using between MPI and MPA.

Fig. 2. Lock contention and latency of EQ and FS are higher in processor set configuration.

hierarchy interference between EQ and FS. However, all-cores configuration gives the lowest Total Turnaround Time (TTT). Here TTT is the sum of turnaround times of EQ and FS in the coscheduled run. Thus, this experiment shows that existing techniques such as DINO that try to avoid high MPA are not always effective in coscheduling multiple multithreaded programs. This is because FS and EQ are not only memory intensive, they are also exhibit high lock contention. Therefore, when threads of EQ and FS compete and consequently increase thread preemptions, they slow down the progress of lock-holder threads. As per state-of-the-art spin-then-block contention management policy, threads waiting for locks have greater likelihood of having to block [McDougall and Mauro 2006; Johnson et al. 2010]. In all-cores configuration, due to large number of cores, the OS scheduler has a better chance of allocating CPU resources and achieving quick lock hand-offs as the lock-holder threads can complete their critical section quickly and release the lock. This leads to lower lock acquisition latencies [McDougall and Mauro 2006]. Figure 2 shows the lock contention and thread latency values of EQ and FS in the coscheduling run. This data is collected by monitoring the percentage of lock contention and the percentage of latency of whole programs at one-second intervals. As we can see, both EQ and FS experience higher

Table II. Coscheduling of CPU-Intensive Programs BT and AP

Configuration	TTT	RQ	CX-Rate
All-cores	116.8	1.6	75552
Processor-set	107.2	4.7	51546

Processor set provides high performance. T_{BT} and T_{AP} are the average running times of BT and AP. $TTT = (T_{BT} + T_{AP})$.

lock contention and thread latency in processor set configuration than in all-cores configuration.

When we run two CPU-intensive and high lock contention programs BT and AP, processor set configuration provides higher performance than all-cores configuration as shown in Table II. We ran these programs in both all-cores and processor set configurations with their respective OPT threads (50 and 24). The best processor set configuration for BT and AP is (40, 24): 40 cores for BT and 24 cores for AP. This coscheduling run is very interesting as there is a trade-off between latency and lock contention, as opposed to MPA and lock contention that was observed in the previous coscheduling run. Since BT and AP are CPU-intensive programs, MPA is not a significant consideration in their coscheduled run. Therefore, techniques presented in Blagodurov et al. [2011], Bhadauria and McKee [2010], Knauerhase et al. [2008], Pusukuri et al. [2011c], and Zhuravlev et al. [2010] may not effectively deal with the coscheduling of BT and AP.

As we can see from Table II, the system run-queue length (RQ) is smaller for the all-cores configuration in comparison to the processor set configuration; RQ is the total number of runnable threads in the dispatcher queues of the system [McDougall and Mauro 2006]. Therefore, as shown in Figure 3, thread latencies (LAT) of BT and AP are lower in all-cores configuration than in the processor set configuration. However, lock contention (LOCK) for both BT and AP is high in all-cores configuration. Since BT and AP are CPU-intensive and high lock contention programs, and because of their high interaction in all-cores configuration, they experience high context-switch (CX) rate [Pusukuri et al. 2011b; Johnson et al. 2010]. In this experiment DTrace scripts were used for measuring RQ and CX-Rate [Cantrill et al. 2004; McDougall and Mauro 2006].

The above experiments demonstrate that MPA alone is not enough for effective coscheduling of multithreaded programs on a multicore system. The OS must also consider application characteristics of lock contention and thread latency along with MPA. Based on these observations, in the next section, we present a framework called ADAPT that dynamically monitors resource usage characteristics of multithreaded programs, and based upon these, it effectively coschedules the programs.

3. THE ADAPT FRAMEWORK

The ADAPT framework has two major components: the Cores Allocator and the Policy Allocator. The Cores Allocator is responsible for selecting appropriate cores configuration. The Policy Allocator is responsible for adaptively applying appropriate memory allocation and scheduling policies. The following sections provide detailed description of these components.

3.1. The Cores Allocator

To capture application resource usage characteristics, the Cores Allocator uses statistical models. These models are constructed using supervised learning, where a set of input-output values is first observed and then a statistical model is trained to predict similar output values when similar input values are observed [Hastie et al. 2009]. The Cores Allocator uses two statistical models: one for approximating performance loss of

Fig. 3. Coscheduling of BT and AP. Thread latencies of BT and AP in all-cores configuration are low compared to processor set configuration. However, lock contention of BT and AP are high in all-cores configuration.

a program due to its coscheduling with another program; and another for approximating performance of a program when we change the configuration from processor set to all-cores configuration, and vice versa. Let us call the first model as *PAAP* (Performance Approximation of a program when it is running with Another Program) and the second model as *PACC* (Performance Approximation of a program when it is running on a different Cores Configuration). Using the PAAP model, the Cores Allocator predicts average performance of a program in all-cores configuration. Using the PACC model, the Cores Allocator predicts average performance of a program in the five different processor set configurations listed in Figure 1(b). Then it chooses the configuration that gives the best average performance.

We developed the models as follows. To cover a wide range of resource usage characteristics, we categorized the programs as memory-intensive, CPU-intensive, high lock contention, or low lock contention programs. To develop the models we selected 12 programs from a total of 26 such that a few programs were chosen from each category. The chosen programs include: bodytrack, facesim, ferret, fluidanimate, streamcluster from PARSEC;

Table III. Initial Predictors and the Target *usr.ab* of the PAAP model

Predictor	Description
<i>mpa_x</i>	average last-level cache miss ratio of program x.
<i>usr_x</i>	the percentage of elapsed time program x spends in user mode.
<i>sys_x</i>	the percentage of elapsed time program x spends in processing system calls, system traps, etc.
<i>lat_x</i>	latency of program x.
<i>lock_x</i>	lock contention of program x.
<i>ct_x</i>	cores to threads ratio of program x, i.e., (<i>#cores</i> / <i>#threads</i> of x).
<i>usr_ab</i>	the percentage of elapsed time program <i>a</i> spends in user mode when it is running with program <i>b</i> .

The goal is to predict *usr.ab* of program *a* when it is running with program *b* and vice versa.

100% of elapsed time of a program = the percentage of time program spends in user space
+ the percentage of time program spends lock contention
+ the percentage of time program spends latency
+ the percentage of time program spends in kernel space.

Fig. 4. Relationship between elapsed time (or turnaround time) and the predictors.

applu, art, swim, equake from SPEC OMP; SPEC JBB2005; kmeans and pca from Phoenix. The resource usage characteristics of these programs are used as inputs to the statistical models.

3.1.1. The PAAP Model. Data Collection. The goal of the PAAP model is to predict the performance of a program A when it is running with another program B. We chose six types of predictors for developing the PAAP model. Since we have two programs, we have 12 predictors in all representing the resource usage characteristics of the two programs A and B. The seventh parameter shown in Table III is the response variable. *r_x* represents a resource usage characteristic value “r” of program “x” in its solo run with OPT Threads. Figure 4 explains the relationship between the elapsed time and the predictors. We use *prstat(1)* utility [McDougall et al. 2006] to monitor the predictors. *prstat(1)* provides microstat information for the individual threads and for the whole application, which is simply the average across threads. Since ADAPT performs application-level scheduling, as opposed to thread-level scheduling, we monitor microstat information of the whole application in this work. The elapsed time for the full program in terms of other microstat information is given in Figure 4.

From the combinations of the aforementioned 12 programs, we collect 144 data points, where each data point is a 13 tuple containing the 12 predictors and the observed value of *usr.ab* as the target parameter shown in Table III. Here each of the 12 programs contributes 12 data points including a combination with itself. We collect 100 samples using Solaris 11 utilities *prstat(1)* and *cpustat(1)* with 100 ms time interval and the averages of these samples are used as the final values of the predictors. The *cpustat(1)* utility is used to collect *mpa* and the *prstat(1)* utility is used for the remaining predictors. We assume that the percentage of elapsed time that a program spends in user mode represents its progress or performance in the coscheduling run. Solaris provides utilities that effectively monitors application resource usage characteristics even in the presence of thread migrations.

Finding Important Predictors. To balance the prediction accuracy and the cost of approximation, we use forward and backward input selection techniques with Akaike Information Criterion (AIC) for finding important predictors among the 12 initial predictors. The AIC is a measure of the relative goodness of fit of a statistical model

Table IV. VIF Values of PAAP Predictors

mpa_a	lock_a	lat_b	ct_b	sys_a
1.6	2.1	2.1	1.6	1.3

Table V. PAAP Models

Model	Adjusted R^2	Prediction Accuracy
LR	0.90	88.3
DT	0.94	90.6
KNN	0.88	86.6

$$usr_ab = (65.2) + (-0.6 * lock_a) + (-0.8 * lat_b) + (-9.6 * mpa_a) + (-10.2 * sys_a) + (7.8 * ct_b) \quad (1)$$

[Hastie et al. 2009]. Using R stepAIC() [R] method, we identified the five most important predictors: *lock_a*, *lat_b*, *ct_b*, *mpa_a*, and *sys_a*. We also tested the predictors for the multicollinearity problem to develop robust models. Multicollinearity is a statistical phenomenon in which two or more predictor variables in a multiple regression model are highly correlated. In this situation the coefficient estimates may change erratically in response to small changes in the model or the data. We use R Variance Inflation Factor (VIF) method to observe the correlation strength among the predictors. If $VIF > 5$, then the variables are highly correlated [vif]. As shown in Table IV, the variables are not highly correlated and therefore there is no multicollinearity problem.

Model Selection. Using the preceding five important predictors we develop three popular models based on supervised learning techniques. The models are: (a) Linear Regression (LR); (b) Decision Tree (DT); and (c) K-Nearest Neighbour (KNN) [Hastie et al. 2009]. We use R statistical methods `lm()` [R], `rpart()` [R], and `kknn()` [R] for developing these models. Here the decision tree model is pruned using R `prune()` [R] method to avoid the overfitting problem. As we can see, in the LR model (Eq. (1), lock contention, latency, cache miss-ratio, and system overhead of program A negatively impact A's performance when it is running with program B. Thus, if there is an increase in any of these four predictors, then *usr_ab* decreases. If cores-to-threads ratio of program B is increased (i.e., number of threads of B is decreased), then the performance of program A will improve and vice versa.

We evaluate the three models: LR, DT, and KNN using a 12-fold cross-validation (CV) test [Hastie et al. 2009]. Table V shows the adjusted R^2 values of these models on full training data and prediction accuracies in the 12-fold CV test. In a 12-fold CV test, we split the data (144 points) into 12 equal-sized partitions. The function approximator is trained using all the data except for one partition and a prediction is made for that partition. For testing the models thoroughly, we trained the models using the data from 11 different programs (132 data points) and tested it against the data of the twelfth program. The testing data used is different from the training data.

As shown in Table V, DT model is found to have the highest prediction accuracy among the three models and therefore we chose the DT model as the PAAP model. The metric *prediction accuracy* is defined as: $(100 - MAPE)$, where MAPE is the Mean Absolute Percentage Error.

3.1.2. The PAAC Model. Data Collection. We chose six predictors for developing the PACC model. The goal is to predict the performance of a program A when it is running under different cores configuration. The six predictors are: *usr_A*, *sys_A*, *lat_A*, *lock_A*, *ct_A*, and *rct_A*. As in the case of the PAAP model, the first five predictors represent the resource usage characteristics values of program A in its solo run and the remaining

Table VI. PAAC Models

Model	Adjusted R^2	Prediction Accuracy
LR	0.88	87.6
DT	0.86	87.1
KNN	0.86	83.5

Table VII. VIF values of the PACC Model Predictors

Predictor	lock_a	rct_a
VIF	1.2	1.2

$$usr_acc = (18.6) + (-0.3 * lock_a) + (32.5 * rct_a) \quad (2)$$

predictor *rct.A* is a cores configuration with reduced cores-to-threads ratio for program A. The goal is to predict the performance, i.e., *usr_acc* (the percentage of user-mode time), of program A when it is running with different cores configuration. We ran each of the aforesaid 12 programs using OPT threads on 64, 56, 48, 40, 32, 24, and 16 cores and collected 6 data points for each program. Therefore from the solo runs of the preceding 12 programs, we collected a total of 72 data points, where each data point is a 7 tuple containing six predictors and the observed *usr_acc*.

Finding Important Predictors and Model Selection. We identified the two most important predictors for the PACC model from among the six predictors and these predictors are: *lock_a* and *rct_a*. The LR model developed with these two predictors is shown in Eq. (2). As we can see from the LR model (Eq. (2)), lock contention negatively affects performance, i.e., if there is an increase in lock contention then *usr_acc* decreases. If *rct_a* increases (i.e., number of cores increases) then the performance of the program also increases. As we can see in Table VII, the VIF values of these predictors are also less than 5. Therefore, there is no multicollinearity problem.

As in deriving the PAAP model, we develop LR, DT, and KNN models using the two important predictors: *lock_a* and *rct_a*. As shown in Table VI, LR model (Eq. (2)) has the best prediction accuracy in a 12-fold CV test. Therefore, we chose the LR model as the PACC model.

Using the PAAP and PACC models, Cores Allocator selects appropriate cores configurations based upon the resource usage characteristics of programs. The overhead of these models is modest as they use very few predictors. In the next section, we describe the design of Cores Allocator.

3.1.3. Design of Cores Allocator. Cores Allocator considers a realistic scenario where programs can enter and leave the system at any time. Let us consider the case of coscheduling two programs. While program P_1 is running with its corresponding OPT threads, another program P_2 enters the system. If P_2 is a CPU-intensive and low lock contention program, then irrespective of the current cores configuration and the programs already running on the system, the Cores Allocator assigns all-cores configuration to P_2 . Otherwise, it predicts performances of P_1 and P_2 using the PAAP and PACC models, and allocates the cores configuration that gives lower average TTT. As additional programs enter the system, their appropriate cores configuration is similarly computed using the PAAP and PACC models. Next let us consider the scenario where programs $P_1, P_2, \dots, P_N$ are running on the system and then the program P_i completes its execution and leaves the system. If P_i is a CPU-intensive and contention-free

Table VIII. The Actual and Predicted usr_FA and usr_SM Values with the PAAP and PACC Models are Shown Here

Program	All-cores		Processor-set	
	Actual	Predicted	Actual	Predicted
FA	52.3	56.4	41.2	44.5
SM	49.4	45.2	46.8	41.6

Fig. 5. While APSI has steady behavior, FMA shows a significant phase change.

program, the current configuration for the remaining programs is maintained. Otherwise, the cores released by P_i are distributed equally among the remaining programs.

Let us consider an example that shows how Cores Allocator selects an appropriate cores configuration for programs fluidanimate (FA) and swim (SM). Both FA and SM are memory-intensive and low lock contention programs, and their OPT threads are 49 and 32 respectively. Using the PAAP and PACC models, first Cores Allocator predicts the performance of FA and SM in all-cores and in the best processor set configuration (40 cores to FA, 24 cores to SM). Table VIII shows that both FA and SM have high %USR (the percentage of elapsed time a program spends in user-mode) in the all-cores configuration. Therefore, Cores Allocator selects the all-cores configuration for coscheduling FA and SM. The all-cores configuration improves TTT of FA and SM by 14% compared to the processor set configuration.

Dealing with Phase Changes. Since FA and SM do not exhibit *significant* phase changes on our machine, need for switching back and forth between different cores configurations never arises. The initially predicted processor set configuration gives the best performance. Thus the Cores Allocator maintains all-cores configuration for the entire duration of the FA and SM coscheduled run. However, some programs show significant phase changes. Therefore, we continuously monitor the program for adaptively allocating appropriate cores configuration according to the phase changes of the programs.

Let us consider a coscheduled run of two high lock contention programs apsi (APSI) and fma3d (FMA) with their OPT threads of 16 and 56 respectively. As we can see from Figure 5, FMA has a significant phase change, while APSI shows steady behavior. FMA experiences very high lock contention in the first 11 seconds of its life-time, while APSI experiences very high lock contention throughout its life-time. Therefore, by continuous monitoring, and the use of PAAP and PACC models, the Cores Allocator applies (16, 48)

processor set configuration during the first 11 seconds and then all-cores configuration for the remaining time. This results in performance improvement of 8% relative to the default OS scheduler (i.e., all-cores configuration).

Overhead of Cores Allocator. Since we monitor resource usage information of the whole *application* instead of individual *threads*, the overhead of Cores Allocator is negligible and it scales well. For n programs, $(n)C(n-2)$ combinations are evaluated by PAAP. For example, for 4 applications (A, B, C, D), we evaluate 6 combinations (AB, AC, AD, BC, BD, CD). Cores Allocator takes a maximum of 2 milliseconds for selecting the best cores configuration for coscheduling of four applications on our machine.

We have shown that, using the PAAP and PACC models, Cores Allocator adaptively allocates appropriate cores configuration according to the resource usage characteristics of the programs and effectively deals with the program phase changes. In the next section, we describe how the Policy Allocator adaptively selects appropriate memory allocation and processor scheduling policies based on the resource usage characteristics.

3.2. The Policy Allocator

Contemporary operating systems such as Solaris and Linux do not distinguish between threads from multiple single-threaded programs and multiple threads corresponding to a single multithreaded program. Though the default OS scheduling and memory allocation policies work well for multiple single-threaded programs, this is not the case for multithreaded programs. This is because many multithreaded programs involve communication between threads that leads to contention for shared objects and resources. Since the OS does not consider application-level characteristics in scheduling and memory allocation decisions, the default OS scheduling and memory allocation policies may not be appropriate for achieving scalable performance. Most of the existing contention management techniques are primarily designed for single-threaded programs and they only deal with allocation of cores among the threads. To address these limitations, ADAPT uses another component, the Policy Allocator, which is responsible for dynamically selecting appropriate memory allocation and process scheduling policies based upon program resource usage characteristics.

3.2.1. Memory Allocation vs OS Load Balancing. As shown in Figure 1, the HyperTransport (HT) serves as the CPU interconnect and provides path to the I/O controllers. Using HT, CPUs can access each other's memory, and any data transferred from the I/O cards travels via the HT. Effective utilization of HT on a NUMA machine is important for achieving scalable performance for multithreaded programs, particularly because for a memory-intensive multithreaded program, the OS scheduler distributes the threads across the CPUs.

In Solaris 11, the *next policy* that allocates memory next to the thread is the default memory allocation policy for private heap/stack and the *random policy* is the default memory allocation policy for shared memory when the size of shared memory exceeds the threshold value of 8MB. This threshold is set based upon the communication characteristics of Message Passing Interface (MPI) programs [McDougall and Mauro 2006]. Therefore, it is not guaranteed that the *random policy* will always be used for the shared memory of multithreaded programs that are based on pthreads. If the shared memory is less than 8MB, then the *next policy* is also used for shared memory. With the *next policy*, a memory-intensive thread can experience high memory latency overhead and consequently high cache miss-ratio, when it is started on one core and then migrated to another core that does not belong to its home lgroup. More importantly, such thread migration makes HT a performance limiting hot spot. Therefore, the interaction between memory allocation policy and load balancing performed by the OS prevents scalable performance for memory-intensive multithreaded programs.

Fig. 6. CPI is high with next policy. Random policy improves memory bandwidth.

Unlike the *next policy*, the *random policy* picks a random leaf lgroup to allocate memory for each page. Thus, the random policy allocates memory across all the leaf lgroups. The threads of memory-intensive programs get a chance to reuse the data in both private and shared memory. This reduces memory latency penalty and cache miss-ratio. Moreover, since random policy spreads the allocated memory across the memory banks, it distributes the load across memory controllers and bus interfaces, thereby preventing any single component from becoming a performance limiting hot spot [McDougall and Mauro 2006]. Next we demonstrate the above benefits of the random policy over the next policy by running a highly memory-intensive program StreamCluster (SC) using the two policies in all-cores configuration.

In this experiment, we run SC with its OPT Threads of 17 in all-cores configuration. Cycles per instruction (CPI) indicates whether or not HT and memory buses are performance limiting spots. As shown in Figure 6, CPI of SC with *next* is higher than with *random policy* and total memory bandwidth (GB/sec) is improved by 17% with *random policy*. Therefore, random policy relieves pressure on HT, improves overall performance of memory-intensive programs, and also improves system utilization. Thus, multithreaded programs with huge private memory benefit greatly from the *random policy*. Moreover, our prior work shows that the random policy not only improves performance, it simultaneously reduces performance variation in multithreaded programs [Pusukuri et al. 2012].

Memory Allocation vs Access Latency of Locks. The performance of SC improves dramatically (by 56%) when *random policy* is used instead of the *next policy*. This improvement is not exclusively caused by improved memory bandwidth. It also results from reduction in lock contention because of random allocation of private memory (heap and stack) across lgroups. Allocating private memory across lgroups using *random policy* allows threads to quickly access lock data structures in the shared cache, thus minimizing memory traffic and the delay in acquiring locks. As shown in Figure 7(a), applying *random policy* for private memory reduces lock contention of SC by 19%.

3.2.2. Scheduling Policy vs Lock Contention. The default Time Share (TS) scheduling policy is not appropriate for high lock contention multithreaded programs under high loads. Prior work has shown that the interaction between TS policy and the state-of-the-art spin-then-lock contention management policy dramatically increases the thread context-switch rate and leads to drastic degradation in performance [Johnson

(a) random policy reduces lock contention (b) FF policy reduces context-switch rate

Fig. 7. Random policy reduces lock contention of SC by 19%. FF policy reduces context-switch rate of APSI.

et al. 2010; Pusukuri et al. 2011b]. We considered the use of both the Load Controller [Johnson et al. 2010] and the FF policy [Pusukuri et al. 2011b] as replacement of TS policy for dealing with lock contention during coscheduled runs. However, since Load Controller requires changes to application code, and its overhead increases linearly with the number of threads, we decided to make use of the FF policy. Our Policy Allocator selectively uses the FF policy. By assigning same priority to all the threads of a given multithreaded program, FF policy breaks the vicious cycle between thread priority changes and context switches. This dramatically reduces the context-switch rate (CX-Rate) and improves the performance especially under high loads. FF policy allocates time-quantum based on the resource usage of a multithreaded program for achieving fair allocation of CPU cycles among the threads. For example, when we run a high lock contention program *apsi* with its OPT threads (16), FF policy reduces its CX-Rate (shown in Figure 7(b)) and improves the performance by 9%.

3.2.3. Design of Policy Allocator. Policy Allocator continuously monitors cores configuration selected by the Cores Allocator and the resource usage characteristics, MPA and Lock contention, of the multithreaded programs for guiding the selection of appropriate memory allocation and scheduling policies. If the program is CPU intensive and low lock contention, and it is in all-cores configuration, then the Policy Allocator applies the *next* policy and the TS policy. Otherwise, it applies the *random* policy (or *random.pset* policy for processor set configuration) and the FF policy with appropriate time quantum. Since there is interference between programs in all-cores configuration, Policy Allocator applies only one policy, either TS or FF, to all applications in the *all-cores* configuration. This is because we observe that running the programs with two different scheduling policies in all-cores configuration dramatically degrades overall performance for some programs. However, we selectively apply both policies (TS and FF) in processor set configuration according to the resource usage characteristics of applications. It should be noted that Pusukuri et al. [2011b, 2012] do not explore coscheduling of multithreaded programs. In this work, we selectively apply FF policy for effectively coscheduling multithreaded programs.

3.3. Implementation of ADAPT

Our implementation of ADAPT uses a daemon thread for continuously monitoring running programs, maintaining their resource usage characteristics, assigning cores configuration using the Cores Allocator, and selecting memory allocation and scheduling

policies using the Policy Allocator. A Resource Usage Vector (RSV) is maintained for each program. More specifically, RSV of a program contains the following: resource usage characteristics including *usr*, *lock*, *lat*, *sys*, *ct*, *mpa*; CPU utilization per processor set if the program is coscheduled in a processor set configuration; and *cores configuration* selected by the Cores Allocator. Based on the cores configuration, cores-to-threads (*ct.a*) ratio is interpreted as either *ct.a* for the PAAP model or *rct.a* for the PACC model.

For monitoring programs and collecting resource usage data, assigning different cores configurations as well as memory allocation and scheduling policies, ADAPT uses the following Solaris 11 utilities: *prstat*, *mpstat*, *priocntl*, *pmadvise*, *mdb*, and *cpustrack*. *prstat* is used to collect *usr*, *lock*, *lat*, and *sys* characteristics, while *cpustrack* is used to collect *mpa*. *mpstat* collects system-wide resource usage characteristics such as overall system utilization and CPU utilization per processor set. We use *mdb* and *pmadvise* to apply memory allocation policies and *priocntl* for applying scheduling policies. While *mdb* is used to apply memory allocation policies system-wide for all programs, *pmadvise* is used for applying the memory allocation policy per program.

With the minimum time interval of one second provided by the default implementation of *prstat* and *mpstat* utilities, it is difficult to respond to rapid phase changes in programs. Therefore, we enhanced these utilities² to allow time intervals with millisecond resolution and capture phase changes of programs. Furthermore, the default *cpustat* utility does not support the use of performance monitoring events to collect system-wide resource usage characteristics (e.g., last-level cache miss-ratio) when there is more than one active processor set. Therefore, we have also enhanced the *cpustat* utility to collect system-wide characteristics with arbitrary number of processor sets.

Selecting an Appropriate Monitoring Time Interval. Using the above enhanced utilities, ADAPT is able to collect resource usage characteristics of the target programs with millisecond resolution. The time interval for collecting resource usage data directly impacts the overhead of ADAPT. Although a small time interval allows fine-grain details of the resource usage data to be collected, it increases the monitoring overhead. Therefore selecting an appropriate time interval is very important. To select an appropriate time interval, as shown in Figure 8, we evaluated ADAPT with different time intervals for monitoring four multithreaded programs simultaneously running on our machine. As Figure 8 shows, when we use ADAPT with 50 ms and 100 ms time intervals, the system overhead is considerably high. This is because the high rate of interprocessor interrupts and cross-calls leads to high system time [McDougall and Mauro 2006]. With a time interval of 200 ms or greater, the overhead of ADAPT is negligible (<1.5% of system time). Therefore, ADAPT uses 200 ms time interval for collecting RSVs of the multithreaded programs. ADAPT collects 10 samples of the resource usage data of the target programs with 200 ms time interval and updates RSVs of programs with the average of these 10 samples every two seconds. Therefore, every two seconds, based on the phase changes, it applies the appropriate cores configuration, memory allocation, and scheduling policies.

We have observed that rapid changes in the *cores configuration* diminishes the benefits of ADAPT. Therefore ADAPT keeps the last three RSVs of each program and then changes the cores configuration if one of the following conditions is satisfied.

- (1) if programs are running in a processor set configuration, and the average CPU utilization of any processor set is less than that of any other processor set by a threshold of at least α ;
- (2) for any program P_i , if its *usr* P_i decreases at a rate greater than a threshold of β in the last two intervals;

²Source code is at <http://www.cs.ucr.edu/~kishore/hipeac13.html>.

Fig. 8. Time interval duration vs. system overhead.

Table IX. Target Machine and Operating System

Supermicro 64-core server:
4 × 16-Core 64-bit AMD Opteron TM 6272 Processors (2.1 GHz);
L1 : 48 KB; Private to a core; L2 : 1024 KB; Private to a core;
L3 : 16384 KB; Shared among 16 cores; Memory: 64 GB RAM;
Operating System: Oracle Solaris 11 TM

- (3) $\frac{\sum_{i=1}^N (usr_P_i^P)}{N} > (\frac{\sum_{i=1}^N (usr_P_i^C)}{N} + \gamma)$ in the last two intervals, where γ is the threshold value;

where $usr_P_i^C$ is the actual %USR time of P_i in the current cores configuration, while $usr_P_i^P$ is the predicted %USR time of program P_i using either the PAAP or the PACC model based on the current cores configuration. From extensive experimentation with the programs used, we derived the threshold values of α , β , and γ as 6%, 4%, and 8%. By employing these thresholds, we are able to reduce the impact of unnecessary rapid changes between cores configurations on the performance of the programs. In the current implementation of ADAPT, we assume that solo run RSVs of the target programs are available. Alternatively, we can run the application for a few milliseconds as it enters the system and collect its RSV. Using signals (SIGSTOP/SIGSTART), we can pause the other applications while the RSV of the new application is being collected.

4. EXPERIMENTAL SETUP

4.1. Target Machine and OS

Our experimental setup consists of a 64-core machine running Solaris 11. Table IX shows its configuration.

Why Solaris. The memory placement optimization feature and chip multithreading optimization allow Solaris OS to effectively support hardware with asymmetric memory hierarchies such as NUMA [McDougall and Mauro 2006]. Specifically, the Solaris kernel is aware of the latency topology of the hardware via lgroups that allows it to optimize scheduling and resource allocation decisions. Moreover, Solaris provides a rich user interface to modify process scheduling and memory allocation policies. It also provides several effective low-overhead observability tools including DTrace, a dynamic kernel tracing framework [Cantrill et al. 2004].

4.2. Benchmarks and Performance Metrics

We evaluate ADAPT using 26 programs: TATP [TATP 2003] database transaction application; SPECjbb2005 [SPECOMP 2001]; eight programs from PARSEC [Bienia et al. 2008] including streamcluster (SC), facesim (FS), canneal (CA), x264 (X264), fluidanimate (FA), swaptions (SW), ferret (FR), and bodytrack (BT); 11 programs from SPEC OMP [SPECOMP 2001] including swim (SM), equake (EQ), wupwise (WW), gafort (GA), art (ART), apsi (AS), ammp (AM), applu (AP), fma3d (FMA), galgel, (GL), and mgrid (MG); and five programs from Phoenix [Yoo et al. 2009] including kmeans (KM), pca (PCA), matrix-multiply (MM), word-count (WC), and string-match (STRM).

The implementations of PARSEC programs are based upon pthreads and we ran them using native inputs (i.e., the largest inputs available). SPEC OMP programs were run on medium input datasets. SPECjbb2005 (JBB) with single JVM is used in all our experiments. TATP (a.k.a NDBB and TM-1) uses a 10000 subscriber dataset of size 20MB with a solidDB [solidDB] engine. TATP is not IO-intensive and thus the disk performance does not affect its performance significantly [Johnson et al. 2010]. Phoenix programs are based upon MapReduce. We were unable to use some of the other programs from the preceding benchmark suites because they have short running-times. In this work, we ran each experiment 10 times and present average results from the ten runs.

Performance Metrics. We use two metrics inspired by Eyerman and Eeckhout [2008] to evaluate ADAPT: a user-oriented metric: average total turnaround time (TTT); and a system-oriented performance metric: average system utilization, where system utilization = $100 - (\% \text{CPU idle time})$.

5. EVALUATING ADAPT

In this section, we analyze the effectiveness of the ADAPT framework using several coscheduling experiments with the 26 multithreaded programs as shown in Figure 9(a). As we can see, we evaluate ADAPT by coscheduling either two, three, or four programs. Figure 9(b) shows that the improvement in TTT achieved by ADAPT is on an average 21% (average lies between 16.1% and 25.2% with 99% confidence interval) and up to 44% relative to the default Solaris 11 scheduler.

As shown in Figure 9(b), while ADAPT achieves high TTT improvements for the coscheduled runs of memory-intensive and high lock contention programs (e.g., FS), it achieves moderate TTT improvements for the coscheduled runs of CPU-intensive and low lock contention programs (e.g., SW). ADAPT achieves significant improvements in throughput for the coscheduled runs of TATP database transaction application and JBB. ADAPT improves throughputs of TATP and JBB by 23.7% and 18.4% respectively in comparison to the default Solaris scheduler. For CPU-intensive and low lock contention programs, Policy Allocator contributes more to the improvements in TTT than the Cores Allocator because Cores Allocator allocates all-cores configuration like the default OS scheduler for these programs. Figure 9(c) shows that ADAPT achieves high system utilization, compared to the default Solaris scheduler. Thus ADAPT simultaneously improves performance of programs and the system utilization.

ADAPT vs Existing Techniques. As discussed in Section 3, the existing coscheduling algorithms [Zhuravlev et al. 2010; Blagodurov et al. 2011; Pusukuri et al. 2011c] are primarily designed for a mix of *single-threaded* workloads or threads of a *single multithreaded* workload. Therefore they use thread-level scheduling while ADAPT uses application-level scheduling to enable handling of multiple multithreaded programs. Since DINO [Blagodurov et al. 2011] uses one thread per core configuration, to compare ADAPT with DINO, we have chosen coscheduling runs where ADAPT also uses number of threads that equal the number of cores. There are 6 coscheduled runs

#	Programs	OPT Threads	#	Programs	OPT Threads
1	FS;EQ	(32,32)	9	AM;SC	(56,17)
2	BT;AP	(50,24)	10	TATP;JBB	(54,69)
3	AS;FMA	(16,56)	11	SM;EQ	(32,32)
4	SW;MG	(73,16)	12	PCA;STM	(48,16)
5	FA;SC	(49,17)	13	MG;PCA	(16,48)
6	GL;BT	(16,50)	14	KM;ART	(24,40)
7	FA;SM	(49,32)	15	MM;SW	(64,73)
8	ART;x264	(40,68)	16	WC;MG	(48,16)
17	KM;CA;X264	(16,33,68)	20	FS;AP;STM	(32,24,16)
18	AS;BT;FR	(16,50,83)	21	AM;WW;PCA	(56,24,48)
19	GA;SM;FA	(64,32,24)			
22	FS;SC;EQ;WW	(32,17,32,24)	24	PCA;AM;AS;ART	(48,48,34,40)
23	AS;AP;BT;FMA	(16,24,50,56)	25	GA;FMA;x264;FA	(64,56,68,49)

(a) coschedule run numbers and corresponding programs

(b) % improvement in TTT

(c) improvement in system utilization

Fig. 9. ADAPT improves TTT and system utilization compared to the default Solaris scheduler. Here, improvement in system utilization = (utilization with ADAPT - utilization with Solaris).

Fig. 10. TTT improvements are relative to the default Solaris scheduler. ADAPT significantly outperforms DINO.

Fig. 11. ADAPT improves performance of all the four memory-intensive programs: FS, SC, EQ, and WW.

with this configuration (see Figure 9(a)). Figure 10 shows the % TTT improvements of both ADAPT and DINO relative to the default Solaris scheduler. As we can see from Figure 10, ADAPT significantly outperforms DINO. Although DINO is also effective in coscheduling of some multithreaded programs, its performance is worse for coscheduling of high lock contention programs.

Since the above existing techniques are based on cache usage, they are very effective for a mix of workloads where half of the threads are memory intensive and the other half are CPU intensive. However, they may not work well on a mix of workloads where all the threads are either CPU intensive or memory intensive. Therefore, we evaluated ADAPT for a mix of four memory-intensive multithreaded programs (FS:SC:EQ:WW) as well as for a mix of four CPU-intensive multithreaded programs (AS:AP:BT:FMA). Figures 11 and 12 show the TTTs of the four programs in each run. As we discussed in Section 3, for memory-intensive and high lock contention programs, ADAPT simultaneously improves memory bandwidth and reduces lock contention. It relieves pressure on the HT module, and consequently reduces paging activity and improves performance. In the second coscheduled run of four programs, we evaluated ADAPT for a mix of four CPU-intensive and high lock contention programs. By assigning appropriate cores configuration and the FF scheduling policy with appropriate time quanta, ADAPT dramatically reduces context-switch rate and improves overall TTT of the programs.

Fig. 12. ADAPT improves performance of all the four CPU-intensive programs: AS, AP, BT, and FMA.

Table X. Normalized Running-Times of the Memory-Intensive Programs in a Coschedule Run

Program	Normalized Running-time	
	Solaris	ADAPT
FS	2.1	1.6
SC	1.6	1.2
EQ	1.5	1.3
WW	1.7	1.4

Table XI. Normalized Running-Times of the CPU-Intensive Programs in a Coschedule Run

Program	Normalized Running-time	
	Solaris	ADAPT
AS	2.3	1.2
AP	1.9	1.2
BT	1.5	1.3
FM	1.9	1.3

Furthermore, as we can see in Figures 11 and 12, ADAPT not only improves performance of programs, it simultaneously reduces variation in their performance. We also computed normalized running-times of the programs to see how fairly the resources are distributed across the four programs in the above mentioned coscheduled runs. Here normalized running-time is computed as the ratio of running-time of the program in the coscheduled run and the running-time of the program in the solo run. As we can see from Tables X and XI, ADAPT not only improves TTT, it also distributes the resources fairly across the coscheduled programs. Thus, ADAPT provides fairness and performs better than the default Solaris scheduler.

In summary the above experiments demonstrate that ADAPT is effective in coscheduling multithreaded programs on multicore systems. By using simple and efficient modern OS performance monitoring utilities, ADAPT continuously monitors the resource usage characteristics of programs, adaptively allocates resources such as

cores, and assigns appropriate memory allocation and scheduling policies. In addition, ADAPT does not require any changes to either the application source-code or the OS kernel. Furthermore, the overhead of ADAPT is negligible for the appropriately chosen monitoring interval. Thus ADAPT scales well with the number of multithreaded programs on machines with large number of cores.

6. RELATED WORK

While coscheduling of programs on a multicore system is a well-studied area, there are only a few works [Bhadauria and McKee 2010] that deal with contention management during the coscheduling of *multithreaded* programs. Bhadauria and McKee [2010] proposed a symbiotic scheduler based on memory hierarchy contention considerations (i.e., last-level cache miss-rate) for coscheduling multithreaded programs on a machine with a small number of cores (eight cores). However, Bhadauria and McKee [2010] pursue a different goal of balancing power and performance. Moreover, one thread per core configuration is used for evaluating the scheduler using the metric of overall throughput per watt.

In Moore and Childers [2012] statistical models are used to predict appropriate number of threads for a multithreaded program when it is running with another multithreaded program on a multicore system for achieving high throughput. It is shown that by varying the number of threads of an application, according to the workload, improvements in throughput can be obtained. This is a good observation but exploiting this observation requires that the applications written to accept the number of threads as an input be modified. In contrast to this work, ADAPT does not require modification of applications. ADAPT always runs multithreaded program with OPT threads. Here, OPT threads is the minimum number of threads of a multithreaded program in its solo run on our multicore machine. Moreover, unlike the above work, ADAPT aims to achieve both low total turnaround times and high system utilization. Most importantly, Moore and Childers [2012] do not exploit application characteristics such as lock contention, scheduling policies, and memory allocation policies that ADAPT exploits. Allocating appropriate number of cores is also a critical factor in coscheduling multithreaded programs that ADAPT considers. One can also view the techniques of Moore and Childers [2012] and ADAPT as complementary techniques that can be potentially integrated to further improve performance.

Other existing contention management techniques [Zhuravlev et al. 2010; Blagodurov et al. 2011; Pusukuri et al. 2011c; Knauerhase et al. 2008; Merkel et al. 2010] are also based upon consideration of memory hierarchy contention factors such as cache usage. However, as we demonstrated in this work, cache usage alone is not enough for coscheduling multithreaded programs on machines with large number cores. We achieved better results by considering characteristics of lock contention and thread latency. Complementary to these works, some researchers [Tam et al. 2007] have developed techniques for coscheduling threads of a single multithreaded program that share data on the same chip. However, these techniques do not address coscheduling of multithreaded programs.

Several researchers [Brecht 1993; LaRowe et al. 1992; Corbalan et al. 2003; VMware 2005; Gamsa et al. 1999; Li et al. 2007] have developed NUMA-related optimization techniques for efficient colocation of computation and related memory on the same node. However, they have not addressed resource contention management in multicore machines. Likewise, Severance and Enbody [1997] present an adaptive scheduling technique for parallel applications based on MPI on large discrete computers. However, this scheduling technique does not address the contention of shared resources when coscheduling multiple programs concurrently. Corbalan et al. [2000] use techniques to allocate processors adaptively based on program efficiency. However, like the above

works, they also do not consider resource contention among programs. McGregor et al. [2005] developed coscheduling techniques using architectural factors such as cache resource usage for coscheduling NAS parallel benchmarks on a quad-core machine. However, each of the workload used in this work is either single-threaded or multithreaded with only two threads. Like the above existing contention management techniques, this technique also will not work for coscheduling multithreaded programs on large multicore machines. Gupta et al. [1991] explored the impact of the scheduling strategies on the caching behavior of the applications. Likewise, Chandra et al. [1994] evaluated different scheduling and page migration policies on a CC-NUMA multiprocessor system.

Unlike the above approaches, by considering appropriate contention factors and using supervised learning techniques for identifying the interference between multithreaded programs, our work provides efficient coscheduling techniques for multithreaded programs on a multicore machine. Several other researchers also explored machine learning and control theory for developing adaptive resource optimization techniques for utility computing [Padala et al. 2007, 2009], network management [Barham et al. 2008], mobile computing [Narayanan and Satyanarayanan 2003], computer architecture [Ipek et al. 2008], and compiler optimizations [Pekhimenko and Brown 2008].

7. CONCLUSIONS

Coscheduling multithreaded programs on a multicore machine is a challenging problem. We presented ADAPT, a framework for effective coscheduling multithreaded programs on multicore systems. ADAPT is based on supervised learning techniques for identifying the effects of the interference between multithreaded programs on their performance. It uses simple modern OS performance monitoring utilities for continuously monitoring the resource usage characteristics of the target programs, adaptively allocating resources such as cores, and selecting appropriate memory allocation and scheduling policies. Moreover, it is an attractive approach as it does not require any changes to either the application source code or the OS kernel. Furthermore, the overhead of ADAPT is negligible with an appropriate choice of the monitoring interval. Thus, ADAPT scales well with the number of multithreaded programs on machines with a large number of cores.

REFERENCES

- BARHAM, P., BLACK, R., GOLDSZMIDT, M., ISAACS, R., MACCORMICK, J., MORTIER, R., AND SIMMA, A. 2008. Constellation: Automated discovery of service and host dependencies in networked systems. Tech. rep. (MSR-TR-2008-67), Microsoft Research.
- BHADOURIA, M. AND MCKEE, S. A. 2010. An approach to resource-aware co-scheduling for cmps. In *Proceedings of the 24th ACM International Conference on Supercomputing (ICS'10)*. ACM, New York, 189–199.
- BIENIA, C., KUMAR, S., SINGH, J. P., AND LI, K. 2008. The parsec benchmark suite: characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT'08)*. ACM, New York, 72–81.
- BLAGODUROV, S., ZHURAVLEV, S., DASHTI, M., AND FEDOROVA, A. 2011. A case for numa-aware contention management on multicore systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX-ATC'11)*. USENIX Association, 1.
- BOYD-WICKIZER, S., MORRIS, R., AND KAASHOEK, M. F. 2009. Reinventing scheduling for multicore systems. In *Proceedings of the 12th Conference on Hot Topics in Operating Systems (HotOS'09)*. USENIX Association, 1.
- BRECHT, T. 1993. On the importance of parallel application placement in numa multiprocessors. In *Proceedings of the USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems* Vol. 4. Sedms'93. USENIX Association, 1.
- CANTRILL, B. M., SHAPIRO, M. W., AND LEVENTHAL, A. H. 2004. Dynamic instrumentation of production systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '04)*. USENIX Association, 2.

- CHANDRA, R., DEVINE, S., VERGHESE, B., GUPTA, A., AND ROSENBLUM, M. 1994. Scheduling and page migration for multiprocessor compute servers. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*. ACM, New York, 12–24.
- CORBALÁN, J., MARTORELL, X., AND LABARTA, J. 2000. Performance-driven processor allocation. In *Proceedings of the 4th Symposium on Operating System Design & Implementation*. Vol. 4. (OSDI'00). USENIX Association, 5.
- CORBALÁN, J., MARTORELL, X., AND LABARTA, J. 2003. Evaluation of the memory page migration influence in the system performance: the case of the sgi o2000. In *Proceedings of the 17th Annual International Conference on Supercomputing (ICS'03)*. ACM, New York, 121–129.
- EYERMAN, S. AND ECKHOULT, L. 2008. System-level performance metrics for multiprogram workloads. *IEEE Micro* 28, 3, 42–53.
- GAMSA, B., KRIEGER, O., APPAVOO, J., AND STUMM, M. 1999. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI'99)*. USENIX Association, Berkeley, 87–100.
- GUPTA, A., TUCKER, A., AND URUSHIBARA, S. 1991. The impact of operating system scheduling policies and synchronization methods of performance of parallel applications. *SIGMETRICS Perform. Eval. Rev.* 19, 120–132.
- HASTIE, T., TIBSHIRANI, R., AND FRIEDMAN, J. H. 2009. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, 2nd ed. Springer Series in Statistics.
- IPEK, E., MUTLU, O., MARTÍNEZ, J. F., AND CARUANA, R. 2008. Self-Optimizing memory controllers: A reinforcement learning approach. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA'08)*. IEEE Computer Society, Washington, DC, USA, 39–50.
- JOHNSON, F. R., STOICA, R., AILAMAKI, A., AND MOWRY, T. C. 2010. Decoupling contention management from scheduling. In *Proceedings of the 15th edition of ASPLOS on Architectural support for Programming Languages and Operating Systems (ASPLOS'10)*. ACM, New York, 117–128.
- KNAUERHASE, R., BRETT, P., HOHLT, B., LI, T., AND HAHN, S. 2008. Using os observations to improve performance in multicore systems. *IEEE Micro* 28, 3, 54–66.
- LAROWE, JR., R. P., ELLIS, C. S., AND HOLLIDAY, M. A. 1992. Evaluation of numa memory management through modeling and measurements. *IEEE Trans. Parallel Distrib. Syst.* 3, 6, 686–701.
- LI, T., BAUMBERGER, D., KOUFATY, D. A., AND HAHN, S. 2007. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'07)*. ACM, New York, 53:1–53:11.
- MCDUGALL, R. AND MAURO, J. 2006. *Solaris Internals*, 2nd Ed. Prentice Hall.
- MCDUGALL, R., MAURO, J., AND GREGG, B. 2006. *Solaris Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris*. Prentice Hall.
- MCGREGOR, R. L., ANTONOPOULOS, C. D., AND NIKOLOPOULOS, D. S. 2005. Scheduling algorithms for effective thread pairing on hybrid multiprocessors. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium - Papers - Vol. 01 (IPDPS'05)*. IEEE Computer Society, USA, 28.1–.
- MERKEL, A., STOESS, J., AND BELLOSA, F. 2010. Resource-conscious scheduling for energy efficiency on multicore processors. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys'10)*. ACM, New York, 153–166.
- MOORE, R. W. AND CHILDERS, B. R. 2012. Using utility prediction models to dynamically choose program thread counts. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'12)*, R. Balasubramonian and V. Srinivasan, Eds. IEEE, 135–144.
- NARAYANAN, D. AND SATYANARAYANAN, M. 2003. Predictive resource management for wearable computing. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services (MobiSys'03)*. ACM, New York, 113–128.
- PADALA, P., HOU, K.-Y., SHIN, K. G., ZHU, X., UYSAL, M., WANG, Z., SINGHAL, S., AND MERCHANT, A. 2009. Automated control of multiple virtualized resources. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys'09)*. ACM, New York, 13–26.
- PADALA, P., SHIN, K. G., ZHU, X., UYSAL, M., WANG, Z., SINGHAL, S., MERCHANT, A., AND SALEM, K. 2007. Adaptive control of virtualized resources in utility computing environments. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys'07)*. ACM, New York, 289–302.
- PEKHIMENKO, G. AND BROWN, A. D. 2008. Machine learning algorithms for choosing compiler heuristics. Tech. rep. MSc. Thesis CS Department, University of Toronto.
- PETER, S., SCHÜPBACH, A., BARHAM, P., BAUMANN, A., ISAACS, R., HARRIS, T., AND ROSCOE, T. 2010. Design principles for end-to-end multicore schedulers. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Parallelism (HotPar'10)*. USENIX Association, 10.

- PUSUKURI, K., GUPTA, R., AND BHUYAN, L. 2011a. Thread reinforcer: Dynamically determining number of threads via os level monitoring. In *Proceedings of the International Symposium on Workload Characterization (IISWC'11)*. IEEE Computer Society, 116–125.
- PUSUKURI, K. K., GUPTA, R., AND BHUYAN, L. N. 2011b. No more backstabbing. . . A faithful scheduling policy for multithreaded programs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'11)*. IEEE Computer Society, Washington, DC, 12–21.
- PUSUKURI, K. K., GUPTA, R., AND BHUYAN, L. N. 2012. Thread tranquilizer: Dynamically reducing performance variation. *ACM Trans. Archit. Code Optim.* 8, 4, 46:1–46:21.
- PUSUKURI, K. K., VENGEROV, D., FEDOROVA, A., AND KALOGERAKI, V. 2011c. Fact: A framework for adaptive contention-aware thread migrations. In *Proceedings of the 8th ACM International Conference on Computing Frontiers (CF'11)*. ACM, New York, 35:1–35:10.
- R. `lm()`, `stepaic()`, `prune()`, `vif()`, `rpart()`, `kknn()`. <http://www.statmethods.net/>.
- SEVERANCE, C. AND ENBODY, R. J. 1997. Comparing gang scheduling with dynamic space sharing on symmetric multiprocessors using automatic self-allocating threads (asat). In *Proceedings of the 11th International Symposium on Parallel Processing (IPPS'97)*. IEEE Computer Society, 288.
- SOLIDDB. IBM soliddb 6.5 (build 2010-10-04). <https://www-304.ibm.com/support/docview.wss?uid=swg24028071>.
- SPECJBB. 2005. <http://www.spec.org/jbb2005>.
- SPECOMP. 2001. <http://www.spec.org/omp>.
- TAM, D., AZIMI, R., AND STUMM, M. 2007. Thread clustering: Sharing-Aware scheduling on smp-cmp-smt multiprocessors. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys'07)*. ACM, New York, 47–58.
- TATP. 2003. IBM telecom application transaction processing benchmark description. <http://tatpbenchmark.sourceforge.net>.
- VIF. Multicollinearity. <http://en.wikipedia.org/wiki/Multicollinearity>.
- VMWARE. 2005. Vmware esx server 2 numa support. white paper. Tech. rep. <http://www.vmware.com/pdf/esx2.NUMA.pdf>.
- YOO, R. M., ROMANO, A., AND KOZYRAKIS, C. 2009. Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system. In *Proceedings of the IEEE International Symposium on Workload Characterization. (IISWC'09)*. IEEE Computer Society, 198–207.
- ZHURAVLEV, S., BLAGODUROV, S., AND FEDOROVA, A. 2010. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the 15th Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS'10)*. ACM, New York, 129–142.

Received June 2012; revised November 2012; accepted November 2012