Exploration of the Influence of Program Inputs on CMP Co-Scheduling*

Yunlian Jiang Xipeng Shen

Department of Computer Science
The College of William and Mary, Williamsburg, VA, USA 23185
{jiang,xshen}@cs.wm.edu

Abstract. Recent studies have showed the effectiveness of job co-scheduling in alleviating shared-cache contention on Chip Multiprocessors. Although program inputs affect cache usage and thus cache contention significantly, their influence on co-scheduling remains unexplored. In this work, we measure that influence and show that the ability to adapt to program inputs is important for a co-scheduler to work effectively on Chip Multiprocessors. We then conduct an exploration in addressing the influence by constructing cross-input predictive models for some memory behaviors that are critical for a recently proposed co-scheduler. The exploration compares the effectiveness of both linear and non-linear regression techniques in the model building. Finally, we conduct a systematic measurement of the sensitivity of co-scheduling on the errors of the predictive behavior models. The results demonstrate the potential of the predictive models in guiding contention-aware co-scheduling.

1 Introduction

As industry rapidly switches to multi-core processors, on-chip cache sharing is becoming common on modern machines. Although the sharing is good for hiding inter-thread communication latency and permitting flexible cache usage, it results in cache contention on Chip Multiprocessors (CMP), often causing cache thrashing and considerable performance degradation [3, 5–7, 15, 17].

Recent studies have shown that co-scheduling—that is, assigning suitable jobs onto the same chip—is beneficial for alleviating cache contention. The previous research on co-scheduling falls into two categories. The first relies on runtime sampling [5, 7, 15, 17]. Symbiotic scheduling, for example, samples program runtime performance under various schedules and picks the best one as the optimal schedule [15]. Although runtime sampling works well for a small number of programs, it may be difficult to scale up because the number of possible schedules is exponential in the number of jobs.

The second category includes profiling-directed techniques. These techniques first conduct a profiling run of the executions and then co-schedule them accordingly [3, 6]. Although they showed effectiveness, it is unclear how they would work if the real executions' inputs differ from the training ones.

* Supported by the National Science Foundation (CNS-0720499) and IBM CAS Fellowship.

On a given CMP architecture, cache contention depends on two factors: the programs that run together, called *corunners*, and their inputs. The first factor has been the main focus of previous studies. This work is distinctive in concentrating on the second factor. The goal is to uncover the effects of program inputs on CMP co-scheduling and to explore the solutions.

This study contains two major components. First, we conduct a systematic measurement to the influence of program inputs on corun performance. The experiments employ hardware performance counters in an Intel quad-core machine to measure all possible coruns of a dozen programs on different inputs. The results show that a schedule, although suitable for runs on one set of inputs, may cause 4 times more performance degradation to the runs on a different set of inputs. A CMP co-scheduler, therefore, must have the capability to adapt to different program inputs.

To address the effects of program inputs, in the second part of this work, we explore the construction of cross-input predictive models. We model the relation between program inputs and memory behavior through statistical learning techniques, and compare the effectiveness of both linear and non-linear regression techniques. We use a recently proposed cache-contention-aware proactive scheduler, CAPS [12], to evaluate the predictive models. A broader sensitivity study shows the different effects of various memory behaviors on co-scheduling, suggesting the opportunities for further improvement of the predictive models. The evaluation on CAPS shows an accuracy of over 85% for memory behavior prediction and a 26.3% reduction of the performance degradation that the default coruns cause.

In the rest of the paper, Section 2 uncovers the influence of program inputs on corun performance on CMPs. Section 3, after giving an overview of CAPS, concentrates on the approaches to constructing predictive models for a set of memory behaviors. Section 4 reports the sensitivity of CAPS on the prediction errors of memory behaviors. Section 5 discusses related work, followed by a summary in Section 6.

2 Influence of Program Inputs on Corun Performance

To explore the influence of program inputs on co-scheduling, we measure the coruns of a dozen SPEC CPU2000 programs on their *test*, *train*, and *ref* inputs. The machine we use is a Dell PowerEdge 1850 server with two Intel Xeon 5150 2.66 GHz dual-core processors, each equipped with a 4MB shared L2 cache. The machine runs Fedora Core 5 Linux x86_64 distribution with a Linux kernel of 2.6.17. We use Gcc 4.1 as our compiler. For performance measurement, the Linux kernel is patched to support Performance API (PAPI) version 3.5, which collects performance events by accessing the hardware performance counters on the machine [2].

In the experiment, each time we bind two programs on a dual-core processor and start running them at the same time. To avoid the distraction from the difference between programs' executions, we follow Tuck and Tullsen's method [16], letting each program run 10 times consecutively, and only collecting the behavior of those runs that overlap with the other's execution. In that way, we conduct all possible coruns of the 12 programs for each kind of input, totally 198 coruns. For each corun, PAPI reports, for each of the two programs, the average cycles per instruction (CPI), denoted by *cCPI* (*c*

stands for corun). We also measure the CPI of the single run of each program, denoted by *sCPI* (*s* for single-run). Single-runs are subject to no cache contention. We define a program's performance degradation (because of cache contention) as follows:

$$corun\ \ degradation = \frac{cCPI - sCPI}{sCPI}.$$

The larger a degradation is, the worse the cache sharing affects the program's running speed.

The boxplots in Figure 1 show the results. The differences among the boxplots inside a group reveal the strong influence of program inputs on corun performance. Among the 12 benchmarks, *twolf* and *vpr* are the two that have the largest performance variation across inputs. The *test* runs of both of them have no performance degradation, no matter which program is their corunner. Whereas, their *train* runs show up to 15% and 36% degradations, and their *ref* runs show up to 76% and 64% degradations. For the other programs, the *train* and *ref* runs are 15% to 564% worse than those of their respective *test* runs (in terms of median values). The results demonstrate that program inputs affect corun performance significantly.

The results also show a second phenomenon. Although the working sets of the programs usually increase as input size increases, the corun performance degradation doesn't necessarily increase. For instance, the *ref* runs of *equake*, *mcf*, and *parser* clearly have less degradation than their *train* runs. This phenomenon shows that corun degradation does not necessarily increase when the single-run cache miss rate increases. An extreme case may convey the intuition behind: A program whose single run has no cache hits clearly won't have any more cache misses when it coruns with other programs; hence, its corun performance degradation must be negligible. This observation suggests that in the design of co-scheduler, cache miss rate may not provide the sufficient information.

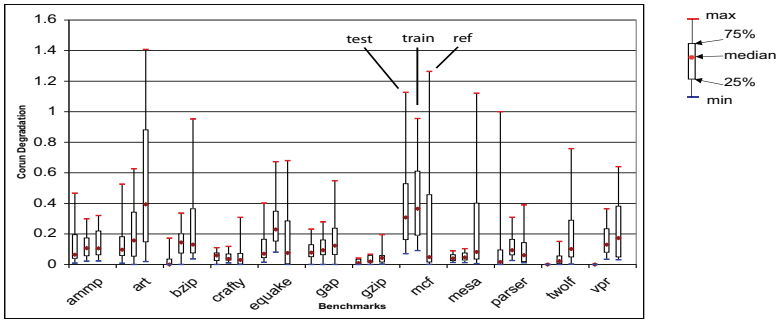

Fig. 1. The boxplot showing the distribution of the performance degradation of each program when it coruns with the other 11 programs. The three boxplots in a group respectively correspond to the executions on *test*, *train*, and *ref* inputs.

To see the influence of inputs on corun scheduling, we use CAPS (described next) to find three best schedules, respectively for the *test*, *train*, and *ref* runs of the 12 programs.

We then apply the best schedule of *ref* runs to the other two sets of runs. The two sets show up to 4.3 times more performance degradation than their executions under their own best schedules. These results suggest the great importance for a scheduler to adapt to different program inputs when dealing with cache contention on CMPs.

3 Handling Program Inputs for Co-scheduling

Our approach to addressing the influence of program inputs is to build predictive input-behavior models, which can predict program memory behavior from a given input. Because some co-schedulers can estimate corun performance from single-run memory behavior and then derive the best schedules, we need only the mechanism to accurately predict the memory behavior of a program's single-runs (on arbitrary inputs).

Before describing the model construction, we briefly describe a contention-aware scheduling system, CAPS [12], as it is our underlying framework for evaluation. We choose CAPS for two of its desirable features. First, CAPS is able to efficiently produce the schedule that minimizes the total degradation given the performance of all possible coruns. Thereby, we can easily use the desirable schedule as the baseline to evaluate the schedules produced upon the input-behavior models. Second, some of the memory behaviors (e.g., reuse signatures) used by CAPS have been showed to be cross-input predictable [4], which simplifies some parts of the construction of the input-behavior models.

3.1 Overview of CAPS

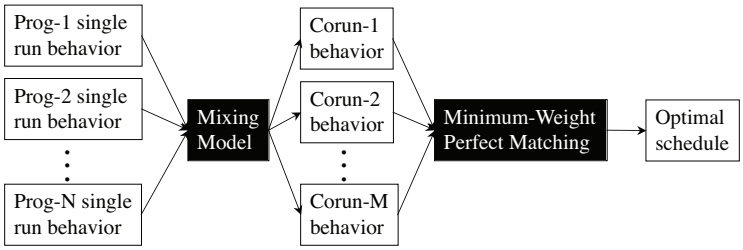

Fig. 2. The key components of the cache-contention-aware proactive scheduler (CAPS).

CAPS is a system for proactive job-scheduling on CMPs by exploiting the prediction of cache contention. It has two versions, respectively for runtime process scheduling and batch job scheduling. This work uses the version for batch processing, in which case, all jobs to be scheduled are known beforehand.

As depicted in Figure 2, at the heart of CAPS are two components. The first component predicts the performance degradation of each possible corun using the memory behavior of single-runs of each program. The second component maps the corun performance to a fully connected graph, with each vertex representing a program, and each

edge having a weight equal to the total performance degradation of the corun of the two vertices. It then applies the minimum-weight perfect matching algorithm to efficiently determine the schedule that minimizes the total of the corun degradation of all the programs.

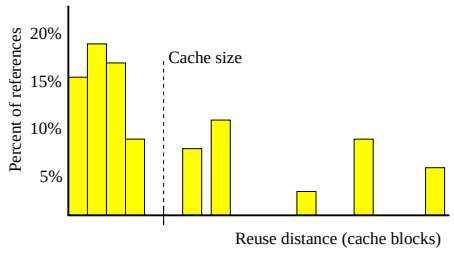

Fig. 3. An example of cache-block reuse signature

CAPS complements previous co-scheduling in that it is proactive, relying on predicted corun performance rather than runtime sampling, and thus overcoming the scalability issue of previous techniques. The details of CAPS are out of the scope of this paper. Here we only describe the single run behaviors used by the mixing model in CAPS; they are the prediction targets of the input-behavior models that are to be built.

- **Reuse Signature**: a histogram showing the distribution of data reuse distances in an execution. **Reuse distance**, also called LRU stack distance, is the number of *distinct* data elements accessed between this and the previous access to the same data element [4]. For the second access to "a" in the reference trace "a b b c d a", the reuse distance is 3 because "b","c" and "d" are accessed in the between. Figure 3 illustrates a reuse signature, with reuse distance on the horizontal axis and the percentage of memory references on the vertical axis. Each bar on the graph shows the percentage of memory references whose reuse distances are in a particular range. If the counting unit for reuse distance is a cache line, a reuse signature can be used to approximate the cache miss rate of the execution for a cache of arbitrary size: All accesses with reuse distance larger than the cache size are simply considered as misses. The approximation has shown an accuracy of over 94% for both fully-associative and set-associative caches [18]. CAPS uses reuse signatures of single runs as the basis to estimate corun cache performance.
- **Accesses per Instruction**: the average number of memory accesses per instruction. This statistic reflects the density of memory references in an execution. It is one of the factors CAPS uses to approximate program performance from memory behaviors (for both single-run and coruns).
- **Distinct Blocks per Cycle**: the average number of distinct memory blocks that are accessed in a CPU cycle. This statistic reflects how aggressive a process competes for caches. CAPS uses it for the prediction of corun cache performance. In CAPS, the counting unit of memory blocks is simply a cache block.

Next, we focus on the construction of the predictive models for each of the three kinds of memory behaviors through statistical learning techniques.

3.2 Constructing Predictive Input-Behavior Models

The memory behavior of a single-run of a program, denoted by B, depends on the running environment E, the program code P, and the input I. In this work, E and P are given, and the goal is to find the function $f()$ mapping from I to B. With such a function, plugging any input into $f()$ will generate the predicted behavior of the program's corresponding single-run execution. We formalize the task as a statistical learning problem. By feeding a program with different inputs, $I_1, I_2, \ldots, I_N$, we observe the corresponding behavior of the program's executions, represented by $B_1, B_2, \ldots, B_N$. The input-behavior pairs, $< I_i, B_i > (i = 1, 2, \ldots, N)$, compose a training set, from which we use regression techniques to approximate function f.

Linear and Non-Linear Regression Regression techniques are designed to discover the relation between a set of input attributes and a set of outputs. Linear regression assumes that the relation can be expressed by a linear function; non-linear regression permits more sophisticated functions.

Least Mean Squares (LMS) is a commonly used linear regression technique. Suppose f is a linear function mapping input $\overrightarrow{I}$ to a behavior B for a given program. Given training data set $< \overrightarrow{I_i}, B_i >$ (i=1,2,. . .,N), the goal of LMS is to find the approximation of function f, represented by $\hat{f}$, such that the mean error squares, $\frac{1}{N}\sum_{i=1}^{N}(B_i - \hat{f}(\overrightarrow{I_i}))^2$, is minimized.

LMS is simple and efficient, but applies to only linear functions. For non-linear regression, we choose the k-*Nearest-Neighbor* method. This method is an instance-based learning technique. For a new query instance, it retrieves a set of similar instances from memory and uses them to estimate the new output value. When $k = 1$, the method is named the Nearest-Neighbor method, or *NN* in short. The approximated function $\hat{f}()$ has an implicit and usually non-linear form [8]. The model building is simple, just recording the training instances into a data structure that can be efficiently searched. There are many other statistical learning techniques, such as Regression Trees and Support Vector Machines; they are more complex and costly. We restrain ourselves to a small number of training runs in order to limit the overhead of the offline profiling. Those more complex learning techniques often require a larger training data set.

Besides LMS and NN, we also use a hybrid method. For a given program, it chooses the better one between LMS and NN in terms of *training* errors. (The training error of a model is the prediction error of the model when being applied to the training data.) For each program showed in Figure 1, besides its *test*, *train*, and *ref* inputs included in the SPEC suite, we obtained another input from the collection of additional representative inputs attained by Berube and Amaral [1]. For programs not included in the collection (*ammp*, *art*, *equake*, *mesa*, and *twolf*), we created an input by modifying the corresponding *ref* input. We use *train* inputs for model testing, and the others for training.

Next, we show the effectiveness of the three regression techniques on each of the three kinds of memory behavior that are used in CAPS.

Prediction of Accesses per Instruction The first question for building a model between program inputs and accesses per instruction is the representation of program inputs. Given the close relation between program data size (i.e., the number of distinct data items) and memory behavior, we adopt the approach proposed by Ding and Zhong, characterizing a program input by the estimated data size that can be obtained through distance-based sampling. Distance-based sampling observes data reuses at the beginning of an execution and estimates data size based on long reuse distances [4]. So, in this and the rest experiments, data size is the I_i in the input-behavior pair $< I_i, B_I >$, whereas the B_i is specific to each experiment; it is the accesses per instruction in this experiment.

The left half of Table 1 reports the accuracy in predicting accesses per instruction. The three methods produce similar accuracies: 86.43% by LMS, 88.27% by NN, and 88.69% by the hybrid method. Program *equake* shows the lowest accuracy (54.58%) mainly because of its more complex relations between inputs and accesses per instruction. More training inputs and more sophisticated models may be helpful.

Table 1. Prediction accuracies of linear (LMS) and non-linear (NN and Hybrid) models.

Programs	Accesses per instruction			DPI		
	LMS	NN	Hybrid	LMS	NN	Hybrid
ammp	89.58	98.76	98.76	39.83	86.72	86.72
art	98.86	94.25	98.86	98.96	94.25	98.96
bzip	75.79	78.62	78.62	67.69	64.05	67.69
crafty	99.54	99.24	99.54	76.31	72.50	76.31
equake	54.58	54.42	54.58	82.27	82.13	82.27
gap	74.75	79.35	79.35	79.87	78.08	79.87
gzip	82.76	86.98	86.98	77.85	66.47	77.85
mcf	90.25	92.45	92.45	89.73	88.11	89.73
mesa	96.39	96.98	96.98	89.43	93.33	93.33
parser	96.02	98.61	98.61	89.49	70.42	89.49
twolf	97.11	98.10	98.10	52.12	86.75	86.75
vpr	81.50	81.50	81.50	96.30	95.28	96.30
Average	**86.43**	**88.27**	**88.69**	**78.32**	**81.51**	**85.44**

Prediction of Distinct Blocks per Cycle The statistic, distinct blocks per cycle, reflects the average cache requirement of a process. It can be regarded as a product of two factors:

$$DPC = DPI * IPC$$

where, DPI is the average number of distinct blocks accessed per instruction, and IPC is the instructions per cycle. DPI is an attribute solely determined by the program; whereas IPC is a runtime behavior, attainable from hardware performance counters. The prediction of DPC therefore can be conducted in two steps. Given a new input, an offline-trained model predicts the DPI of the new execution. During the new execution, the

DPC can be obtained by multiplying the predicted DPI with the runtime IPC. Therefore, building a predictive model for DPI is the key to the prediction of distinct blocks per cycle.

Because DPI is an average value for an interval, it is determined by the interval length. For an interval containing nothing except one memory access instruction, the DPI is 1, which is the upper bound of DPI under the assumption that one instruction may conduct at most one memory access. As the interval becomes larger, DPI changes non-monotonically, determined by the ratio of non-memory-access instructions and the frequency in which memory-access instructions access a new object. When the interval length becomes large enough to cover at least one access to all the blocks in the program, DPI decreases as the interval length increases.

The DPI used in CAPS is the average DPI of all the reuse intervals[1], computed in the following formula:

$$w = \frac{\sum_{i=1}^{B} r_i \bar{w}_i}{\sum_{i=1}^{B} r_i}$$

where, B is the number of bars in the reuse signature of the execution, r_i is the number of memory references in bar i, and $\bar{w}_i$ is the average of all the DPIs of the reuse intervals in bar i.

The right half of Table 1 shows that NN is slightly more accurate than LMS, 81.51% versus 78.3%. The hybrid model yields an accuracy of 85.4%.

Reuse Signatures Previous work has explored the cross-input predictability of reuse signatures. For example, Ding and Zhong have shown an accuracy of over 94% for the prediction of the reuse signatures of 15 complex programs [4]. Their technique is based on a desirable property of reuse signatures: No reuse distance of an execution can be larger than the data size of the execution. (This property comes from the definition of reuse distance.) They therefore test a set of sub-linear functions in training runs and choose the best one as the model for the prediction of reuse signatures. This work adopts their established technique.

4 Influence of Prediction Errors on Co-Scheduling

We feed CAPS the predicted memory behaviors to test the influence of the prediction errors on co-scheduling. Figure 4 shows the average performance degradation of the benchmarks included in Figure 1.

The baseline is an *a posteriori* schedule, which is the best over all possible schedules. We obtained it by applying the minimum-weight perfect matching to all real coruns. (Recall that the algorithm minimizes the total degradation.) The *random* bar shows the average result of 100 random schedules. It reflects the performance of the default scheduler in the current CMP system.

[1] The reuse interval of a data reuse is the interval between the previous and the current access to the same data item.

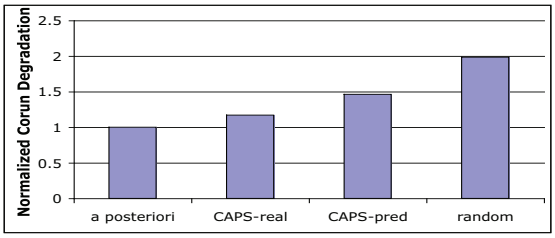

Fig. 4. The average performance degradation under different schedules. The "a posteriori" schedule is the best schedule obtained on all corun information; "CAPS-real" is the schedule by CAPS on real single-run behaviors; "CAPS-pred" is the schedule by CAPS on single-run behaviors predicted by the models described in Section 3.2; "random" reflects the default schedule in the CMP system.

The difference between *caps-pred* and *caps-real* shows the influence of the prediction errors of the behavior models—0.28 more degradation. With that influence, *caps-pred* still reduces the performance degradation of the random schedule from 1.99 to 1.46. The extra degradation that *caps-real* has over the *a posteriori* schedule is due to the inaccuracy inside CAPS (e.g., the mixing model).

To achieve a better understanding of the influence from prediction errors, we conducted a broader study to the sensitivity of CAPS on each of the three kinds of memory behaviors. We introduce a range of random errors into the three kinds of memory behavior, one kind per time, and then measure the resulting performance of CAPS.

For lack of space, we leave the detailed results in a technical report [9]. As a summary, CAPS is most sensitive to the errors in accesses per instruction and DPIs: an error of 8% in them respectively causes the performance degradation to increase by 12.3% and 17.6%; 16% causes an increase of 18.3% and 18.8%. CAPS is less sensitive to the errors in reuse distance histograms: an error of 8% causes an increase of 3.6%; 16% causes an increase of 12.2%.

To improve the accuracy of the predictive models, more training inputs may help. A combination with hardware performance counters may also be beneficial, especially for accesses per instruction. In addition, it is potentially helpful to characterize program inputs more sophisticatedly [13] rather than only relying on sampled data size. Finally, the combination of CAPS with locality phase analysis [14] can make the scheduler adaptive to runtime program behavior changes.

We emphasize that the main contributions of this paper are the exploration of the influence of program inputs on co-scheduling and cross-input memory behavior modeling. The details on the integration of the models into CAPS and the extension to runtime scheduling (presented in our technical report [12]) are out of the scope.

5 Related Work

We are not aware of any work on the study of program inputs for co-scheduling on CMPs. The closest work on program inputs exists in runtime adaptive optimizations

and feed-back directed optimizations behavior (e.g., [1, 10]). Most recent studies on CMP (or SMT) co-scheduling either rely on runtime estimation of cache usage from hardware performance counters [5, 7, 15, 17], or offline profiling [3, 6]. None of them systematically explores the effects of program inputs on co-scheduling. Although runtime techniques implicitly adapt to input changes, they require periodic sampling of many different coruns, making them subject to some scalability or applicability limitations as discussed in Section 1. Architecture extensions, e.g. cache activity vector [11], are complementary to software co-scheduling in offering fine-grained cache behavior.

6 Conclusion

This work focuses on the exploration of the influence of program inputs on corun performance of programs running on CMPs equipped with shared caches. It draws the conclusion that the influence of program inputs is so strong that a cache-contention-aware scheduler has to adapt to them. The second part of the paper describes our practice in constructing cross-input predictive models for a set of memory behaviors that are used in a recently proposed proactive co-scheduling system. The experiments show reasonably accurate prediction through the uses of linear and non-linear regression techniques. The paper then presents a systematic measurement of the influence of the prediction errors on co-scheduling. The results suggest that the cross-input prediction models are able to help the scheduler significantly reduce cache contention on shared caches.

References

1. P. Berube and J. N. Amaral. Benchmark design for robust profile-directed optimization. In *Standard Performance Evaluation Corporation (SPEC) Workshop*, 2007.
2. S. Browne, C. Deane, G. Ho, and P. Mucci. Papi: A portable interface to hardware performance counters. In *Proceedings of Department of Defense HPCMP Users Group Conference*, 1999.
3. D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of HPCA*, 2005.
4. C. Ding and Y. Zhong. Predicting whole-program locality with reuse distance analysis. In *Proceedings of PLDI*, 2003.
5. A. El-Moursy, R. Garg, D. H. Albonesi, and S. Dwarkadas. Compatible phase co-scheduling on a cmp of multi-threaded processors. In *Proceedings of IPDPS*, 2006.
6. A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum. Performance of multithreaded chip multiprocessors and implications for operating system design. In *Proceedings of USENIX Annual Technical Conference*, 2005.
7. A. Fedorova, M. Seltzer, and M. D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proceedings of PACT*, 2007.
8. T. Hastie, R. Tibshirani, and J. Friedman. *The elements of statistical learning*. Springer, 2001.
9. Y. Jiang and X. Shen. Study of cross-input predictability of inclusive reuse distance. Technical Report WM-CS-2007-13, Computer Science Department, The College of William and Mary, 2007.
10. X. Li, M. J. Garzaran, and D. Padua. A dynamically tuned sorting library. In *Proceedings of CGO*, 2004.
11. A. Settle, J. L. Kihm, A. Janiszewski, and D. A. Connors. Architectural support for enhanced smt job scheduling. In *Proceedings of PACT*, 2004.
12. X. Shen, Y. Jiang, and F. Mao. Caps: Contention-aware proactive scheduling for cmps with shared caches. Technical Report WM-CS-2007-09, Computer Science Department, The College of William and Mary, 2007.
13. X. Shen and F. Mao. Modeling relations between inputs and dynamic behavior for general programs. In *Proceedings of LCPC*, 2007.
14. X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *Proceedings of ASPLOS*, 2004.
15. A. Snavely and D. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Proceedings of ASPLOS*, 2000.
16. N. Tuck and D. M. Tullsen. Initial observations of the simultaneous multithreading Pentium 4 processor. In *Proceedings of PACT*, 2003.
17. X. Zhang, S. Dwarkadas, G. Folkmanis, and K. Shen. Processor hardware counter statistics as a first-class system resource. In *Proceedings of HotOS*, 2007.
18. Y. Zhong, S. G. Dropsho, X. Shen, A. Studer, and C. Ding. Miss rate prediction across program inputs and cache configurations. *IEEE Transactions on Computers*, 56(3), 2007.