Combining Locality Analysis with Online Proactive Job Co-Scheduling in Chip Multiprocessors

Yunlian Jiang Kai Tian Xipeng Shen

Computer Science Department
The College of William and Mary, Williamsburg, VA, USA 23187
{jiang,ktain,xshen}@cs.wm.edu

Abstract. The shared-cache contention on Chip Multiprocessors causes performance degradation to applications and hurts system fairness. Many previously proposed solutions schedule programs according to runtime sampled cache performance to reduce cache contention. The strong dependence on runtime sampling inherently limits the scalability and effectiveness of those techniques. This work explores the combination of program locality analysis with job co-scheduling. The rationale is that program locality analysis typically offers a large-scope view of various facets of an application including data access patterns and cache requirement. That knowledge complements the local behaviors sampled by runtime systems. The combination offers the key to overcoming the limitations of prior co-scheduling techniques.

Specifically, this work develops a lightweight locality model that enables efficient, proactive prediction of the performance of co-running processes, offering the potential for an integration in online scheduling systems. Compared to existing multicore scheduling systems, the technique reduces performance degradation by 34% (7% performance improvement) and unfairness by 47%. Its proactivity makes it resilient to the scalability issues that constraints the applicability of previous techniques.

1 Introduction

With the advent of Chip Multiprocessor (CMP) and Simultaneous Multithreading (SMT), on-chip cache sharing becomes common on various computing systems, including embedded architectures, desktop computers, clusters in data centers, and so on. The sharing shortens inter-thread latency and allows flexible cache usage. However, it also brings cache contention among co-running jobs, causing programs different degrees of performance degradation, hence impairing system fairness and overall performance [3, 6–9, 11, 19, 29]. The problem is especially important for embedded systems due to the more limited cache on them [20].

In operating systems (OS) research, the recent attempts in alleviating cache contention mainly focus on reactive process scheduling[1] [2,4,6–8,18,25]. These techniques typically sample job executions periodically. During the sampling, they track hardware performance counters to estimate the cache requirement of each process and derive a

[1] In this work, we concentrate on the schedule of *independent* jobs.

better schedule. (For a system containing multiple CMP chips, a better schedule usually means a different assignment of jobs to processors or a different allocation of CPU timeslices to processes.)

Although these techniques work well under certain conditions, the strong reliance on runtime sampling imposes some limitations on their effectiveness and applicability. The main obstacle is that the sampled behavior only reflects the behavior of a process during a *certain time period* when it co-runs with a *certain subset* of processes. Whereas, good scheduling need recognize the inherent cache requirement of a process and its influence on and from all possible co-runners. As a result, most prior techniques require both periodic re-sampling and frequent reshuffles of processes among different co-run groups [8, 25].

These requirements not only cause more sampling overhead (cache performance is often inferior during sampling periods) but also limit the applicability of previous scheduling techniques. For instance, cache-fair scheduling needs the sampling of 10 different co-runs (i.e., runs with different co-runners) per process in every sampling phase, and requires the system to contain a mix of *cache-fair* and *best-effort* processes [8]; symbiotic scheduling [4,25], which samples program performance under various schedules and estimates the best schedule, is difficult to be applied to large problems—the number of possible schedules increases exponentially with the numbers of jobs and processors (e.g., there are 2 million ways to co-schedule 16 jobs on 8 dual-cores).

This work attempts to free prior techniques from those constraints by integrating the knowledge of programming systems[2]. Our exploration combines program behavior analysis with operating systems' control of underlying resources. The motivation for the combination is that *program characteristics determine the cache requirement of a program, and it is programming systems that have the best knowledge of those characteristics*.

To be beneficial, the combination must meet two requirements. First, it needs a lightweight locality model to efficiently predict the cache requirement and co-run performance of programs. Based on reuse distance analysis, this work develops a lightweight co-run locality model—a unified sensitivity and competitiveness model—to enable fast prediction of the influence that a process may impose on and receive from its co-runners. The model is cross-input predictive. The second requirement is that the system scheduler must effectively integrate the locality model into runtime scheduling. This work presents the design of cache-contention aware proactive scheduling (CAPS), which assigns processes to processors according to the predicted cache-contention sensitivities. In our experiments, CAPS reduces performance degradation by 33.9% (7% performance improvement) and unfairness by 47.5%.

This work is not the first one to combine program-level locality analysis with thread or process scheduling. In time-sharing environment, there have been some studies [28, 31] in exploiting footprint size of programs to help schedule threads to minimize the influence on cache imposed by context switches. In traditional SMP (Symmetric Multiprocessing) systems, there have been some work [18] transforming programs to minimize cache false sharing. But none of those studies have tackled systems with shared

[2] Programming systems is an expanded term for compilers, referring to both static and dynamic systems for program behavior analysis.

cache. To the best of our knowledge, this work is the first offering a lightweight formulation of program-level data locality applicable for runtime CMP co-scheduling.

In the rest of this paper, we describe the background on co-run locality prediction in Section 2, present a lightweight locality model for scheduling in Section 3, explain the design of CAPS in Section 4, report results in Section 5, compare with previous co-scheduling in Section 6, and summarize the paper in Section 7.

2 Background on Co-Run Locality

Co-run locality analysis enables the prediction of the performance of co-running programs, laying the foundation for contention-aware proactive scheduling. This section first introduces concurrent reuse distance, a concept closely related to shared-cache performance, and then describes a theorem establishing the basis for the cache-contention sensitivity model to be presented in the next section.

In the following discussion, we assume that the architecture is an SMP machine with multiple CMP chips, the cores on a chip share an on-chip L2 cache, and each core has a dedicated L1 cache. Since our focus is on cache sharing, by default, the memory references in the discussion do not include the references that are L1 cache hits.

2.1 Reuse Distance

Underlying the co-run locality model is the concept of **reuse distance** or LRU stack distance, defined as the number of *distinct* data items accessed between the current and the previous reference to the same data item [5, 16]. Treating a cache block as a single data item leads to cache-block reuse distance. Researchers have used cache-block reuse distance histograms, also called *reuse signatures* [36], to predict the performance of dedicated caches. Figure 1 illustrates the prediction: Every memory reference on the right side of the cache-size line is considered a cache miss because too many other data have entered the cache between two references to the same data item. The estimation is precise for fully-associative cache, but also applicable for set-associative cache [15, 35].

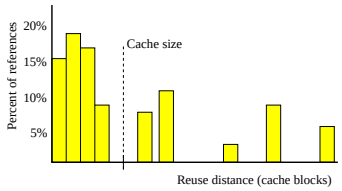


Fig. 1. An example of cache-block reuse signature

We use **concurrent reuse distance** to refer to the extension of reuse distance on shared caches. It is defined as the number of distinct data items that are referenced by *all* cache sharers (i.e., processes sharing a cache) between the current and the previous access to the same data item. For clarity, we use *standalone reuse distance* for the

traditional concept of reuse distance. The histograms are named as *concurrent reuse signatures* and *standalone reuse signatures* respectively. Using concurrent reuse signatures, we can predict shared-cache misses in the manner similar to the prediction of dedicated-cache misses.

2.2 Derivation of Concurrent Reuse Signatures

Concurrent reuse signatures, although good for corun-locality prediction, is hard to measure. The main reason is that direct measurement requires detailed memory monitoring, which both disturbs the order of memory references conducted by cache sharers, and slows down program executions by hundreds of times.

Fortunately, concurrent reuse signatures can be derived from the corresponding standalone reuse signatures through a statistical model [22,23]. Furthermore, prior work has shown that standalone reuse signatures of a program can be accurately predicted across the program's inputs [5]. These make concurrent reuse signature cross-input predictable: The concurrent reuse signatures of new executions of a group of co-running programs can be derived from the executions of those programs on some training inputs. Cross-input predictability is vital to the use in contention-aware scheduling because of the strong dependence of cache contention on program inputs.

The cost of the statistical model increases quadratically in the length of an execution [22], infeasible to be used in runtime scheduling. We use **distinct data blocks per cycle (DPC)** to simplify the process. The DPC of a process in a given interval is the average number of distinct data blocks (typically in the size of a cache line) accessed by the process in a CPU cycle through the interval. Roughly speaking, DPC is the average footprint in a cycle. It reflects the aggressiveness of a process in competing for cache resource. As an example, suppose a program accesses the following data blocks in 100 cycles: b1 b1 b3 b5 b3 b1 b4 b2. The corresponding DPC is 5/100 = 0.05 (footprint is 5). Correspondingly, the DPC of a set of processes, P, is defined as the number of distinct data blocks that are accessed in a CPU cycle by all the processes in P, that is, $DPC(P) = \sum_{q \in P} DPC(q)$.

To ease explanation, we define the *reuse interval* of a reference as the interval between this and the previous reference to the same data item. The following theorem more precisely characterizes the connection between DPC and cache contention.

Theorem 1 *Suppose, process p shares a fully-associative cache of size L with a set of processes P ($p \notin P$). (No shared data among processes.) Consider a cache hit by p that has standalone reuse distance of d ($d < L$). Let σ and σ' be the average DPC of p and P during the reuse interval of the reference. Then, if and only if $\frac{d}{L-d} < \frac{\sigma}{\sigma'}$, the reference remains a cache hit in the presence of cache competition from P.*

The proof of this theorem is straightforward if it is noticed that the concurrent reuse distance of the reference is $(d + \sigma' d/\sigma)$.

This theorem suggests that along with standalone reuse signatures, knowing the DPC of every reuse interval is enough to compute the miss rate of a co-run. It is however too costly to get DPCs in such detail. We use the averaged value of the DPCs of all

reuse intervals of a program to obtain a suitable tradeoff between the accuracy and efficiency of locality prediction. Although the theorem assumes fully-associative caches, its application produces good estimation for set associative caches as well [22].

3 Cache-Contention Sensitivity and Competitiveness

Theorem 1 provides a way to estimate concurrent reuse signatures and thus co-run performance. It is lightweight enough for offline analysis and batch job scheduling [22], but not for runtime uses: It takes more than 2,000 μs to predict the performance of 32 jobs co-running on dual-core processors. This work simplifies it to a competitiveness-sensitivity model. *Competitiveness* and *sensitivity* respectively characterize the statistical expectation of the influence that a process may impose on and receive from random co-runners. This model is important for making runtime proactive scheduling scalable. As we will see in Section 5, CAPS capitalizes on the model to make sensitive processes co-run with uncompetitive ones to achieve better performance.

3.1 Sensitivity

The definition of cache-contention sensitivity is as follows:

$$Sensitivity = \frac{\overline{CPI_{co}} - CPI_{si}}{CPI_{si}} \tag{1}$$

where, CPI_{si} is the cycles per instruction (CPI) of a process's single run, and $\overline{CPI_{co}}$ is the statistical expectation of the CPI of that process when it co-runs with random processes.

The estimation of CPI_{si} is straightforward: As explained in Section 2.1, we can predict the cache miss rate of a process's single run from its standalone reuse signatures; the corresponding CPI (given the cache miss rate) can be estimated using existing techniques (e.g. [26]).

To estimate $\overline{CPI_{co}}$ in the same way, we have to obtain the statistical expectation of the cache miss rates of the process's co-runs. The number of co-run misses equals the sum of single-run misses and the extra misses caused by co-run contention. Since single-run misses are obtainable as mentioned in the previous paragraph, the problem becomes the computation of the statistical expectation of the number of extra misses. The following corollary of Theorem 1 offers the solution.

Corollary 1 *Let $F()$ be the cumulative distribution function of the DPCs of all programs, and L be the shared cache size. Suppose a process p has H memory references whose standalone reuse distances, d_i, are smaller than L (i=1, 2, $\cdots$, H). Let σ_i represent the DPC of the corresponding reuse interval. When process p co-runs with some randomly-picked programs that share no data with p, the expectation of the cache miss rate of the H memory references is*

$$\delta = 1 - \frac{1}{H}\sum_{i=1}^{H} F(\sigma_i(L - d_i)/d_i). \tag{2}$$

Proof. Let σ' represent the average DPC of the co-runners of p in the reuse interval corresponding to σ_i. Theorem 1 tells us that if and only if $\sigma' < \sigma_i(L - d_i)/d_i$, reference i remains a hit. Since the probability for that condition to happen is $F(\sigma_i(L - d_i)/d_i)$, the expectation of the number of cache hits among the H references is $\sum_{i=1}^{H} F(\sigma_i(L - d_i)/d_i)$. The conclusion follows.

With this corollary, we can compute the sensitivity of a process from its DPC and standalone reuse signature. Since references are grouped in bars in reuse signatures, the computation uses a bar as a unit; H thus equals the number of bars whose reuse distances are smaller than L. For computing the $F()$ items efficiently, we build a lookup table for F by using 3.9 billion data reuses from a dozen randomly chosen SPEC CPU2000 programs (included in Figure 2). The table contains 200 items corresponding to 200 evenly-spaced points between 0 and 0.237.

3.2 Competitiveness

We initially intended to use a process's average DPC as competitiveness. But our experiments reveal the strong correlation between the influence a process imposes on and receives from its co-runners. This observation leads to a unified competitiveness and sensitivity model.

Figure 2 plots the performance degradation of all the 66 pair-wise co-runs of a dozen SPEC CPU2000 programs (*train* runs) on an Intel Xeon 5150 processor (specified in Section 5). In the graph, points on solid curves show the program's own degradation and points on broken curves show the degradation of its co-runner. For legibility, each program's data are sorted in ascending order of self degradation and then connected into curves. The two curves corresponding to every program show similar trends. The correlation coefficient between all the self and co-runner degradations is 0.75. (As an extra evidence, the coefficient is 0.73 for the 13 SPEC programs shown in Figure 3.) The intuition behind the strong correlation is that, a program that is sensitive to cache contention tends to fetch data from a large portion of the shared cache frequently. Hence, it tends to impose strong influence on its co-runners, that is, it tends to be competitive. As an exception, stream programs are competitive but insensitive. Although they access cache intensively, those programs have few data reuses and thus rely on no cache for performance. Fortunately in offline training, it is easy to detect stream programs thanks to their distinctive data access patterns. The scheduling process, CAPS, treats those programs as competitive programs and pair them with other insensitive programs (detailed next). For other programs, CAPS simply uses sensitivity for competitiveness. This unified model simplifies the design of runtime scheduler.

4 Contention-Aware Proactive Scheduling (CAPS)

The principle of CAPS is to couple sensitive processes with insensitive (thus likely uncompetitive) processes. This section uses Linux as an example to explain how CAPS can be integrated in runtime schedulers.

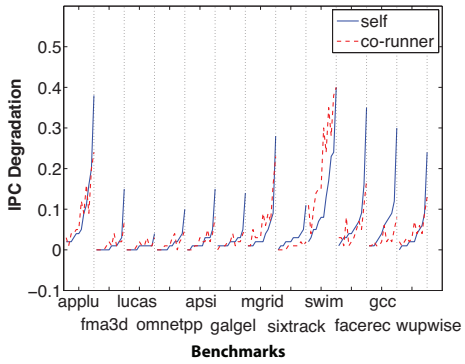


Fig. 2. Each program has 11 pair-wise co-runs, respectively with each of the other 11 programs. The points on the solid curve show the degradations of this program in those co-runs; the points on the broken curve are of its co-runners. (The points are connected for legibility.) The similarity between the two kinds of curves shows the strong correlations between the degradations of a program and those of its co-runners.

In default Linux SMP scheduling (e.g., Linux 2.6.23), when a program is launched, one of the CPUs will receive that signal and assign the process to the best available CPU for execution. Each CPU has a scheduler managing the jobs assigned to it.

For CAPS, CPUs are classified evenly into two groups, G_s and G_i, dedicated to sensitive and insensitive processes respectively. For the CPUs sharing a cache, half of them belong to G_s and the others belong to G_i. The scheduler on each CPU maintains a sensitivity threshold h, which is equal to the decayed average of the sensitivities of all the processes that the scheduler has assigned (may or may not to this CPU). Formally, h is computed as follows when the scheduler assigns the nth process:

$$h_n = \alpha h_{n-1} + (1 - \alpha) S_n \tag{3}$$

where, α is a decay factor (0 to 1), and S_n is the sensitivity of the newly launched process. The use of the decay factor makes the scheduler adaptive to workload changes. Similar to other factors in OS, its appropriate value should be determined empirically.

When a program is launched, the CPU that receives the launching signal computes the sensitivity of the process, S_n. It then updates h using equation 3. If $S_n > h$, it schedules the process to a CPU in G_s, otherwise, to a CPU in G_i. The way to select a CPU inside a group is the same as in the default Linux scheme. (Stream programs are assigned to G_s directly.) For processes without locality models, the scheduler falls back to the Linux default scheduling.

Equation 3 attempts to obtain load balance by dynamically adjusting threshold h. If unbalance still occurs due to certain patterns in the sensitivities of subsequent jobs, the existing load balancer in Linux, which is invoked periodically, can rebalance the workload automatically.

We note on two facts. First, the scheduler makes no change to the default management of run-queues and timeslice allocation in Linux. This is essential for maintaining the proper treatment to priorities. Second, although it is possible for different CPUs to get different h values, some degrees of difference is tolerable for CAPS. Furthermore, during rebalance, the rebalancer can obtain the average of all CPUs' h values and update the h values for every CPU accordingly.

The sensitivity of a program is obtained from its predicted reuse signature and DPC, both of which have shown to be cross-input predictable [5, 10]. But predictive models have to be constructed for each program through an offline profiling and learning process. This step, although being automatic, may still seem to be a burden to scheduling. There are two ways to make it transparent to the users of CAPS. First, the learning step can occur during the typical performance tuning or correctness testing stage in the development of a software. The program developers only need to run the program on several of the inputs they have; whereas, the outcome is beneficial: Besides for scheduling, the predictive locality model can also benefit data reorganization [5], cache resizing [24], and cache partition [12]. In this case, the scheduler can use the model for free. The second solution is to make the learning occur implicitly in the real runs of an application through incremental learning techniques. Through multiple runs, online learner learns the relation between memory behavior and program inputs, and builds the predictive model for co-run locality prediction. Detailed studies are out of the scope of this paper.

5 Evaluation

For evaluation, we employ 12 randomly chosen SPEC CPU2000 programs, as shown in Table 1, and a sequential stream program (derived from [17] with each data element covering one cache line) on a Dell PowerEdge 1850 server. The machine is equipped with Intel Xeon 5150 2.66 GHz quad-core processors; every two cores have a 4MB shared L2 cache (64B line, 4-way). Each core has a 32KB dedicated L1 data cache. The information shown in Table 1 are collected on the *ref* runs of the benchmarks on the Xeon machine. We use PIN as the instrumentation tool [14] for locality measurement, and use the PAPI [1] library for hardware performance monitoring. In the collection of co-run behavior, in order to avoid the distraction from program lengths, we follow Tuck and Tullsen's practice [33], wrapping each program to make it run 10 times consecutively, and only collecting the behavior of co-runs—that is, the runs overlapping with another program's run.

The focus of our evaluation is the examination of the effectiveness of the unified sensitivity model in serving as a locality model for shared-cache-aware scheduling. To avoid distractions from the many random factors (e.g., job arriving time, load balance) in online schedulers, we use offline measurement to uncover the full potential.

We compute the sensitivities of the programs from their reuse signatures and DPCs, based on which, we separate the 12 SPEC programs into two equal-size classes shown as the two sequences of *caps-pred* below. For comparison, we report the ideal separation as *caps-real*. We obtain them by first running all possible pairs of the 12 programs, and then taking the average co-run degradation of each program as its real

Table 1. Performance Ranges of Benchmarks on Intel Xeon 5150

Program	cycles per instruction			L2 misses per mem. acc.(%)		
	single-run	corun-min	corun-max	single-run	corun-min	corun-max
ammp	1.01	1.03	1.31	0.51	0.60	1.6
art	0.93	0.96	1.55	0.0028	0.095	3.8
bzip	0.49	0.49	0.66	0.11	0.18	0.76
crafty	0.72	0.73	0.80	0.00010	0.0028	0.21
equake	1.28	1.38	2.13	3.8	3.9	4.5
gap	0.91	0.91	1.16	1.3	1.5	1.6
gzip	0.72	0.72	0.77	0.078	0.079	0.14
mcf	2.47	2.70	4.84	4.4	5.0	8.6
mesa	0.51	0.52	0.56	0.23	0.26	0.38
parser	1.15	1.18	1.50	0.31	0.44	1.2
twolf	1.06	1.07	1.24	0.0014	0.0015	0.40
vpr	1.06	1.09	1.44	0.0053	0.0067	0.015

sensitivity. In both separation results, we list the programs in descending order of sensitivity.

caps-pred:

Sensitive:	mcf	art	equake	vpr	parser	bzip
Insensitive:	twolf	ammp	crafty	gap	mesa	gzip

caps-real:

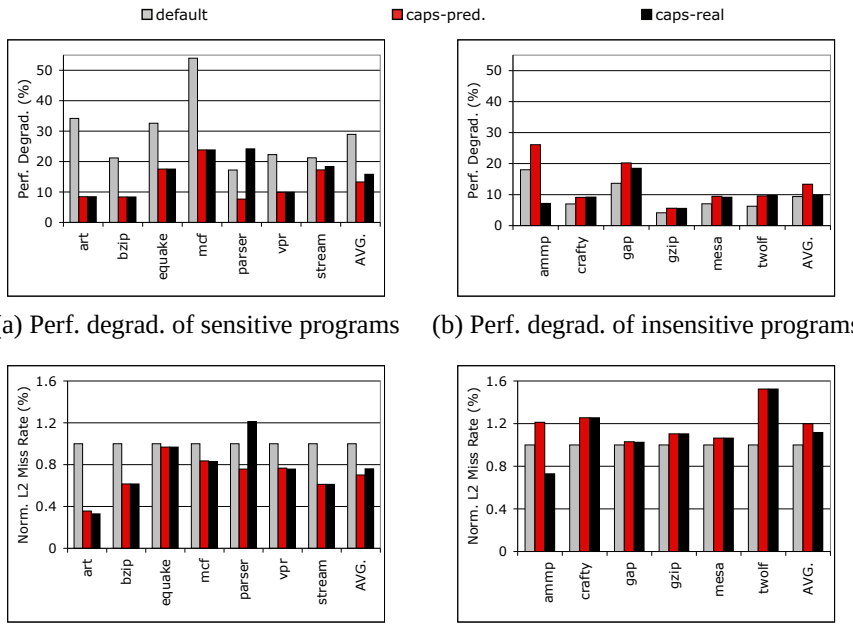
Sensitive:	mcf	equake	art	vpr	bzip	ammp
Insensitive:	parser	gap	crafty	mesa	twolf	gzip

The sequences, although differing in the relative positions of the benchmarks, only mismatch on two programs, *parser* and *ammp*. Two reasons cause the differences: locality prediction errors and the difference between statistical expectation and a particular problem instance. We note that CAPS has good tolerance to ordering difference: As long as programs are put into the right sequences, the order inside a sequence has no effects on CAPS. This property is essential for making the lightweight locality prediction applicable for CAPS.

We compare the performance result of CAPS on predicted sensitivities (denoted as *caps-pred*) with the results of the default Linux scheduler (*default*) and CAPS on real sensitivities (*caps-real*). We measure the performance of a program by **degradation factor**, defined as $(CPI_{co} - CPI_{si})/CPI_{si}$, where, CPI_{co} and CPI_{si} are the respective CPIs of the program's co-run and single run. Following prior work [34], we measure the fairness of a schedule by **unfairness factor**, defined as the coefficient of variation (standard deviation divided by the mean) of the normalized performance (IPC_{co}/IPC_{si}) of all applications.

To prevent randomness from obscuring the comparison, we obtain a program's performance in a schedule by averaging the performance of all the program's co-runs that are allowed by the schedule. The *default* scheduler, for example, allows all 12

possible co-runs per program, whereas *caps-pred* and *caps-real* allow a program to run with only the programs in a different class.



(a) Perf. degrad. of sensitive programs (b) Perf. degrad. of insensitive programs

(c) L2 miss rates of sensitive programs (d) L2 miss rates of insensitive programs

Fig. 3. Performance degradation and normalized L2 miss rates by different scheduling

Figure 3 shows the performance of the three schedulers, with sensitive programs (judged by *caps-pred*) on the left and insensitive programs on the right. For sensitive programs, *caps-pred* reduces performance degradation by 4% to 30.2% (15.7% on average); as a tradeoff, insensitive programs have 1.4% to 8.1% more degradation (4.1% on average). In comparison, *caps-real* shows 2.5% less reduction for sensitive programs and 3.3% more for insensitive programs than *caps-pred*. It is important to note that the goal of job co-scheduling is to increase the overall computing efficiency of the system rather than maximize the performance of each individual program. So it is normal that some programs (e.g. *parser*) perform better in *caps-pred* than in *caps-real*.

Table 2 reports the performance, normalized to the default performance, of each program when they run in *caps-real* and *caps-pred*. The sensitive programs show 12% and 14% speedup on average. All of them have speedup over 11% except *parser* and *stream*. In *caps-real*, *parser* has 6% slowdown because it is classified as insensitive programs and co-runs with sensitive programs. The small speedup of *stream* is consistent with our intuition conveyed in Section 3.2—such programs are competitive but insensitive for their special memory access patterns. It is remarkable that the

significant speedup for sensitive programs comes with almost no slowdown of insensitive programs. The average slowdown is 1% in *caps-real* and 3% in *caps-pred*. The small slowdown is no surprise given that those program are insensitive to cache sharing. The program *ammp* shows 10% speedup in *caps-real* because the scheduler labels the program as a sensitive program and lets it co-run with insensitive programs. The intuition behind the effectiveness of CAPS is that it successfully recognizes the programs to which cache contention matters significantly. By giving an favorable schedule to those programs, CAPS accelerates them without hurting the programs that are not sensitive to cache contention.

Table 2. Whole-Program Speedup Brought by CAPS

Sensitive Programs			Insensitive Programs		
Programs	*caps-real*	*caps-pred*	Programs	*caps-real*	*caps-pred*
art	1.24	1.24	ammp	1.10	0.94
bzip	1.12	1.12	crafty	0.98	0.98
equake	1.13	1.13	gap	0.94	0.94
mcf	1.24	1.24	gzip	0.99	0.99
parser	0.94	1.09	mesa	0.98	0.98
vpr	1.11	1.11	twolf	0.97	0.97
stream	1.03	1.02	–	–	–
Average	1.12	1.14	Average	0.99	0.97

Table 3 contains the overall performance degradation factors and unfairness factors of the schedules. The two *reduction* columns report the relative reduction ratios of *caps-pred* and *caps-real* compared to *default*. Schedule *caps-pred* reduces degradation factor by 32.6% and unfairness factor by 46.9%, respectively 1.3% and 2.4% less than *caps-real*.

Figure 3 (c) and (d) show the normalized L2 miss rates (L2 misses per memory reference) collected using PAPI library [1]. Although they roughly match the performance results, the L2 miss rates impose different influence on the programs. For example, the 52% more L2 miss rates of *twolf* only cause 3.2% performance difference, while 3.3% less miss rates of *equake* reduce 15% performance degradation. This difference is due to bus-contention differences and the different significance of L2 misses. The L2 miss rates of *twolf* are hundreds of times smaller than those of *equake*. This agrees with the fact that both *caps-pred* and *caps-real* label *twolf* insensitive and *equake* sensitive.

These results demonstrate the potential of the locality model in supporting job co-scheduling. The performance of actual on-line schedulers depends on many other factors, such as the job arrival time and order, system load balance and its dynamic adjustment, job priorities, and so forth. Detailed discussion is out of the scope of this paper.

Overhead of CAPS. The major runtime overhead of CAPS consists of the prediction of standalone reuse signatures and the computation of sensitivities, both determined by

Table 3. Overall Performance Degradation Factors and Unfairness Factors

	Performance Deg. (%)		Unfairness (%)	
	factor	reduction	factor	reduction
default	20.0	–	11.6	–
caps-pred	13.5	32.6	6.2	46.9
caps-real	13.4	33.3	6.0	48.5

the granularity of standalone reuse signatures. Since reuse distances smaller than cache size are more critical for CAPS, reuse signatures organize them in linear scale (1K distance per bar), and use log scale for others. Because each bar in a signature corresponds to one linear function, there are $A + log(N/L)$ linear functions to solve in the reuse-signature prediction, where, A is the number of bars in the linear range, N is program data size (the upper bound of reuse distance), and L is cache line width. The computation of sensitivity relies on only reuse distances smaller than cache size, because only those references can be the victims of cache contention. Thus, the time complexity is $O(A)$.
In our experiments, $L = 64$, A is 64 and N is from 32,606 (*crafty*) to 4.1 million (*gap*) with average of 1.0 million. The numbers of linear functions range from 79 to 86 per program. The computation cost of CAPS is negligible.

6 Comparison

Recent years have seen a number of studies on scheduling in CMP. Some concentrate on scheduling threads in a single application. For example, thread clustering [30] tries to recognize patterns of data sharing using hardware performance counters and locates threads accordingly. The technique cannot apply to the problems tackled in this work as no data are shared among jobs. Some studies [13] tackle the scalability and fairness of scheduling on CMP, but without considering interferences on shared cache in the fairness criterion. Some studies [11, 32] conduct theoretical analysis to uncover the complexity of optimal co-scheduling on CMP. They are useful for offline analysis but not for runtime scheduling.
This section concentrates on the studies that schedule independent jobs to reduce the interferences on shared cache. Most of those studies have used simulators (e.g., [6, 8, 21, 25]), whereas, we use a real machine for all the experiments. Furthermore, CAPS has applicability different from previous techniques (elaborated next). We hence concentrate on qualitative comparisons.
First, the applicability of CAPS differs from prior techniques. Unlike techniques based on cache activity vectors or other hardware extensions (e.g., [6, 21, 27]), this work is a pure software solution applicable to existing systems. On the other hand, hardware extensions may reveal fine-grained cache conflicts, complementary to the coarse-grained locality information used in this work.
Previous explorations in scheduling for CMP or SMT rely on either hardware performance counters or offline memory profiling, showing different applicability

from CAPS. The cache-fair scheduling [8] from Fedorova et al. is applicable when the processes have various cache-access patterns and have already been labeled either *cache-fair* or *best-effort*. Its main goal is performance isolation, accomplished by controlling CPU timeslice allocation instead of process assignment. Zhang et al. use hardware counters to guide scheduling on SMP machines without shared caches [34]. Snavely et al. have proposed symbiotic scheduling, which is based on sampling of various co-runs [4, 25], suiting the problems having a small number of jobs and processors. Some explorations use offline collected memory information to guide scheduling [3, 6]. They use the same program inputs for training and testing, not applicable to input-sensitive programs.

CAPS overcomes the above constraints, but requires each process of interest to be equipped with a cross-input predictive locality model (whose construction, fortunately, can be transparent to the users of CAPS as discussed in Section 4). The combination of CAPS with runtime sampling-based techniques may be beneficial: The former overcomes scalability issues, and the latter offers on-line adaptivity. In addition, the combination of CAPS with locality phases [24] may add adaptivity to phase shifts as well.

Second, the program behavior models introduced in this work may benefit other techniques as well. Essentially, the models offer an alternative way to estimate cache requirement and co-run performance, which are exactly the major sources of guiding information used by many cache management schemes.

7 Conclusion

This work, based on the concept of concurrent reuse distance, develops a lightweight locality model for shared-cache contention prediction. The model offers the basis for a runtime contention-aware proactive scheduling system. Experiments on a recent CMP machine demonstrate the effectiveness of the technique in alleviating cache contention, improving both system performance and fairness. On the high level, this work shows the potential of combining program behavior analysis by programming systems and global resource management by operating systems. Interactions between these two layers may also help other issues in computing systems.

Acknowledgment

We are grateful to the anonymous reviewers for their helpful suggestions. We thank Avi Mendelson at Microsoft for his help on the preparation of the final version of this paper. This material is based upon work supported by the National Science Foundation under Grant No. 0720499 and 0811791 and IBM CAS Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or IBM.

References

1. 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.
2. James R. Bulpin and Ian A. Pratt. Hyper-threading aware process scheduling heuristics. In *2005 USENIX Annual Technical Conference*, pages 103–106, 2005.
3. D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pages 340–351, 2005.
4. M. DeVuyst, R. Kumar, and D. M. Tullsen. Exploiting unbalanced thread scheduling for energy and performance on a cmp of smt processors. In *Proceedings of International Parallel and Distribute Processing Symposium (IPDPS)*, 2006.
5. C. Ding and Y. Zhong. Predicting whole-program locality with reuse distance analysis. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 245–257, San Diego, CA, June 2003.
6. Ali El-Moursy, R. Garg, D. H. Albonesi, and S. Dwarkadas. Compatible phase co-scheduling on a cmp of multi-threaded processors. In *Proceedings of the International Parallel and Distribute Processing Symposium (IPDPS)*, 2006.
7. 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.
8. A. Fedorova, M. Seltzer, and M. D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, 2007.
9. L. R. Hsu, S. K. Reinhardt, R. Lyer, and S. Makineni. Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, 2006.
10. Y. Jiang and X. Shen. Exploration of the influence of program inputs on cmp co-scheduling. In *European Conference on Parallel Computing (Euro-Par)*, August 2008.
11. Y. Jiang, X. Shen, J. Chen, and R. Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*, October 2008.
12. S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, 2004.
13. T. Li, D. Baumberger, and S. Hahn. Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. In *Proceedings of ACM Symposium on Principles and Practice of Parallel Programming*, pages 65–74, 2009.
14. C-K Luk et al. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, Chicago, Illinois, June 2005.
15. G. Marin and J. Mellor-Crummey. Cross architecture performance predictions for scientific applications using parameterized models. In *Proceedings of Joint International Conference on Measurement and Modeling of Computer Systems*, pages 2–13, New York City, NY, June 2004.
16. R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM System Journal*, 9(2):78–117, 1970.
17. J.D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE TCCA Newsletter*, 1995. http://www.cs.virginia.edu/stream.

18. S. Parekh, S. Eggers, H. Levy, and J. Lo. Thread-sensitive scheduling for smt processors. Technical Report 2000-04-02, University of Washington, June 2000.
19. N. Rafique, W. Lim, and M. Thottethodi. Architectural support for operating system-driven cmp cache management. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, 2006.
20. S. Sarkar and D. Tullsen. Compiler techniques for reducing data cache miss rate on a multithreaded architecture. In *Proceedings of The HiPEAC International Conference on High Performance Embedded Architectures and Compilation*, 2008.
21. A. Settle, J. L. Kihm, A. Janiszewski, and D. A. Connors. Architectural support for enhanced smt job scheduling. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, pages 63–73, 2004.
22. 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.
23. X. Shen, J. Shaw, B. Meeker, and C. Ding. Locality approximation using time. In *Proceedings of the ACM SIGPLAN Conference on Principles of Programming Languages (POPL)*, 2007.
24. X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 165–176, Boston, MA, 2004.
25. A. Snavely and D.M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Proceedings of ASPLOS*, 2000.
26. Y. Solihin, V. Lam, and J. Torrellas. Scal-tool: Pinpointing and quantifying scalability bottlenecks in dsm multiprocessors. In *Proceedings of the 1999 Conference on Supercomputing*, 1999.
27. G. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *Journal of Supercomputing*, 28:7–26, 2004.
28. G.E. Suh, S. Devadas, and L. Rudolph. Analytical cache models with applications to cache partitioning. In *Proceedings of the 15th international conference on Supercomputing*, 2001.
29. G.E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, 2002.
30. D. Tam, R. Azimi, and M. Stumm. Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors. *SIGOPS Oper. Syst. Rev.*, 41(3):47–58, 2007.
31. D. Thiebaut and H.S. Stone. Footprints in the cache. *ACM Transactions on Computer Systems*, 5(4), 1987.
32. K. Tian, Y. Jiang, and X. Shen. A study on optimally co-scheduling jobs of different lengths on chip multiprocessors. In *Proceedings of ACM Computing Frontiers*, 2009.
33. N. Tuck and D. M. Tullsen. Initial observations of the simultaneous multithreading Pentium 4 processor. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, New Orleans, Louisiana, September 2003.
34. X. Zhang, S. Dwarkadas, G. Folkmanis, and K. Shen. Processor hardware counter statistics as a first-class system resource. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems*, 2007.
35. 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):328–343, March 2007.
36. Y. Zhong, M. Orlovich, X. Shen, and C. Ding. Array regrouping and structure splitting using whole-program reference affinity. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 255–266, June 2004.