FACT: a Framework for Adaptive Contention-aware Thread Migrations

Kishore Kumar Pusukuri
University of California,
Riverside, USA.
kishore@cs.ucr.edu

David Vengerov
Oracle Corporation
Menlo Park, USA.
david.vengerov@oracle.com

Alexandra Fedorova
Simon Fraser University
Vancouver, Canada.
fedorova@cs.sfu.ca

Vana Kalogeraki *
Athens University of
Economics and Business
Greece.
vana@aueb.gr

ABSTRACT

Thread scheduling in multi-core systems is a challenging problem because cores on a single chip usually share parts of the memory hierarchy, such as last-level caches, prefetchers and memory controllers, making threads running on different cores interfere with each other while competing for these resources. Data center service providers are interested in compressing the workload onto as few computing units as possible so as to utilize its resources most efficiently and conserve power. However, because memory hierarchy interference between threads is not managed by commercial operating systems, the data center operators still prefer running threads on different chips so as to avoid possible performance degradation due to interference.

In this work, we improved the system's throughput by minimizing inter-workload contention for memory hierarchy resources. We achieved this by implementing FACT, a Framework for Adaptive Contention-aware Thread migrations, which measures the relevant performance monitoring events online, learns to predict the effects of interference on performance of workloads, and then makes optimal thread scheduling decisions. We found that when instantiated with a fuzzy rule-based (FRB) predictive model, FACT achieves on average a 74% prediction accuracy on the new data. In experiments conducted on a quad-core machine running OpenSolaris^{TM}, SPEC-cpu2006 workloads under FACT-FRB ran up to 11.6% faster than under the default OpenSolaris scheduler. FACT-FRB was also able to find the best combination of workloads more consistently than the state-of-the-art algorithms that aim to minimize contention for memory resources on each chip. Unlike these algorithms that based on fixed heuristics, FACT can be easily adapted to consider other performance factors so as to accommodate changes in architectural features and performance bottlenecks in future systems.

*The research of this author has been supported by a gift from SUN, the European Union through the Marie-Curie RTD (IRG-231038) project and by AUEB through a PEVE project.

CF'11, May 3–5, 2011, Ischia, Italy.

Categories and Subject Descriptors

D.4.1 [**Process Management**]: Scheduling

General Terms

Performance, Measurement, Algorithms

Keywords

Multicore, Scheduling, Operating Systems, Supervised Learning

1 Introduction

In recent years, there has been an increase of computational power demand to satisfy modern user needs and solve complex scientific problems such as genome analysis, molecular analysis, weather prediction, video encoding, etc. In the quest for huge computational power, hardware engineers are building multi-core systems because multi-core processors offer significantly greater parallelism, and performance relative to uniprocessor systems. However, multi-core architectures bring new challenges to the system software (compilers, OS). For example, applications running on different cores may require efficient inter-process communication mechanisms, a shared-memory data infrastructure, and synchronization primitives to protect shared resources. Efficient code migration is also a challenging problem in such systems[7].

Modern operating systems (OS) do not effectively make use of the fact that cores often share resources within the memory hierarchy such as cache, prefetcher, memory bus, memory controller, etc. Therefore, the OS may fail to fully exploit the capabilities of the system. The operating systems must therefore evolve in order to fully support the new multi-core systems with appropriate process scheduling and memory management techniques. There are two important challenges [3] that need to be addressed: 1) the OS needs to understand how different workloads utilize resources within the memory hierarchy and the effect of resource sharing on performance of all workloads; 2) the OS needs an efficient way to migrate workloads between the cores so that workloads don't hurt each other's performance when sharing such resources.

As a step toward addressing these challenges, we developed a Framework for Adaptive Contention-aware Thread migrations (FACT), which is based on machine learning (more specifically, supervised learning) techniques. FACT trains a statistical model to predict contention for memory resources between threads based on performance monitoring events recorded by the OS (see [28, 2] for a list of possible events). The trained model is then used to dynamically schedule together threads that interfere with each

other's performance as little as possible. This paper explains the development and evaluation of FACT.

Instruction per Cycle (IPC) is a standard way of measuring performance of a workload. The following is the basic idea behind FACT: it considers some number of possible thread migrations, and for each migration it predicts the change in the IPC of all threads affected by the migration. Predictions are made using a performance model that is learned online based on hardware performance counter measurements taken in real-time. If some migrations are predicted to increase the average IPC of all affected threads, then the migration resulting in the largest IPC increase is performed; otherwise, no migration takes place at this cycle and the next cycle of observing the relevant performance monitoring events begins.

We implemented FACT on a quad-core Intel x86 based server machine running OpenSolaris (2009.06). OpenSolaris libcpc (3LIB) [29] and libpctx (3LIB) [30] interfaces were used to read and program performance monitoring counters. We developed a variety of statistical models for predicting the IPC of co-scheduled workloads: linear regression models(LR)) [33], fuzzy rule-based models (FRB) [21], decision tree (Rpart) models [40], and K-nearest neighbor models (k-nn) [27]. The FRB model had the highest prediction accuracy of 74% in our experiments and was chosen as the preferred statistical model to be used inside the FACT framework. Our experimental results demonstrate that the FACT-FRB combination resulted in a speedup of up to 11.6% on the considered SPECcpu2006 [38] benchmarks.

When comparing FACT to another state-of-the-art contention management scheduling algorithm [4], we found that FACT outperforms it by finding the best combination of workloads more consistently. Moreover, that algorithm is based on a fixed heuristic that works well only when the whole complexity of contention for shared resources can be captured with a single value, while FACT can dynamically learn relationships between any number of performance-affecting factors on any target architecture. Therefore, we believe FACT will be able to better evolve despite changes in processor architecture and in resulting performance bottlenecks. Moreover the overhead of FACT is negligible.

This paper focuses on single-threaded workloads, and therefore the term workload will always mean a *single-threaded* workload unless explicitly stated otherwise.

The following are the main contributions of this work:

- Identified the best predictors (based on the cache usage data) for predicting the IPC of co-scheduled workloads by fully exploiting performance monitoring events
- Developed several statistical models that use performance monitoring events as inputs and identified the best model for predicting the IPC
- Developed adaptive migration techniques to reduce total running-times of workloads
- Showed the usage of supervised learning techniques to capture the complexity of modern multi-core systems.
- Provided the possibility of extending FACT for predicting any thread resource usage characteristics (performance, power etc.) by adding the relevant performance monitoring events (such as those reflecting the usage of CPU, Main Memory, Disk, IO, etc.)

This remainder of this paper is organized as follows: Section 2 gives the motivation and Section 3 gives a high-level description of the FACT framework. The process of developing a statistical model for predicting IPC of co-scheduled workloads is described in Section 4. Section 5 describes the implementation of FACT using the OpenSolaris libraries libpctx and libcpc. Section 6 presents evaluation results for the FACT framework and its comparison with a state-of-the-art thread allocation algorithm. Finally, the related work is described in Section 7 and conclusions and the future work are discussed in Section 8.

2 Motivation and Background

Multi-core processors allow running multiple threads simultaneously, but contention for caches and other shared resources, such as memory controllers and prefetchers, becomes an obstacle in achieving a proportional improvement in the system's throughput.

Previous work observed that on modern Intel and AMD systems the degree of contention for shared resources can be explained by the relative *memory-intensity* of threads that share resources [4]. In that work, threads were classified as *memory-intensive* or *CPU-intensive*. Memory-intensity was approximated by a thread's relative misses per instruction (MPI): memory-intensive threads have a higher MPI than the CPU-intensive threads. That work found that in order to significantly reduce contention for shared resources the scheduler must run memory-intensive threads (those with high MPI) on separate core groups[1], thus preventing them from competing for shared resources.

Even though the MPI heuristic, used to determine memory-intensity, does not directly capture how much a thread would suffer from cache contention alone (for instance, a streaming application would have a high MPI but no cache reuse and so it will not suffer from cache interference), it still turned out to be a rather good heuristic for predicting *overall* memory hierarchy resource contention, because on modern systems the contention is dominated by memory controller or memory bus contention, and MPI is a good approximation of how intensely an application uses these resources [4].

As will be shown in Section 6, the degree of thread interference cannot be predicted accurately by using just MPI alone, and our proposed FACT framework addresses the shortcomings of single-heuristic algorithms that use only MPI by considering other factors as well. However, for the sake of providing a sufficient background on the state-of-the-art, we first show why simple MPI-based algorithms do significantly better than naïve schedulers that ignore contention for memory hierarchy resources.

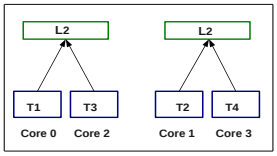

Figure 1: A typical quad-core system.

Consider the following experiment. Figure 1 shows a typical quad-core system, core-0 and core-2 are sharing one last-level cache and other associated memory resources (a prefetcher and a memory controller on our experimental system), and core-1 and core-3 are sharing another last-level cache and the associated set of memory resources. Let's assume that threads T1 and T3 are memory-intensive and threads T2 and T4 are CPU-intensive. We will next show that keeping the two memory-intensive threads (T1 and T3) on the same core group results in a performance degradation for both T1 and T3, and as a result degrades the overall system throughput.

[1] A core group is a group of cores sharing various resources, such as caches, prefetchers, memory controllers, system request queue controllers, etc.

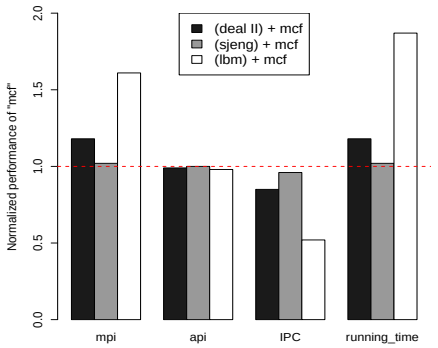

Figure 2: Normalized performance (MPI, API, IPC, and running-time) of *mcf* with different co-runners

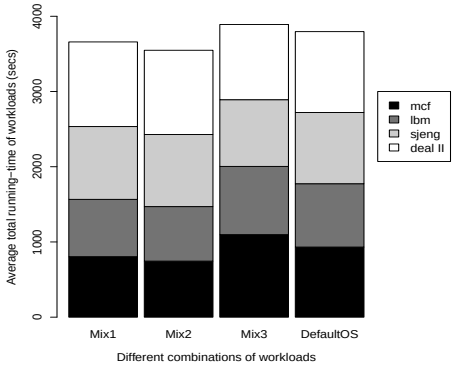

Figure 3: Average total running times of four workloads (*mcf, lbm, sjeng, deal II*) under different bindings and also under the default OS scheduler.

In our example we will use four SPECcpu2006 workloads (*mcf, lbm, deal II*, and *sjeng*). Table 1 shows MPI, API (last-level cache Access per Instruction), and IPC values in their solo-runs, where each workload was executed ten times without any other co-running workloads using the default OS scheduler (OpenSolaris 2009.06 scheduler). The workloads *mcf* and *lbm* have relatively high MPI – so they are deemed memory intensive workloads in this group; they stand for threads T1 and T3 in Figure 1. The workloads *deal II* and *sjeng* are CPU-intensive; they stand for threads T2 and T4.

Table 1: Workloads' MPI, API, IPC and running-time (secs) when running solo under the default OS scheduler.

Workload	**MPI**	**API**	**IPC**	running-time
mcf	0.0221	0.14	0.24	589
lbm	0.0177	0.48	0.51	586
sjeng	0.0005	0.01	1.06	808
deal II	0.0007	0.02	1.02	926

Figure 2 shows the impact of various co-running worklads on the performance of *mcf*. (The workloads were bound to cores using the "pbind" command for the duration of the experiment.) The MPI, API, IPC, and running-time of *mcf* in the presence of other workloads are normalized by its corresponding solo metrics. We observe that *mcf* suffers more (higher MPI, lower IPC and higher running-times) when it is running together with *lbm* than with the other workloads. This is because both *mcf* and *lbm* are memory-intensive, so they hurt each other when competing for shared resources. While this is only a single example showing that memory-intensity can be used to predict the degree of contention, previous work presented extensive data for a wide range of workloads [4].

Based on this observation, researchers proposed schedulers that separate memory-intensive threads on different core groups [4, 5, 6, 25]. In other words, these schedulers combine threads in a way that minimizes total contention. We illustrate how this is done by continuing our example with the (*mcf, lbm, sjeng, deal II*) workloads.

Table 2: Different combinations of workloads (*mcf, lbm, sjeng,* and *deal II*). In every combination (except Default OS), half of the workloads are bound to cores (0,2) and other half are bound to cores (1,3).

Combination	**cores (0,2)**	**cores (1,3)**
Mix1	(*mcf, deal II*)	(*lbm, sjeng*)
Mix2	(*mcf, sjeng*)	(*lbm, deal II*)
Mix3	(*mcf, lbm*)	(*sjeng, deal II*)
Default OS	Run the workloads with Default OS scheduler.	

Table 2 shows different combinations of these applications, and Figure 3 shows the impact of the workload-mix on the total running-time of those workloads. In Mix1 and Mix2 the memory-intensive workloads *mcf* and *lbm* were run (using the "pbind" command) on different core groups. As shown in Figure 3, separating memory-intensive workloads (*mcf* and *lbm*) on different core groups is better for the overall performance of the workloads. Therefore, we can say that (*mcf, sjeng*) and (*lbm, deal II*) are the best pairs, (*mcf, deal II*) and (*lbm, sjeng*) are the good pairs, and (*mcf, lbm*) and (*sjeng, deal II*) are the worst pairs.

While this example, as well as experimental data presented in other studies [4, 6], show that MPI-based contention management algorithms can reduce resource contention, we found that they do not always find the optimal thread combinations for maximizing IPC of all workloads. To address this shortcoming, additional performance factors must be considered, but the earlier contention-aware scheduling algorithms are not designed to do so, because they are fundamentally built for a single heuristic. We address this problem by designing the FACT framework. FACT can take into account any number of performance factors and then it dynamically learns how these factors interact and affect workload performance.

To demonstrate the effectiveness of FACT we show how it can find better combinations of threads by considering an additional heuristic: last-level cache *Accesses Per Instruction* (API). API was chosen because it approximately captures an application's *cache-sensitivity* – how much an application suffers if a competing application evicts its cached data – something that the MPI heuristic does not reflect. Intuitively, if an application frequently accesses the cache, it reuses cached data and so it would suffer if this data is prematurely evicted. While we do not claim that API is the only additional factor needed to improve contention management, we use it to demonstrate that FACT can be effectively used for modeling effects of multiple interacting factors. For example, when we used the (*milc, lbm, mcf, sjeng*) workload, FACT outperformed the state-of-the-art MPI based algorithm by finding best combination of workloads more consistently (detailed results are presented in the Section 6).

Thus, FACT is able to improve on performance of state-of-the-art algorithms. Moreover, FACT shows the usage of supervised learning techniques to capture the complexity of modern multi-core systems.

3 The FACT Framework

In order to capture a variety of factors that may determine the degree of contention on multicore processors, we based the FACT framework on the idea of machine learning, more specifically supervised learning, where a set of sample input-output values is first observed and then a statistical model is trained to predict similar output values when similar input values are observed.

The FACT framework includes two phases:

1. Training a statistical model to predict the IPC of co-scheduled workloads based on the relevant input variables.

2. Performing thread migrations based on the IPC of possible workload combinations as predicted by the statistical model.

The first phase can be performed offline using a training set of workloads. Alternatively, it can also be performed continuously online as real thread migration data is observed. We will focus on the offline version in this paper since it allows us to judge the quality of the IPC forecasts after a given amount of training data. The prediction accuracy of the offline model can then be treated as a lower bound on the accuracy that can be achieved after the amount of observed data during an online deployment exceeds the amount of the training data used for the offline evaluation. While the offline model can be employed during system installation by training, for instance, on workloads in a standard benchmark suite, an online model could be used directly on the workload of interest and would apply to any workload that takes more than a few seconds to run (a few seconds are necessary to train the model).

A graphical depiction of FACT for a quad-core machine (the one we used for our experiments) is shown in Figure 4 and Algorithm 1 describes the Phase-2 of FACT.

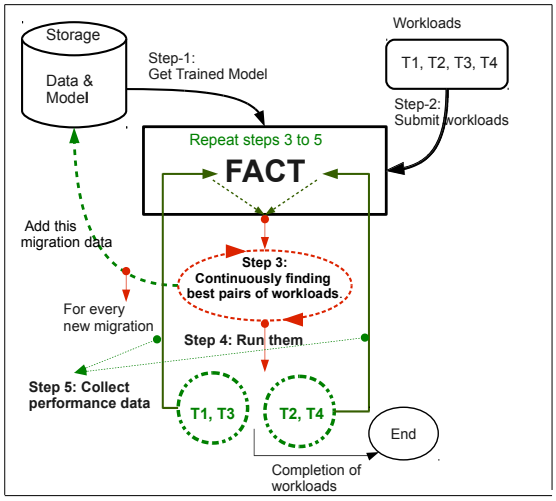

Figure 4: A graphical depiction of FACT for a quad-core machine.

4 Statistical Model for IPC Prediction

This section starts by introducing the workloads that were used for developing the statistical models that can predict IPC of co-scheduled workloads inside the FACT framework. Then the process of deriving the most important predictors for these statistical is described. Finally, the considered statistical models are presented and compared in terms of IPC prediction accuracy.

4.1 Experimental Workloads

SPECcpu2006 and SPECcpu2000 benchmark suites were used for developing the IPC prediction models. These benchmark suites are designed for stressing the system's processor, memory subsystem and compiler. We chose 12 workloads from SPECcpu2006 and 5 from SPECcpu2000 that we could easily compile on our machine, which included both memory-intensive and CPU-intensive workloads. Table 3 lists these benchmarks.

4.2 Performance Events and Predictors

We derived the predictors shown in Table 4 from L2 cache misses, L2 cache accesses and instruction counts on each core. L2 cache is

Algorithm 1: For the Phase-2 of FACT

```
for each time step do
    for each possible pair of core groups where each group of
    cores shares the last-level cache do
        MaxIPC=0;
        t* = null;
        for each thread t in the considered pair of core groups
        do
            Predict IPC_t the new average IPC of all threads if
            thread t were migrated from one core group to the
            other;
            if IPC_t > MaxIPC then
                MaxIPC = IPC_t;
                t* = t;
            end
        end
        if MaxIPC > IPC of the current allocation then
            Migrate thread t* to the other core group;
        end
    end
end
```

Table 3: Benchmarks

Suite	Workloads
SPEC CPU 2006	*mcf, lbm, sjeng, bzip2, soplex, namd, hmmer, deal II, povray, gcc, xalancbmk, omnetpp*
SPEC CPU 2000	*wupwise, vpr, art, twolf, gap*

the last-level cache on our system. These predictors were chosen as a compilation of events used by other researchers in the field and from our own experience [24]. Let us say our goal is to predict the new IPC of thread T1 if it were swapped with the thread T4 in Figure 4. The possible inputs (predictors) for the prediction model are described in the Table 4. The predictors are the performance events of the co-threads (the current co-thread T3 and the future co-thread T2) of the thread T1. Therefore, each data point is a 9-tuple containing eight predictors shown in Table 4 and the observed T1_T2_ipc (IPC of thread T1 when co-scheduled with T2) as the target parameter. Even though in the examples in this paper we assume two cores per group (and thus two co-scheduled threads) since this is the configuration of our experimental system, there is nothing in the framework that prevents it from being used on systems with more than two cores per group.

4.3 Data Collection

We used a total of 16 workloads and divided them into 4 groups, each of which contained 4 different workloads (some memory-intensive and some CPU-intensive), as shown in Table 5. Since our test machine had 2 core groups (with each group having 2 cores) and we naturally ran one thread per core so as to isolate effects of cache interference from processor sharing issues, we could run a total of 3 different arrangements of 4 threads: T1 running with T2, with T3 and with T4 in the same core pair (core group). For each arrangement of 4 threads, the values of MPI, API and IPC were measured for each thread over a 5-second interval. Then, for each group of 4 workloads, 12 different data points were formed from the raw data: 3 data points with the independent variables being T1_T2_ipc, T1_T3_ipc, T1_T4_ipc, 3 more data points with the independent variables being T2_T1_ipc, T2_T3_ipc, T2_T4_ipc, etc. For each independent variable, the possible predictor variables (listed in Table 4) were computed from the IPC, MPI, API values of each thread. A total of 48 data points were collected from the 4 groups of 4 workloads, which were divided into 4 sets (or folds) of

Table 4: Possible predictors (or inputs) to statistical models. Threads T1, T3 are running initially on core 0 and 2 (one core group) and threads T2, T4 are running in another core group. The goal is to predict the IPC of the thread T1 if it were swapped with the thread T4 and start running with T2, with which it has never ran before.

Predictor	Description
T3_mpi	average (avg) misses per instruction of first co-thread T3 not including the runs with thread T2.
T3_api	avg accesses per instruction of first co-thread T3 not including the runs with thread T2.
T2_mpi	avg misses per instruction of second co-thread T2 not including the runs with thread T1.
T2_api	avg accesses per instruction of second co-thread T2 not including the runs with thread T1.
T1_mpi	avg misses per instruction of the thread T1 not including the runs with thread T2.
T1_api	avg accesses per instruction of the thread T1 not including the runs with thread T2.
dmpi	T2_mpi - (average mpi of the previous co-runners of T1 not including T2).
T1_ipc	avg IPC of thread T1 not including the runs with thread T2.

data, each set containing data that came from one unique group of 4 workloads.

Table 5: Four sets of different workloads

1. *sjeng, omnetpp, soplex, wupwise*
2. *namd, hmmer, xalancbmk, vpr*
3. *mcf, twolf, bzip2, gap*
4. *lbm, deal II, gcc, art*

4.4 Finding Important Predictors

The most important predictors from Table 4 were selected in order to balance the prediction accuracy and the overhead of monitoring extra performance counters. We used the methodology [24] and the R language described in Figure 5 for selecting the important predictors.

Step 1: Develop a multiple linear regression model with all reasonable predictors for the target parameter (i.e. IPC of co-scheduled threads)

Step 2: Assess the model with ``leave-one-out cross-validation`` (CV) test and also against an independent test-data set, and measure prediction accuracy with any reasonable metric.

Step 3: Rank the accuracy of predictors using "Step-wise Regression" and ``Relative Importance Measures" (or any other statistical technique designed for this purpose).

Step 4: Perform model selection by choosing the most effective predictors, develop a model, and assess it as described in Step 2.

Step 5: Based on the trade-off between prediction accuracy and the number of predictors, either choose the model obtained in Step 4 or choose different predictors based on their ranking in Step 3.

Figure 5: A methodology for finding most important predictors.

R is a programming language for statistical computing [33] and we used R step-wise regression (stepAIC() method) [39] and all subsets regression [34] to find the best predictors. Referring to the example of Section 4.2, these techniques selected the following 6 predictors for *T1_T2_ipc: T1_ipc, T1_mpi, T1_api, T2_mpi, T2_api,* and *dmpi*. To balance prediction accuracy and number of predictors, we further applied R relative importance measures (RIM) [31, 32] method to find the most important predictors among the six selected in the previous step. RIM ranks the predictors in terms of their effectiveness by computing the R-square value for each predictor. As shown in Figure 6, RIM suggested the following 4 predictors as the best predictors of *T1_T2_ipc: (T1_ipc, T1_mpi, T1_api, dmpi)*.

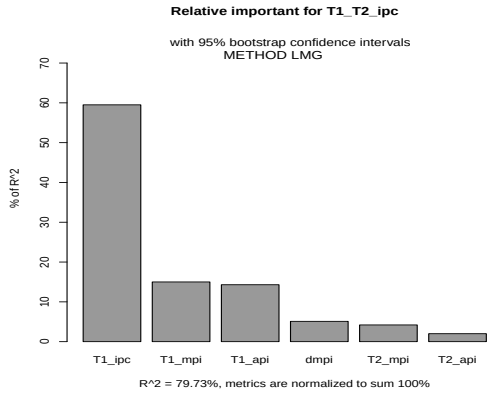

Figure 6: Most Important Predictors of T1_T2_ipc.

4.5 Developing Statistical Models

Using the most important predictors (inputs to the models) shown in Figure 6, we developed a linear regression model (LR), a fuzzy rule-based model (FRB), a decision tree model (Rpart), and a K-nearest neighbor (Knn) model.

4.5.1 Linear regression (LR) model

A linear regression model [27] assumes that the regression function $E(Y|X)$ is linear in its inputs X_1,...,X_p. In our case, the inputs are the chosen predictors and the target parameter is T1_T2_ipc. Linear models are simple and often provide interpretable description of how the inputs affect the output. They can sometimes outperform fancier nonlinear models (which are prone to over-fitting the training data) on extrapolation tasks, especially in situations with a small number of training cases, low signal-to-noise ratio or sparse data.

We used the R lm() method [33] to develop the linear regression models for predicting T1_T2_ipc. The general form of a linear regression model is shown in Equation 1.

$$T1\_T2\_ipc = a_0 + a_1 \cdot T1\_ipc + a_2 \cdot T1\_mpi + a_3 \cdot T1\_api + a_4 \cdot dmpi \qquad (1)$$

4.5.2 Fuzzy-rule based (FRB) model

An FRB is a function f that maps an input vector $x \in \Re^K$ into a scalar output y. We used the following common form [21] of the fuzzy rules:

Rule i: IF (x_1 is A_1^i) and (x_2 is A_2^i) and ... (x_K is A_K^i)
THEN ($y = p^i$),

where x_j is the jth component of x, A_j^i are fuzzy categories used in rule i and p^i are the tunable output parameters. The output of the FRB $f(x)$ is a weighted average of p^i:

$$y = f(x) = \frac{\sum_{i=1}^{M} p^i w^i(x)}{\sum_{i=1}^{M} w^i(x)}, \qquad (2)$$

where M is the number of fuzzy rules and $w^i(x)$ is the weight of rule i computed as $w^i(x) = \prod_{j=1}^{K} \mu_{A_j^i}(x_j)$, where $\mu_{A_j^i}(x_j)$ is a *membership function* taking values in the interval [0,1] that determines the degree to which an input variable x_j belongs to the fuzzy category A_j^i. A separate rule is used for each combination of fuzzy categories A_j^i, which should jointly cover the space of all possible values that the input vector x can take. Therefore, each parameter p^i gives the output value of the FRB when the input vector x completely belongs to the region of the state space described by the fuzzy

categories A_j^i of rule i. Since some or all of the fuzzy categories can overlap, several p^i usually contribute to the rulebase output, with their contributions being weighted by the extent to which x belongs to the corresponding regions of space. An example of a fuzzy rule in our domain is:

IF (T1_ipc is high) AND (T1_api is high) AND (T1_mpi is low) AND (dmpi is low) THEN T1_T2_ipc is high.

4.5.3 Rpart (Decision Tree)

Recursive partitioning creates a decision tree for predicting a categorical (classification tree) or continuous (regression tree) outcome [40]. The goal in decision tree learning [41] is to create a model that predicts the value of a target variable based on several input variables. Each interior node corresponds to one of the input variables; there are edges to children for each of the possible values of that input variable. Each leaf represents a value of the target variable given the values of the input variables represented by the path from the root to the leaf. A tree can be learned by splitting the source set into subsets based on an attribute value test. This process is repeated on each derived subset in a recursive manner called recursive partitioning. The recursion is completed when the subset at a node all has the same value of the target variable, or when splitting no longer adds value to the predictions. We used a regression tree based recursive partitioning tool in R called rpart() [27, 40] to develop this model.

4.5.4 K-nn

The K-nearest neighbours algorithm (K-NN) [35] is a technique for classifying objects based on closest training examples in the feature space. K-NN is a type of instance-based learning, or lazy learning where the function is only approximated locally and all computation is deferred until classification: an object is classified by a majority vote of its neighbors, with the object being assigned to the class most common amongst its K nearest neighbors.

We used a nearest-neighbor method [27] that forms the output $Y(x)$ as a weighted average of outputs in the training set for which the inputs are closest (based on Euclidean distance) to x:

$$Y(x) = \frac{1}{k} \sum_{x_i \in N_k(x)} y_i, \tag{3}$$

We developed the K-NN learning model using the R kknn() method [36].

4.6 Model Assessment

This section shows evaluation results for the models developed in Section 4.5. The performance of each model is assessed by its prediction accuracy on independent test data. In order to make the best use of the available data, we conducted 4-fold cross-validation tests [27], where each model was trained on 36 data points derived from 3 groups of workloads in Table 5 and predictions were made on the 12 data points derived from the fourth group of workloads in Table 5, where each group of workloads was sequentially chosen as the testing group. The *prediction accuracy (PA)* of each model was computed as follows:

$$PA = (1 - MAPE) * 100. \tag{4}$$

where MAPE (Mean Absolute Percentage Error) [37] is defined as follows:

$$MAPE = \frac{1}{N} \sum_{i=1}^{N} \frac{|M_i - m_i|}{|m_i|} \tag{5}$$

where N is the number of test data points, M is the modeled (predicted) value and m is the measured value.

We have also tested the models by training them on a large data set (obtained by forming 12 different subsets of the 12 training workloads and then obtaining 12 data points from each subset) and computed their prediction accuracy using Equation 4. We found that an increase in the size of the training data set improved the prediction accuracy of all considered models. Figure 7 shows the average prediction accuracy of the models. The FRB model achieves the best prediction accuracy **70%** on the small training data set and **74%** on the large training data set. In all experiments below, FACT was combined with the FRB model for predicting the IPC of co-scheduled threads.

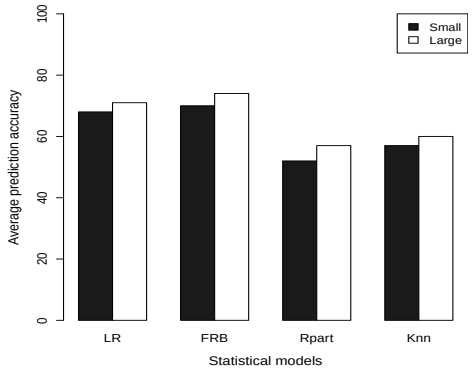

Figure 7: Average prediction accuracy of the models with small and large training data sets. LR: Linear Regression, FRB: Fuzzy Rule-Base, Rpart: Decision Tree, and K-nn: K-nearest neighbor (k = 1).

5 Implementation of FACT

The first step in implementing FACT is to collect the performance statistics (MPI, API, IPC data) of the threads running on a multi-core system. There is an efficient way for doing this in OpenSolaris by using its libpctx library [30] and libcpc(3LIB) library [29]. Functions in the libpctx library provide a simple means to access the underlying facilities of proc(4) to allow the controlling process to manipulate the state of the controlled process. This library is primarily for use in conjunction with the libcpc(3LIB) library. Used together, these libraries allow developers to construct tools that can manipulate CPU performance counters in other processes. The cputrack(1) utility is an example of such a tool.

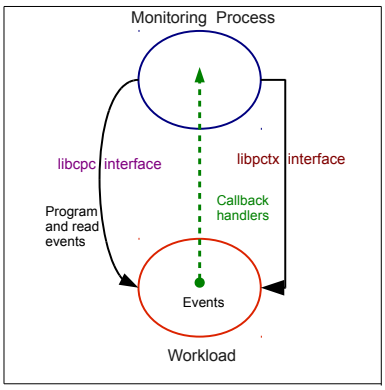

Figure 8: Collecting the performance counters using libpctx and libcpc library interfaces.

However, since we need to read and program performance counters for four different workloads at a time, we developed a utility (multiplexing the access to the library interfaces) as a part of FACT to access performance monitoring events of four processes at a time.

As shown in Fig 8, the library libpctx interface is used to hold the running workloads and then the library libcpc is used to access the performance monitoring counters (cache usage events and CPU events). If four workloads are running on a quad-core system, then there are four child processes to hold the four workloads. These child processes access their performance monitoring counters, collect the MPI, API and IPC data of the workloads and write this data into a pipe together with a sequence number for the workload. The parent process uses this sequence number and maps the workloads with their corresponding data.

6 Evaluation of FACT

Recent work on contention-aware scheduling showed that simple heuristic-based algorithms [25, 6, 4] can find very good contention-aware schedules. In this section, we compare FACT to one such algorithm, Distributed Intensity (DI) [4]. DI measures threads' MPI online, sorts them by the MPI and then pairs threads on cores sharing the last-level cache by taking one thread from the top of the list (a thread with a low MPI) and one thread from the bottom of the list (a thread with a high MPI). This ensures that the miss rate is spread evenly across caches.

6.1 Single-step predictions

Performance comparisons between FACT, the DI algorithm and the default OpenSolaris scheduler were conducted using the following thirteen SPECcpu2006 workloads: *mcf, lbm, sjeng, deal II, bzip2, omnetpp, namd, soplex, hmmer, gcc, xalancbmk, povray, milc*. Out of these workloads, 10 different test sets were formed, which are listed in Table 6. While choosing the workloads for the test sets, we made sure that every test set is a combination of memory-intensive and CPU-intensive workloads. Experiments reported in this section were conducted on Xeon server, whose configuration is described in Table 7.

Table 6: Testing workload sets.

1. (*mcf, lbm, sjeng, deal II*)
2. (*namd, soplex, lbm, sjeng*)
3. (*omnetpp, deal II, lbm, sjeng*)
4. (*lbm, milc, mcf, sjeng*)
5. (*lbm, milc, mcf, deal II*)
6. (*soplex, milc, xalancbmk, omnetpp*)
7. (*xalancbmk, lbm, soplex, omnetpp*)
8. (*lbm, milc, deal II, soplex*)
9. (*soplex, milc, lbm, sjeng*)
10. (*mcf, lbm, milc, namd*)

As was described in Section 4.6, the quality of the IPC prediction for the considered statistical models was shown to improve if a large training data set was used. Therefore, in this section we trained FACT using a large data set of 240 points, which was obtained for each test set by taking 20 different combinations of 4 threads and then running them using 3 possible thread pairings to get 12 data points for each such combination as described in Section 4.3.

Table 7: Configuration of the Testing Machine

Component	Details
Hardware	Quad-core system: two dual-core Intel (r) Xeon (r) CPUs 5150 (2.66GHz), (2 * 4) MB L2-cache, 6GB RAM
Operating System	OpenSolaris 2009.06

The following methodology was used to compare FACT with the DI algorithm. Each algorithm was evaluated on each of the 10 test sets shown in Table 6. For each test set, the FACT framework (instantiated with the FRB predictive model) was first trained on the corresponding training set. The training process consisted of making 10000 passes through the training data set, and for each data point x, the tunable parameters p^i of the FRB were adjusted using the following procedure: $p^i \leftarrow p^i + r(\hat{y} - f(x))\frac{\partial f(x)}{\partial p^i}$, where r is the learning rate that was fixed at 0.01, $\hat{y}$ is the actually observed IPC of the thread in question, $f(x)$ is the predicted IPC using the latest set of parameters p, and $\frac{\partial f(x)}{\partial p^i} = \frac{w^i(x)}{\sum_{i=1}^{M} w^i(x)}$ is the partial derivative of $f(x)$ with respect to the parameter p^i.

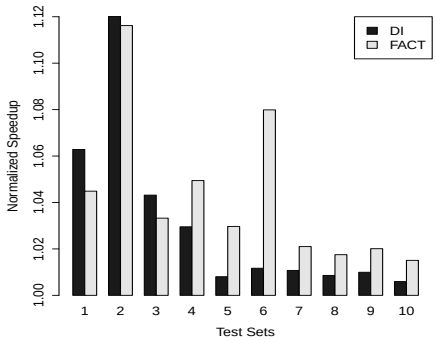

Figure 9: Normalized speedup of DI and FACT relative to OS, computed as the average running time of the test set threads under OS divided by their average running time under DI or FACT.

We would like to point out once again that the performance of the FACT-FRB algorithm in the above setup is a lower bound on the performance that can be achieved during an online deployment, where the IPC prediction model will keep improving as more data is observed and the model parameters keep getting adjusted. A newly observed data point during an online deployment can only be presented once to the model, and the model parameters can be adjusted a little to account for this data point. This one-time adjustment of the 16 parameters for the FRB with 4 input variables described in Section 4.4 requires around 100 floating point operations, and hence the *overhead* of using FACT during an online deployment is *negligible*.

In order to make sure that the test results were not biased by the initial thread allocations chosen by the OS, we explicitly bound threads to cores to create all three possible allocations of the four workloads in the test set, and for each allocation, the relevant performance counters were sampled over a five-second interval, and then the algorithm (FACT or DI, whichever one was being evaluated) was allowed to suggest a thread migration. After that, the suggested thread migration (if any) was implemented, threads were re-bound to their new cores and then ran to completion. This procedure was repeated 10 times for each initial thread allocation.

For each test set, our experiments showed that the average running time of the test set threads under the default OS was larger than under either DI of FACT. Figure 9 shows the normalized speedup of FACT and DI relative to the default OS, computed as the average running time of the test set threads under OS divided by their average running time under DI or FACT. As one can see, the SPECcpu2006 workloads in our test sets ran up to 11.6% faster under FACT than under the default OpenSolaris scheduler. Our results also show that the speedup under FACT can be on average 2% larger than under DI. Actually, 2% is significant if we consider raw performance, i.e., FACT on average *saves* around 100 seconds of running-time over DI. We would get even further improvement on the system throughput if the applications were repetitive.

In order to understand why the DI algorithm can sometimes slightly outperform FACT, we looked at each assignment of threads suggested by these algorithms on our 10 test sets and organized them in Table 8, which shows the quality of thread pairs suggested by each algorithm. There are 3 possible assignments of 4 test threads to 4 cores, which were categorized as Best/Good/Worst according to

the average running time of threads under each assignment. As was described above, each algorithm made a total of 30 suggestions for each test set (10 repetitions for each of the 3 possible initial thread assignments) and hence the total number of Best, Good, and Worst pairs adds up to 30 for each test set for each algorithm. As one can see, the DI algorithm suggests the Best thread assignment in 3 test sets and suggests only the Good assignment in 7 tests sets, while the FACT-FRB algorithm suggest the Best assignment almost all the time. The small variability in assignments suggested from a given initial thread allocation is due to fluctuations of API and MPI during the measurement process, which leads to the average measured API and MPI values for each thread over a 5-second interval to differ between successive repetitions of the experiment.

As the authors of DI explained, DI does best in workloads where the number of memory-intensive threads equals to the number of core groups (data presented in [4]) – in this case DI trivially isolates each memory-intensive thread on its own core group (e. g. first three testing sets). However, when there are more memory-intensive threads than core groups, the DI algorithm cannot find the best thread assignment, as was the case for test sets 4 through 10 in Table 8. As an example of what happens in such cases, consider test set 4: (*lbm, milc, mcf, sjeng*), which has the corresponding MPI values (0.0177, 0.0086, 0.0221, 0.0005) and API values (0.48, 0.42, 0.14, 0.01). As expected, the DI algorithms pairs up the highest-MPI workload *mcf* with the lowest-MPI workload *sjeng* (and correspondingly pairs up *lbm* with *milc*), which happens to give a longer running time than when *mcf* is paired up with *milc* (and *lbm* with *sjeng*). A possible reason for this happening might be that both *lbm* and *milc* have a high *cache-sensitivity*, which is correlated with the API value as shown in [4]. As a result, the negative impact that *lbm* experiences when paired up with *milc* is larger than the one experienced by *mcf* when paired up with *milc*, and so it is better to pair up the less cache-sensitive *mcf* with *milc* and then pair up the more cache-sensitive *lbm* with a very low-MPI *sjeng*, which does not produce any noticeably negative impact on any co-scheduled workload. Since FACT uses both MPI and API as predictors, it can learn the cases when it is important to trade off high MPI vs. high API for co-scheduled workloads.

Table 8: Quality of thread pairs suggested by DI and by FACT

Test Set	DI			FACT		
	Best	Good	Worst	Best	Good	Worst
1	28	2	0	26	3	1
2	30	0	0	28	2	0
3	30	0	0	28	1	1
4	0	30	0	27	3	0
5	0	30	0	30	0	0
6	0	30	0	30	0	0
7	0	30	0	30	0	0
8	0	30	0	30	0	0
9	0	30	0	26	4	0
10	0	30	0	28	2	0

Thus, we believe that the approach used in FACT could be more portable to future systems, especially as systems evolve and factors affecting contention change. For instance, if factors such as prefetching or burstiness of memory accesses also become important contention factors, then these new metrics could be easily incorporated into the IPC prediction model used by FACT, while the existing heuristic-based algorithms may stop working. Furthermore, FACT can be adapted to take into account other factors, such as the positive effects of co-operative data sharing, while heuristic-based approaches may be too rigid to incorporate additional factors. That is why, we believe, the FACT approach is an important contribution to the area of contention-aware scheduling.

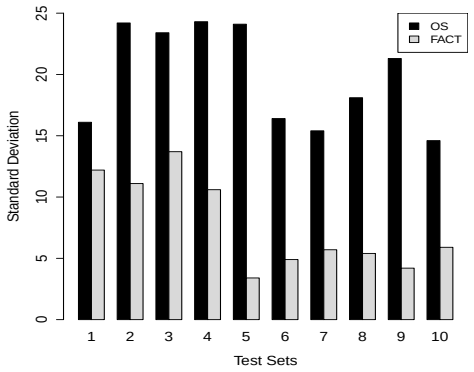

Figure 10: Standard deviation of the workload running times under FACT and default OS.

FACT Improves Performance Predictability: It is also worth mentioning that FACT significantly reduces the variance of the running times of each workload in a test set across multiple repetitions of the experiment. Figure 10 shows the standard deviation (in seconds) of the average total running time of the workloads (which was around 1000 seconds) in each of the 10 test sets under the default OpenSolaris scheduler and under FACT. Since the default scheduler is not aware of contention for resources in the memory hierarchy, it sometimes keeps worst combinations of threads together for a long time, resulting in total running times that are much greater than those when the best combinations of threads are kept together. The default OS scheduler is not the worst algorithm, however, in this respect. At the extreme, we observed the largest variance of the running times when we used 3 different assignments of 4 test threads to 4 cores and then bound threads to their cores. This gave us 3 running times: for the Best, for the Good, and for the Worst combination of threads, and the standard deviation of the running times in this case was 3-4 times larger than under the default OS. The default OS does not give such a large variance of the running times because it performs periodic thread migrations so as to balance the load across the cores when one or more threads gets occasionally blocked. Such migrations, however, are expensive events for memory-intensive threads as they cause a thread to pull its working set into cold caches, often at the expense of other threads [1]. Thus, FACT not only reduces the running times of threads but also reduces the migration costs, which can be very significant for threads with very large working sets.

6.2 Multi-step migrations

During a real deployment, it is expected that the MPI and API characteristics of threads will keep changing over time, and so a thread migration daemon should be running continuously, performing the cycle of evaluating thread characteristics, making a migration if it is predicted to improve the overall IPC, evaluating thread characteristics once again, making another migration, etc. This cycle was described in Algorithm 1.

Initially, when the system has just booted up and the IPC prediction model inside FACT has not collected any training data, thread migrations can be performed using the DI algorithm. Then, when enough training data is collected, the system can switch to using FACT for performing thread migrations. Experiments in Section 6.1 show that after observing just 3 thread migrations of 4 threads and then 3 more migrations of another 4 threads, FACT can already outperform the DI algorithm.

Even if thread characteristics do not change over time, a multi-step migration can still improve the quality of the final thread assignment. In order to verify this claim, we checked for each test set in

Table 8 which initial thread allocations led to FACT predicting Good or Worst thread assignments rather than the Best ones. For example, we found that for test set 1, out of 10 initial Good allocations, FACT suggested 8 Best assignments, 1 Good assignment and 1 Worst assignment. Also, out of 10 initial Worst allocations, FACT suggested 8 Best assignments and 2 Good assignments. Similarly, we found that for test set 1, the DI algorithm suggested 9 Best assignments and 1 Good assignment starting from 10 initial Good allocations and also suggested 9 Best assignments and 1 Good assignment starting from 10 initial Worst allocations (thus suggesting the Best assignment with probability 28/30 from a randomly chosen initial allocation and suggesting the Good assignment with probability 2/30). Therefore, the probability that DI+FACT (using the DI algorithm to perform the first migration and then using FACT to perform the second migration) would suggest the Best assignment in test set 1 can be computed as 28/30 + (2/30)(8/10) = 0.99. When averaged across the 10 test sets, this probability came out to be 0.979, as compared to the probability of 0.29 for a single-step prediction using DI and 0.94 for a single-step prediction using FACT. Similarly, when averaged across the 10 test sets, the probability of FACT(2) (making two successive migrations using the FACT-FRB algorithm) suggesting the Best thread assignment comes out to be 0.994.

The above preliminary evaluation of the multi-step migration architectures used the IPC prediction model inside FACT that was trained only on the initial training set without using the new data point obtained after performing the first thread migration. When this data point is used to adjust the IPC prediction model during a real deployment, then we would expect a multi-step migration architecture to suggest the Best thread assignment with an even higher probability.

7 Related Work

Contention-aware scheduling is a well studied area, but to the best of our knowledge our work is the first to apply a supervised learning approach to this problem domain. We believe that an approach that learns the importance of various performance predictors as opposed to relying on fixed heuristics and thresholds will be able to better evolve over time and across architectural changes.

Most contention-aware schedulers used the last-level cache miss rate as a heuristic [4, 6, 25]. The FACT framework was compared in Section 6 to the Distributed Intensity algorithm proposed in [4]. As was discussed in Section 6, while DI works well on workloads where half the threads are memory-intensive and half are CPU-intensive, FACT finds better combinations for "difficult" workloads, where there is a larger number of memory-intensive threads.

Snavely et al. [23] proposed a scheduling algorithm for reducing contention for various architectural resources on a simultaneous multithreading processor. Their technique samples some of the possible thread assignments and then builds a predictive model according to the observation. In a sense, Snavely's technique also uses learning, but the difference from our approach is that they pursue a different goal, and do not rely on supervised learning. Since Snavely's algorithm targeted a different problem, it was difficult to make a direct comparison with our algorithm.

Boyd-Wickizer et al. [3] proposed an algorithm for scheduling objects and operations to caches and cores, rather than scheduling threads to optimize CPU cycles utilization. However, optimizing CPU cycles utilization is still important and a major limitation of this work is that the algorithm relies on application developers to specify what must be scheduled. Several other works [18, 19, 20, 25] showed various scheduling techniques using cache usage characteristics of applications that dynamically estimate the usage of system resources and then optimize performance or power or both.

There is a large body of other work on reducing cache contention on multicore processors that is largely complementary to our work. Several researchers proposed a technique for detecting sharing among threads and co-scheduling threads that share data on the same chip [17, 22]. Fedorova et al. proposed a technique for improving performance isolation in light of cache contention [8]. A number of researchers used page coloring to reduce contention for cache [10, 9]. Lee et al. [26] proposed a hybrid system model for accelerating executions of warehouse-style queries, which relies on the DBMS knowledge of data access patterns to minimize last-level cache conflicts in multicore systems through an enhanced OS facility of cache partitioning. While in our previous work [24] we focused on a methodology for selecting good predictors, this paper presents a complete framework, including the development of the statistical models and adaptive migration techniques for improving the running times of the workloads.

Finally, a number of studies looked into hardware cache partitioning schemes [11, 12, 13, 14] as well as memory controller allocation algorithms [15]. Ipek et al. [16], presented an approach to designing memory controllers that operate using the principles of reinforcement learning (RL). An RL-based, self-optimizing memory controller continuously and automatically adapts its DRAM command scheduling policy based on its interaction with the system to optimize performance. These approaches can be used in conjunction with FACT to co-operatively reduce contention to even greater levels.

8 Conclusions and Future Work

This paper described the design and implementation of a Framework for Adaptive Contention-aware Thread migrations (FACT). This framework fully exploits the performance monitoring events by using them as inputs to a statistical model that predicts the IPC of co-scheduled threads. This model is then used to make optimal scheduling decisions that minimize cache interference among the threads and maximizes the average IPC of all running threads. We found that a Fuzzy Rule-Base (FRB) model had the highest IPC prediction accuracy of 74% among the considered statistical models, and the FACT-FRB combination resulted in a speedup of up to 11.6% on the selected SPECcpu2006 workloads relative to the default OpenSolaris scheduler. Moreover, FACT resulted in a smaller variance of the running times and in a smaller thread migration cost than the default scheduler. When compared to the state-of-the-art DI thread allocation algorithm, FACT was able to find the best thread assignments more consistently. Finally, we show that multi-step migration architectures can significantly increase the probability of finding the optimal thread assignment. For example, if FACT is allowed to make just two successive thread migrations, then the probability of finding the optimal thread assignment on SPECcpu2006 increases to 0.994, starting with the probability of 0.94 for a single migration based on FACT (for comparison, a single migration based on DI finds the optimal thread assignment with probability of 0.29).

In the future, we plan to evaluate FACT on a server with more than 4 cores running other workloads besides SPECcpu. We believe that multi-threaded workloads will highlight even more the benefit of the FACT framework, since such workloads will require learning a tradeoff between contention for the memory resources of the co-scheduled threads and data sharing of such threads (thread characteristics correlated with the degree of data sharing will simply be used as additional inputs to the IPC prediction model in FACT).

Ultimately, it is desirable for the system to make smart dynamic

tradeoffs between allocating CPU resources to executing workload threads vs. using the CPU resources for I/O processing vs. clocking down the CPU and conserving power, and these kinds of tradeoffs should be performed dynamically based on the current state of the system. FACT can be easily extended to such problems by using additional input variables that are correlated with I/O performance and power consumption and then using a weighted average of IPC, I/O performance and power consumption as the target variable to be predicted.

Acknowledgments

The authors would like to thank Darrin Johnson, Jonathan Chew, Eric Saxe, Kuriakose Kurivilla, and Michael Pogue of Solaris™ Kernel Group of Oracle Corp. for their valuable help throughout this work. The authors would also like to thank the anonymous reviewers for their helpful comments.

9 References

[1] R. McDougall and J. Mauro. Solaris Internals™: Solaris 10 and OpenSolaris Kernel Architecture, Prentice Hall Publications, Second Edition, July 2006 .

[2] R. McDougall, J. Mauro, and B. Gregg. Solaris Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris, pages 20-59, Prentice Hall Publications, July 2006.

[3] S. Boyd-Wickizer, R. Morris, and M. F. Kaashoek, Reinventing Scheduling for Multicore Systems. In the Proceedings of the 12th Workshop on Hot Topics in Operating Systems (HotOS-XII), Monte Verita, Switzerland, May 2009.

[4] S. Zhuravlev, S. Blagodurov, and A. Fedorova, Addressing Shared Resource Contention in Multicore Processors via Scheduling. In Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2010), Pittsburgh, PA - March 13–17, 2010.

[5] A. Fedorova, S. Blagodurov, and S. zhuravlev. Managing Contention for Shared Resources on Multicore Processors. Communications of the ACM, vol 53, no 2, February 2010. pp. 49-57.

[6] R. Knauerhase, B. Hohlt, T. Li and S. Hahn. Using OS Observations to Improve Performance in Multicore Systems, in IEEE Micro, 283, 54-66, 2008.

[7] R. Craig and P. lerous, Operating system support for multi-core processors, Review 2005 - Technical Trends, Ontario, Canada.

[8] A. Fedorova, M. I. Seltzer, and M. D. Smith. Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler. In Proceedings of the Sixteenth International Conference on Parallel Architectures and Compilation Techniques (PACT'07), pages 25-38, 2007.

[9] X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multicore cache management. In Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys'09), pages 89-102, 2009.

[10] S. Cho and L. Jin. Managing Distributed, Shared L2 Caches through OS-Level Page Allocation. In proceedings of the IEEE/ACM Int'l Symposium on Microarchitecture (MICRO), pp. 455 465, Orlando, FL, December 2006.

[11] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture. In HPCA'05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA), pages 340-351, 2005.

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

[13] N. Rafique, W. T. Lim, and M. Thottethodi. Effective management of dram bandwidth in multicore processors. In PACT'07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT), pages 245-258, 2007.

[14] G. E. Suh, S. Devadas, and L. Rudolph. A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning. In HPCA âĂŹ02: Proceedings of the 8th International Symposium on High-Performance Computer Architecture (HPCA), page 117, 2002.

[15] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt. Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems. In Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 335-346, Pittsburgh, PA, March 2010

[16] E. Ipek, O. Mutlu, J. F. Martnez, and R. Caruana. Self Optimizing Memory Controllers: A Reinforcement Learning Approach. In Proceedings of the 35th International Symposium on Computer Architecture (ISCA), pages 39-50, Beijing, China, June 2008.

[17] D. Tam, R. Azimi, and M. Stumm. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In Proceedings of the 2nd ACM SIGOPS/ EuroSys European Conference on Computer Systems, pages 47-58, New York, NY, USA, 2007.

[18] N. Lakshminarayana, S. Rao, and H. Kim. Asymmetricity Aware Scheduling Algorithms for Asymmetric Processors. In Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA), 2009.

[19] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In Proc. of the 31st Annual International Symposium on Computer Architecture (ISCA), 2004.

[20] R. Thekkath and S. J. Eggers. Impact of Sharing-Based Thread Placement on Multithreaded Architectures. In Proceedings of the International Symposium on Computer Architecture (ISCA), 1994.

[21] D. Vengerov, "A Reinforcement Learning Framework for Online Data Migration in Hierarchical Storage Systems. Journal of Supercomputing, Volume 43, Number 1, pp. 1-19, January, 2008.

[22] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson. Scheduling Threads for Constructive Cache Sharing on CMPs. In Proceedings of the 19th ACM Symposium on Parallel Algorithms and Architectures, pages 105-115. ACM, 2007.

[23] A. Snavely, D. M. Tullsen. Symbiotic Jobscheduling for a Simultaneous Multithreading Processor. In Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), November, 2000.

[24] K. K. Pusukuri, D. Vengerov, A. Fedorova. A Methodology for Developing Simple and Robust Power Models using Performance Monitoring Events. In proceedings of WIOSCA'09, June 2009, Austin, Texas, USA.

[25] A. Merkel, J. Stoess, and F. Bellosa. Resource-conscious Scheduling for Energy Efficiency on Multicore Processors. In proceedings of EuroSys 2010, April 2010, Paris, France.

[26] R. Lee, X. Ding, F. Chen, Q. Lu, and X. Zhang. MCC-DB: minimizing cache conflicts in muli-core processors for databases. In Proceedings of 35th International Conference on Very Large Data Bases, (VLDB 2009), Lyon, France, August 24-28, 2009.

[27] T. Hastie and R. Tibshirani and J. H. Friedman, The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Springer Series in Statistics, 2003, pp. 41-75, 193-222, 266-277 and 411-433.

[28] Performance Analysis and Monitoring Using Hardware Counters. http://developers.sun.com/solaris/articles/hardware_counters.html

[29] libcpc(3LIB) library http://docs.sun.com/app/docs/doc/816-5173/libcpc-3lib?a=view

[30] libpctx(3LIB) library http://docs.sun.com/app/docs/doc/819-2242/libpctx-3lib?a=view

[31] Bootstrap Relative Importance Measures. http://cran.rproject.org/web/packages/relaimpo/index.html

[32] U. Gromping. Estimators of Relative Importance in Linear Regression Based on Variance Decomposition. The American Statistician, 2007, vol. 61, pages 139-147.

[33] R `lm()` method. http://rss.acs.unt.edu/Rdoc/library/stats/html/lm.html

[34] All subsets regression, http://hosho.ees.hokudai.ac.jp/~kubo/Rdoc/library/leaps/html/leaps.html

[35] K-nearest neighbor algorithm. http://en.wikipedia.org/wiki/K-nearest_neighbor_algorithm

[36] Weighted k-Nearest Neighbor Classifier. http://bm2.genes.nig.ac.jp/RGM2/R_current/library/kknn/man/kknn.html

[37] Mean Absolute Percentage Error. http://en.wikipedia.org/wiki/Mean_absolute_percentage_error.

[38] SPEC 2000 and SPEC 2006. http://www.spec.org/

[39] stepAIC() http://stat.ethz.ch/R-manual/R-patched/library/MASS/html/stepAIC.html

[40] R Tree-based Models. http://www.statmethods.net/advstats/cart.html

[41] Decision Trees (Recursive Partitioning) http://en.wikipedia.org/wiki/Decision_tree_learning