

Cache Conscious Task Regrouping on Multicore Processors

Xiaoya Xiang, Bin Bao, Chen Ding and Kai Shen
Department of Computer Science,
University of Rochester
Rochester, NY, USA
{xiang, bao, cding, kshen}@cs.rochester.edu

Abstract—Because of the interference in the shared cache on multicore processors, the performance of a program can be severely affected by its co-running programs. If job scheduling does not consider how a group of tasks utilize cache, the performance may degrade significantly, and the degradation usually varies sizably and unpredictably from run to run.

In this paper, we use trace-based program locality analysis and make it efficient enough for dynamic use. We show a complete on-line system for periodically measuring the parallel execution, predicting and ranking cache interference for all co-run choices, and reorganizing programs based on the prediction. We test our system on floating-point and mixed integer and floating-point workloads composed of SPEC 2006 benchmarks and compare with the default Linux job scheduler to show the benefit of the new system in improving performance and reducing performance variation.

Keywords—multicore; task grouping; online program locality analysis; lifetime sampling

I. INTRODUCTION

Today’s computing centers make heavy use of commodity multicore processors. A typical computer node has 2 to 8 processors, each of which has 2 to 6 hyperthreaded cores. A good number of programs can run in parallel on a single machine. However, the performance depends heavily on resource sharing in particular memory hierarchy sharing.

Memory sharing happens at multiple levels. Main memory management is a well-known and well-studied problem. In this paper, we study the effect of cache memory and propose a solution to improve cache sharing. The problem in cache is reminiscent of that of memory sharing. The level of sharing is different, but the concerns, the hope for utilization and the fear for interference, are similar.

Compared to the problem of memory management, the frequency of events in cache is orders of magnitude higher in terms of the number of memory accesses, cache replacements, and memory bus transfers. On today’s processors, a single program may access cache a billion times a second and can wipe out the entire content of cache in less than a millisecond. The intensity multiplies as more programs are run in parallel.

Cache usage can be analyzed on-line and off-line. On-line analysis is usually counter based. The number of cache misses or other events can be tallied using hardware counters for each core with little or no overhead. Such

online event counts are then utilized to analyze the cache sharing behavior and guide cache-aware scheduling by the operating system [1]–[3]. While the counters can measure the interference for the current mix of tasks, it cannot predict how the interference will change when tasks are regrouped. The analysis itself may add to the event counts and performance interference it is observing.

Off-line analysis uses the access trace. It measures the reuse distance of each access and predicts the miss ratio as a function of the cache size. Trace analysis gives the “clean-room” statistics for each program unaffected by the co-run. It can be composed to predict cache interference — multi-program co-run [4]–[6] or single-program task partitioning [7] —and find the best parallel configuration without (exhaustive) parallel testing.

Trace analysis incurs a heavy cost. Recent studies have reduced the cost through sampling [8]–[14]. A new technique, lifetime sampling, can quantify the inter-program interference in shared cache at run time when programs are running together [15].

In this paper, we apply on-line trace sampling to solve the problem of cache conscious task regrouping. Given a multicore machine with p processors and c cores per processor, and $c \cdot p$ tasks to execute, the goal is to divide the $c \cdot p$ tasks among the p processors to maximize the overall performance. We present a complete on-line setup to regroup a given set of programs for the overall speed, which means to minimize the finish time for the longest running task. A similar setup may be used to maximize the throughput, which means to minimize the average slowdown compared to running each task in dedicated cache with no sharing.

In this work, we consider only parallel workloads of independent sequential programs. Multi-threaded workloads pose the additional problems of data and code sharing and thread interleaving, which we will not consider. We evaluate the effect of task regrouping using SPEC 2006 benchmarks. The results show not just the effect of regrouping but also the accuracy and cost of on-line cache sharing analysis.

II. BACKGROUND

This section explains how trace sampling is used to predict cache sharing. We start with the footprint fp (which we will measure through sampling) and then use the footprint

to derive the lifetime lf , miss rate mr and reuse distance rd . Finally, we combine reuse distance and footprint to predict the effect of cache sharing.

All-window footprint: A footprint is the amount of data accessed in a time period, i.e. a time window. Most modern performance tools can measure a program’s footprint in some windows, i.e. snapshots. Three recent papers have solved the problem of measuring the footprint in all execution windows and given a linear-time solution to compute the average footprint [5], [6], [16].

Let W be the set of $\binom{n}{2}$ windows of a length- n trace. Each window $w = \langle l, s \rangle$ has a length l and a footprint s . Let $I(p)$ be a boolean function returning 1 when a predicate p is true and 0 otherwise. The footprint function $\overline{fp}(l)$ averages over all windows of the same length l . There are $n - l + 1$ footprint windows of length l . The following formula adds the total footprint in these windows and divides the sum by $n - l + 1$.

$$\overline{fp}(l) = \frac{\sum_{w_i \in W} s_i I(l_i = l)}{\sum_{w_i \in W} I(l_i = l)} = \frac{\sum_{w_i \in W} s_i I(l_i = l)}{n - l + 1}$$

For example, the trace “abbb” has 3 windows of length 2: “ab”, “bb”, and “bb”. The size of the 3 footprints is 2, 1, and 1, so $\overline{fp}(2) = (2 + 1 + 1)/3 = 4/3$. The footprint function is not just monotone [6] but also concave [15].

Lifetime: The lifetime of a program is the average length of time that the program takes to access the amount of data equal to the size of the cache c . In other words, assuming we start with an empty cache, the lifetime is the time taken by a program to fill the cache without causing a capacity miss. Lifetime was long used to quantify the virtual memory performance [17]. In this work, we define the lifetime function to be the inverse of the footprint function

$$lf(c) = \overline{fp}^{-1}(c)$$

The derivation is shown visually in Figure 1. From the average footprint curve, we find cache size c on the y-axis and draw a straight line to the right. At the point the line meets the curve, the x-axis value is $lf(c)$. The miss rate is then the gradient at this point, as discussed next.

Miss rate: The miss rate can be approximated using the lifetime. In particular, we compute the average time between two consecutive misses by taking the difference between the lifetime of $c + 1$ and c . Formally, let $mr(c)$ be the capacity miss rate, $lf(c)$ the lifetime, and $im(c)$ the inter-miss time. We have:

$$mr(c) = \frac{1}{im(c)} = \frac{1}{lf(c+1) - lf(c)}$$

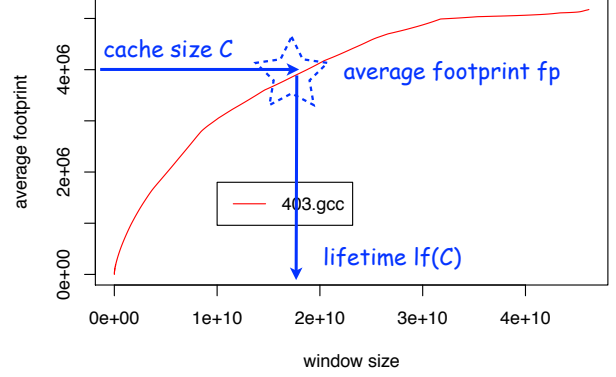


Figure 1. Finding the cache lifetime using the average footprint

Reuse distance: For each memory access, the *reuse distance*, or *LRU stack distance*, is the number of distinct data elements accessed between this and the previous access to the same datum. The reuse distance defines the locality of each memory access. The distribution of all reuse distances gives the capacity miss rate of the program in caches of all sizes [18] and can accurately estimate the effect of conflict misses in direct map and set-associative cache using a statistical formula given by Smith [19]–[21]. From the capacity miss rate, we can compute the reuse distance as

$$rd(c) = mr(c+1) - mr(c)$$

Cache sharing: Off-line cache sharing models were pioneered by Chandra et al. [4] and Suh et al. [22] for a group of independent programs and extended for multi-threaded code by a series of recent studies [7], [8], [23]–[25]. Let A, B be two programs share the same cache but do not share data, the effect of B on the locality of A is

$$P(\text{capacity miss by } A \text{ when co-running with } B) = P((A\text{'s reuse distance} + B\text{'s footprint}) > \text{cache size})$$

Figure 2 shows two program traces first individually and then in an interleaved co-run. Assuming fully associative LRU cache of size 8. The reuse of datum a in program A changes from a cache hit when A runs alone to a cache miss when A, B run together. The model can predict this miss. It takes the reuse distance of a in A and adds the footprint of B to obtain the shared-cache reuse distance. From the reuse distance, we compute the capacity miss rate (and use the Smith formula to estimate for set-associative cache [19]). Therefore, the effect of cache interference, i.e. the additional misses due to sharing, can be computed from single-program statistics. This is known as the composable model because it uses a linear number of sequential tests to predict the performance of an exponential number of parallel co-runs [5].

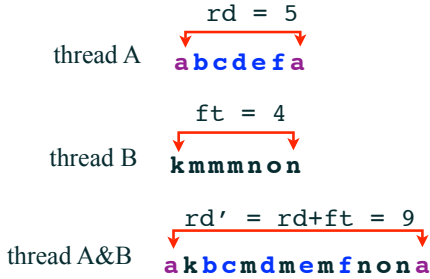


Figure 2. Interference in 8-element shared cache causes the reuse of a to miss in cache. In the base model, the reuse distance in program A is lengthened by the footprint of program B.

If we were to measure reuse distance directly, the fastest analysis would take $O(n \log \log m)$ time, where n is the length of the trace, and m is the size of data [26]. By using the footprint (to derive the reuse distance), the cost is reduced to $O(n)$ [6]. More importantly, footprint can be easily sampled. Sampling is not only fast but also accounts for the phase effect of individual programs and the change in program interaction over time.

III. ON-LINE LOCALITY TESTING

Lifetime sampling per program: We sample a run-time window as long as the lifetime. In other words, we start sampling at a random point in execution and continue until the program accesses as much data as the size of the cache. Specifically, lifetime sampling takes a sample every k seconds for a lifetime window for cache size c . When a program starts, we set the system timer to interrupt every k seconds. The interrupt handler is shown in Figure 3. It forks a sampling task and attaches the binary rewriting tool Pin [27]. The Pin tool instruments the sampling process to collect its data access trace, measures all-window footprint using our technique described in [6], and finds the lifetime $lf(c), lf(c+1)$.

For in situ testing, we do not increase the number of tasks. The sampling algorithm ensures this in two ways. First, the sampling task does not run in parallel with the base task. This is done by the base task waiting for the sampling task to finish before continuing. Second, no concurrent sampling is allowed. Timer interrupt is turned off for the sampling task. The base task sets a flag when a sampling task is running and ignores the timer interrupt if the flag is up.

Predicting the miss rate: For each sample x_i , we predict the miss rate function $mr(x_i, c)$ for each cache size c as follows:

- 1) Use the analysis of Xiang et al. [6] to compute the average footprint function \bar{fp} .
- 2) Compute the lifetime gradient (Section II) to obtain the capacity miss rate for cache size c .

Procedure timer interrupt handler, called whenever a program receives the timer interrupt

- 1: Return if the sampling flag is up
- 2: Set the sampling flag
- 3: $pid \leftarrow fork()$
- 4: **if** $pid = 0$ **then**
- 5: Turn off the timer
- 6: Attach the Pin tool and begin sampling until seeing c distinct memory accesses
- 7: Exit
- 8: **else**
- 9: Reset the timer to interrupt in k seconds
- 10: Wait for pid to finish
- 11: Clear the sampling flag
- 12: Return
- 13: **end if**

Figure 3. The timer-interrupt handler for locality sampling

- 3) Use the capacity miss rate to compute reuse distance distribution and the Smith formula [19] to estimate the number of conflict misses for given cache associativity.

Group sampling per phase: A phase is a unit of time that co-run tasks are regrouped once. Group sampling proceeds in phases. It solves two problems. First, a program collects and analyzes one and only one lifetime sample (trace) in each phase. Second, when all programs have finished sampling, the regrouping routine is called to process the sample results and reorganize the co-run tasks (see the next section).

We use shared memory to coordinate but do so in a distributed fashion without a central controller. When started, each program connects to shared memory of a preset identifier and allocates this shared memory if it is the first to connect. The shared memory contains a sample counter initialized to the number of co-run tasks and a phase counter initialized to 0. When a program finishes collecting a sample, it decrements the sample counter. The last program to finish sampling would reduce the sample counter to 0. It would call the regrouping routine, reset the sample counter, and advance the phase counter. With the two counters, the tasks would sample once and only once in each phase. The pseudo code is shown in Figure 4.

Comparison with reuse-distance sampling: Lifetime by definition is more amenable to sampling. We can start a lifetime sample at any point in an execution and continue until the sample execution contains enough access to fill the size of target cache. We can sample multiple windows independently, which means they can be parallelized. It does not matter whether the sample windows are disjoint or overlapping, as long as the choice of samples is random and unbiased. In contrast, reuse distance sampling must sample evenly for different lengths of reuse windows. When picking

Procedure group sampling, coordinated using shared memory

- 1: {when a program starts}
- 2: connect to shared memory
- 3: **if** shared memory not exists **then**
- 4: allocate shared memory
- 5: $count \leftarrow 8$
- 6: $phase_count \leftarrow 0$
- 7: **end if**
- 8: {when a program finishes a sample}
- 9: lock shared memory
- 10: $count \leftarrow (count - 1)$
- 11: **if** $count == 0$ **then**
- 12: call the regrouping routine
- 13: $count \leftarrow 8$
- 14: $phase_count \leftarrow (phase_count + 1)$
- 15: **end if**
- 16: unlock shared memory

Figure 4. Group sampling per phase

an access, it needs to measure the distance to the next reuse. Since most reuse distances are short, we have to pick more samples. When a reuse distance is long, we do not know a priori how long so we need to keep analyzing until seeing the next reuse. Therefore, reuse distance sampling is more costly than lifetime sampling because it needs more samples and longer samples (a reuse window can be arbitrarily longer than a lifetime window).

IV. DYNAMIC REGROUPING

We use the following terms. A *peer group* is a set of programs that share cache. A *configuration* (or *grouping*) of a set of programs is one way to divide the programs into peer groups. Two groupings differ if their peer groups are not all identical.

The regrouping algorithm is shown in two parts. The first part, shown in Figure 5, takes the sample results, predicts and ranks the performance of all groupings. In this algorithm, we assume that the machine in use is a multicore machine with 2 processors and 4 cores per processor. Our algorithm can be easily generalized to m processors and n cores per processor, where m and n are positive integers. We further assume that the first peer group, denoted by s_1 , includes four programs run on the first processor, and the second peer group, denoted by s_2 , includes the other four programs run on the second processor. The eight programs are denoted as p_0, p_1, \dots, p_7 .

Once a new grouping is selected, we need to move programs between peer groups. Program migration incurs two types of overheads. The first is the direct cost of migration, including the OS delay and re-warming of the cache. The second is the indirect cost that happens on machines that have NUMA memory. When a program is started, its data is

Procedure regrouping routine, called when all programs finish sampling for the current phase.

- 1: d_{old} is the previous grouping
- 2: **for** each $prog$ **do**
- 3: $fp[prog] \leftarrow$ average footprint curves for $prog$
- 4: $rd[prog] \leftarrow$ reuse distance computed from $fp[prog]$
- 5: **end for**
- 6: **for** each even division $d_i = \{s_1, s_2\}$ **do**
- 7: **for** p_i in $s_1 = [p_0, p_1, p_2, p_3]$ **do**
- 8: $mr[p_i] \leftarrow$ shared cache miss rate from
- 9: $(rd[p_i], fp[p_j || p_j \in s_1, j \neq i])$
- 10: $runtime[p_i] \leftarrow time_model(mr[p_i])$
- 11: **end for**
- 12: **for** p_i in $s_2 = [p_4, p_5, p_6, p_7]$ **do**
- 13: $mr[p_i] \leftarrow$ shared cache miss rate from
- 14: $(rd[p_i], fp[p_j || p_j \in s_2, j \neq i])$
- 15: $runtime[p_i] \leftarrow time_model(mr[p_i])$
- 16: **end for**
- 17: $time[d_i] = \max(runtime[p_j || 0 \leq j \leq 7])$
- 18: **end for**
- 19: find d_{new} , where $time[d_{new}] \leq time[d_{j, 0 \leq j \leq 34}]$
- 20: call the remapping routine
- 21: $d_{old} = d_{new}$

Figure 5. The regrouping routine to select the grouping that minimizes the slowest finish time.

allocated in the close-by memory module. Migration would lose the processor-memory affinity and cause memory access to incur additional latency.

For these reasons, we use a remapping routine, shown in Figure 6, to minimize the number of cross-processor program migration. It compares the peer groups in the old and the new grouping. If we assume two peer groups per grouping, we simply check which group assignment has fewer migrations and choose that one.

V. EVALUATION

A. Target Machine and Applications

Our testing platform is a machine with two Intel Nehalem quad-core processors. Each socket has four 2.27GHz cores sharing an 8MB L3 cache. Private L1 and L2 cache are 32KB and 256KB respectively. The machine organizes the main memory in a NUMA structure, and each processor has 4GB 1066MHz DDR3 memory local to it. The machine is installed with Fedora 11 and GCC 4.4.1.

To create a parallel workload, we select the test programs from the SPEC 2006 benchmark suite. To fairly evaluate co-run choices by the longest finish time, we select the programs that have a similar execution time when running by itself on our test machine.

We have selected 12 programs shown in Table I. The targeted time for the stand-alone execution is around 10 minutes. We adjusted the input for several programs to nudge

Procedure remapping routine, called after the regrouping routine to implement the regrouping with minimal cross-socket task migration.

```

1: INPUTS:  $d_{old} = \{s1_o, s2_o\}$ ,  $d_{new} = \{s1_n, s2_n\}$ 
2:  $half = s1_o$ 
3:  $list_1 \leftarrow (half - s1_n)$ 
4: if  $list_1$  has more than 2 elements then
5:    $half = s2_o$ 
6:    $list_1 \leftarrow (half - s1_n)$ 
7: end if
8:  $list_2 \leftarrow (s1_n - half)$ 
9: for  $(p_i, q_i)$  where  $p_i$  and  $q_i$  are the  $i$ -th elements in  $list_1$ 
   and  $list_2$  correspondingly do
10:  swap  $p_i$  and  $q_i$ 
11:  update the cpu id for  $p_i$  and  $q_i$ 
12: end for

```

Figure 6. The remapping routine that minimizes the number of cross-processor program migration.

benchmark (fp)	time (sec.)	benchmark (fp)	time (sec.)	benchmark (int)	time (sec.)
433.milc	530	436.cactus	617	401.bzip2	613
434.zeusmp	704	450.soplex	626	429.mcf	459
437.leslie3d	555	459.Gems	629	458.sjeng	644
444.namd	608	470.lbm	648	462.libquan	693

Table I
BENCHMARK STATISTICS

their run time closer to the target. The stand-alone run time ranges from 530 seconds to 704 seconds for the 8 floating-point programs, shown in the two leftmost groups, and from 459 seconds to 693 seconds for the 4 integer programs, shown in the rightmost group.

We form two workloads from the 12 programs, each with 8 programs. The leftmost 8 programs in Table I form the floating-point workload. The rightmost 8 programs form the mixed workload with both floating-point and integer code. The middle 4 floating-point programs are shared in both workloads.

B. Effect of Task Regrouping

We compare co-run results in two types of graphs. The first plots the finish time of the longest running program in each grouping. The x -axis enumerates all groupings, and the y -axis shows the slowest finish time. We call it a *max-time co-run graph*. For our tests, the x -axis has 35 points for the 35 groupings of 8 programs on two quad-core processors.

The second type of graphs also enumerate all groupings along the x -axis, but the y -axis shows the finish time for all the programs. We connect the points of each program in a line. We call it an *all-time co-run graph*. For our tests, there are 8 lines each connecting 35 points in an all-time graph.

The effect of cache sharing: The main results are shown by the two max-time co-run graphs in Figure 7. The max finish time for all groupings are sorted from shortest to longest. We see that multicore program co-runs have a significant impact in single-program performance. For the mixed workload, the longest execution when running alone is 693 seconds. In the 8-program co-run, the shortest time to finish is 1585 seconds, and the longest 2828 seconds. The results show the strength and the weakness of the multicore architecture. The single-program speed is lower, but the throughput is higher. Cache-conscious scheduling is important because it may improve single-program speed from 24% to 43% of the sequential speed and the parallel throughput from 200% to 350% of the sequential throughput.

The potential benefit is equally significant for the floating-point workload. The longest stand-alone time is 704 seconds. For the 35 co-runs. Cache-conscious scheduling may improve single-program speed from 30% to 50% of the sequential speed and the parallel throughput from 200% to 340% of the sequential throughput.

The difference between the best and the worst co-run performance is 78% for the mixed workload and 69% for the floating-point workload. The choice of task grouping is highly important, considering that the potential improvement is for each of the 8 programs, not just a single program.

Task regrouping for the mixed workload: We tested five runs of task regrouping and five runs of the default Linux scheduling and show the 10 finish times as 10 horizontal lines in the left-hand side graph in Figure 7. Default Linux times, plotted in red, are 1687, 1955, 2560, 2578, and 2844 seconds. Task regrouping times, plotted in blue, are 1764, 1892, 1939, 2043, and 2059 seconds. If we take the geometric mean (to reduce the effect of outliers), the average finish time in the five runs is 2282 seconds for Linux and 1937 for task regrouping. The improvement is 18%.

In addition to being on average faster, the performance variation is smaller from run to run when using task regrouping. The difference between the largest and the smallest numbers in the five runs are 1157 seconds for Linux and 295 seconds for task regrouping, a reduction by a factor of nearly 4 (3.9).

Task regrouping chooses the same grouping in every run. Its performance varies for two reasons. The first is that the current system stops regrouping once one of the 8 programs finishes. The remaining period is at the whim of the default Linux scheduler. The second is the processor-memory affinity, which depends on the initial configuration that varies from run to run.

The all-time co-run graph in Figure 8 shows how individual programs are affected by the program co-run. The groupings are not sorted in the graph. We see that the speed of some programs, in particular *cactus* and *sjeng*, does not vary with the co-run peers (even though the co-run speed is significantly slower than stand alone). The speed of other

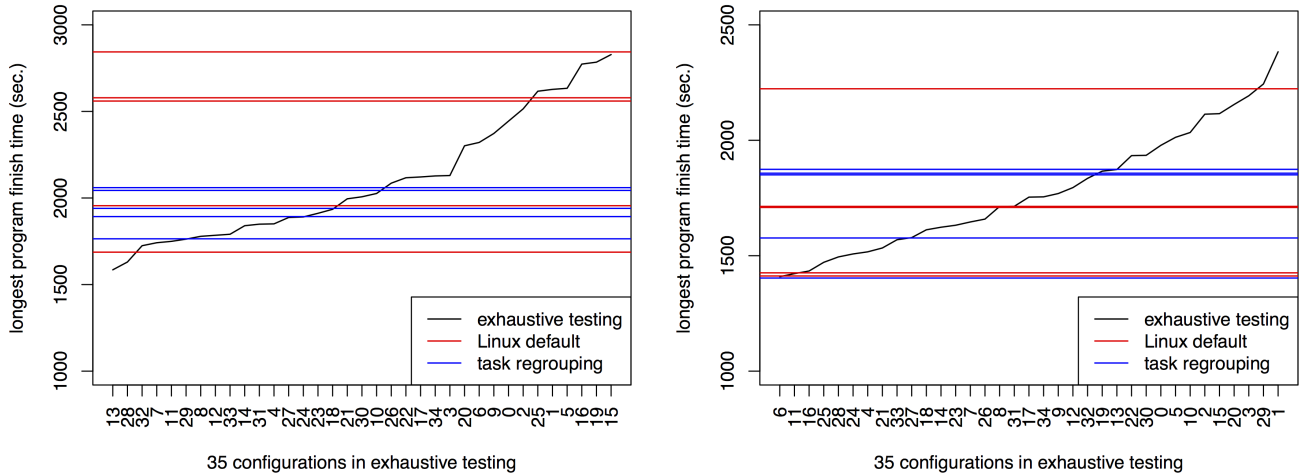


Figure 7. Max-time co-run graphs to compare cache-conscious task regrouping with default Linux scheduling and exhaustive testing. (Left) Mixed floating-point and integer workload. (Right) Floating-point only workload. The choice selected by task regrouping is grouping 27 in both workloads.

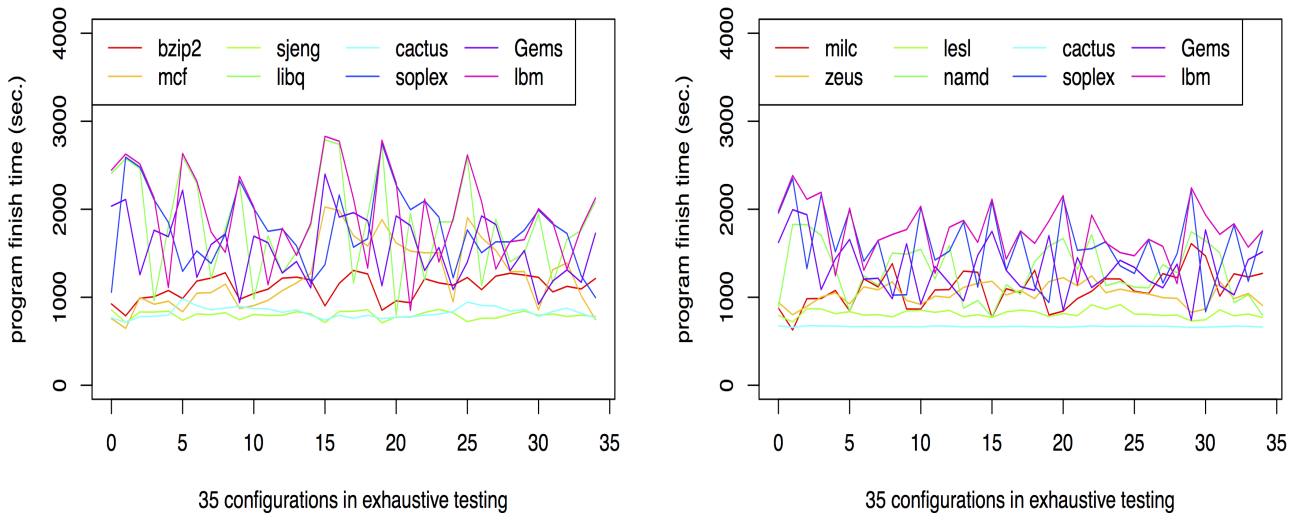


Figure 8. All-time co-run graphs. (Left) Mixed floating-point and integer workload. (Right) Floating-point only workload. The choice selected by task regrouping is grouping 27 in both workloads.

programs, in particular *lbm*, *libq*, and *soplex*, vary by a factor of 2 in performance depending on who their peers are.

The task regrouping chooses grouping 27, which includes *lbm*, *soplex*, *sjeng*, *bzip2* in one peer group and the rest in the other peer group. Each peer group has two floating-point and two integer programs.

Task regrouping for the floating-point workload: For the floating-point workload, task regrouping does not improve the average finish time. Default Linux runs finish in 1412, 1426, 1709, 1713, and 2223 seconds. Task regrouping

runs finish in 1403, 1576, 1850, 1857, and 1874 seconds. The geometric mean is 1673 seconds for Linux and 1701 seconds for task regrouping. The latter is 1.7% slower. However, task regrouping has much smaller variation. The largest difference between the five runs is 811 seconds for Linux and 471 seconds for regrouping. The latter is a factor of 1.7 smaller.

The regrouping chooses also grouping 27, which includes *lbm*, *milc*, *leslie3d*, *namd* in one peer group and the rest in the other peer group. The relatively poor result is partly due to

benchmark	overhead (%)	benchmark	overhead (%)
436.cactus	18.0	401.bzip2	13.7
450.soplex	1.3	429.mcf	1.2
459.Gems	1.3	458.sjeng	5.1
470.lbm	0.9	462.libquan	1.1

Table II
SAMPLING OVERHEAD WHEN REGROUPING FOR THE MIXED
FLOATING-POINT AND INTEGER WORKLOAD

benchmark	overhead (%)	benchmark	overhead (%)
433.milc	1.0	436.cactus	21.1
434.zeusmp	0.6	450.soplex	0.2
437.leslie3d	0.1	459.Gems	0.3
444.namd	31.2	470.lbm	0.1

Table III
SAMPLING OVERHEAD WHEN REGROUPING FOR THE FLOATING-POINT
WORKLOAD

the accuracy of the model and partly due to overhead of sampling, which we discuss next.

C. Overhead of On-line Sampling

Sampling has a direct overhead, because it pauses the base program, and an indirect overhead, because it adds to cache interference and competes for other resources such as memory bandwidth. We have measured the first overhead. Tables II and III show for each program, the total length of the sampling pause as a portion of the total length of the co-run execution.

In the mixed workload, the top three overheads are 18%, 14%, and 5%. The rest are between 0.9% and 1.3%. In the floating-point workload, all overheads are below 1% except for *cactus* 21% and *namd* 31%. The *namd* cost is likely a main reason for the relatively poor performance of task regrouping for the floating-point workload.

VI. RELATED WORK

Cache analysis for HPC systems: Current performance models focus on parallelism, memory demand, and communication pattern. The prevalence of two-level parallelism on multicore systems is widely recognized. A recent example is Singh et al., who characterized a multicore machine as an SMP of CMPs and showed the importance of modeling the two levels in predicting the performance and scalability in multithreaded HPC applications [28]. In their study, the cache performance was measured by counting the L1 cache misses. Sancho et al. evaluated the performance impact on large-scale code from shared and exclusive use of memory controllers and memory channels [29]. A fundamental limitation in performance scaling is memory bandwidth, which depends on how programs share cache on a CMP. Hao et al. proposed a novel processor design that used data migration in shared L2 cache to mitigate the NUMA effect [30]. Cache

migration, although reduces latency, does not directly change the bandwidth demand of a group of tasks.

Most HPC performance tools gather hardware counter results. An example is Open|SpeedShop, which provides a real-time display while an MPI program is running [31]. The emphasis is measurement rather than prediction. HPCView uses a collection of program metrics including the reuse distance and can predict cache performance for different cache sizes and configurations but does not predict the effect of cache sharing [32]. HPCToolkit can analyze optimized parallel code with only a few percent overhead [33]. To control the cost, HPCToolkit does not instrument program data access.

Multicore cache-aware scheduling: Jiang et al. formulated the problem of optimal scheduling and gave a solution based on min-weight perfect matching [34]. Since online scheduling requires low overhead, existing approaches utilize hardware event counters, which can be read at little cost. For instances, Knauerhase et al. [1], Zhuravlev et al. [3], and Shen [2] all advocated using the last-level cache miss rate as the measure of a program’s cache use intensity. The scheduler then tries to group high-resource-intensity program(s) with low-resource-intensity program(s) on a multicore to mitigate the conflicts on shared resources.

However, counter-based approaches have the weakness that the measured cache performance at one grouping situation (with a certain set of co-runners on sibling cores) may not reflect the cache performance in other grouping situations. For instance, a program may incur substantially more cache misses after the peers change because its share of cache space in the new grouping becomes too small for its working set. In contrast, the lifetime sampling approach in this paper properly models the cache sharing in all (hypothetical) program grouping situations. The modeling is realized at an acceptable cost for online management. It collects “clean-room” statistics for each program, unaffected by other co-run programs, program instrumentation or the analysis process itself.

Locality sampling: A representative system was developed by Beyls and D’Hollander [10]. It instruments a program to skip every k accesses and take the next address as a sample. A bounded number of samples are kept—hence the name reservoir sampling. To capture the reuse, it checks each access to see if it reuses some sample data in the reservoir. The instrumentation code is carefully engineered in GCC to have just two conditional statements for each memory access (address and counter checking). Reservoir sampling reduces the time overhead from 1000-fold slowdown to only a factor of 5 and the space overhead to within 250MB extra memory. The sampling accuracy is 90% with 95% confidence. The accuracy is measured in reuse time, not reuse distance or miss rate.

To accurately measure reuse distance, a record must be kept to count the number of distinct data appeared in a reuse

window. Bursty reuse distance sampling divides a program execution into sampling and hibernation periods [9]. In the sampling period, the counting uses a tree structure and costs $O(\log \log M)$ per access. If a reuse window extends beyond a sampling period into the subsequent hibernation period, counting uses a hash-table, which reduces the cost to $O(1)$ per access. Multicore reuse distance analysis uses a similar scheme for analyzing multi-threaded code [8]. Its fast mode improves over hibernation by omitting the hash-table access at times when no samples are being tracked. Both methods track reuse distance accurately.

StatCache is based on unbiased uniform sampling [12]. After a data sample is selected, StatCache puts the page under the OS protection (at page granularity) to capture the next access to the same datum. It uses the hardware counters to measure the time distance till the reuse. Time-based conversion is used in reuse distance profiling [35] and recently modeling cache sharing [14]. Another approach, more efficient but not entirely data driven, is to assume common properties in data access and distinguish programs through parameter fitting [36], [37].

Continuous program optimization (CPO) uses the special support in an experimental IBM system to mark exact data addresses [11]. Subsequent accesses to marked data are trapped by hardware and reported to software. Similar hardware support has been used to predict the miss-rate curve [38] and quantify data locality [39]. Hardware sampling, however, is necessarily sparse or short in order to be efficient. StatCache and CPO use a small number of samples. HPCToolkit is constrained by the hardware limit of 64K events on the AMD machine [39]. Lifetime sampling is based on the locality theory described in Section II. It instruments and collects the data access trace as long as needed based on the cache lifetime. The current implementation is entirely software, which is portable. Lifetime sampling may take advantage of special hardware and OS support if they are available.

VII. SUMMARY

We have presented cache-conscious task regrouping, a system that reorganizes multicore co-run executions to minimize interference in shared cache. We have developed algorithms for lifetime sampling, group sampling, task regrouping, and task remapping. When evaluated using 12 SPEC 2006 benchmarks, cache-conscious regrouping significantly improved over Linux scheduling for mixed floating-point and integer workload while gave a similar result for the floating-point workload. In both workloads, it reduced the run-to-run performance variation by factors of 2 and 4. The on-line sampling overhead was negligible for most of the tested programs but could be as high as 30% for a small number of programs.

ACKNOWLEDGMENT

We would like to thank CCGrid reviewers for carefully reading our paper and providing important feedback to our work. The presentation has been improved by the suggestions from the systems group at University of Rochester. Xiaoya Xiang and Bin Bao are supported by two IBM Center for Advanced Studies Fellowships. The research is also supported by the National Science Foundation (Contract No. CCF-1116104, CCF-0963759, CNS-0834566).

REFERENCES

- [1] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn, "Using OS observations to improve performance in multicore systems," *IEEE Micro*, vol. 38, no. 3, pp. 54–66, 2008.
- [2] K. Shen, "Request behavior variations," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010, pp. 103–116.
- [3] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010, pp. 129–142.
- [4] 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*, 2005, pp. 340–351.
- [5] X. Xiang, B. Bao, T. Bai, C. Ding, and T. M. Chilimbi, "All-window profiling and composable models of cache sharing," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2011, pp. 91–102.
- [6] X. Xiang, B. Bao, C. Ding, and Y. Gao, "Linear-time modeling of program working set in shared cache," in *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, 2011, pp. 350–360.
- [7] M.-J. Wu and D. Yeung, "Coherent profiles: Enabling efficient reuse distance analysis of multicore scaling for loop-based parallel programs," in *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, 2011.
- [8] D. L. Schuff, M. Kulkarni, and V. S. Pai, "Accelerating multicore reuse distance analysis with sampling and parallelization," in *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, 2010, pp. 53–64.
- [9] Y. Zhong and W. Chang, "Sampling-based program locality approximation," in *Proceedings of the International Symposium on Memory Management*, 2008, pp. 91–100.
- [10] K. Beyls and E. D'Hollander, "Discovery of locality-improving refactoring by reuse path analysis," in *Proceedings of HPCC. Springer. Lecture Notes in Computer Science Vol. 4208*, 2006, pp. 220–229.
- [11] C. Cascaval, E. Duesterwald, P. F. Sweeney, and R. W. Wisniewski, "Multiple page size modeling and optimization," in *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, 2005, pp. 339–349.

- [12] E. Berg and E. Hagersten, "Fast data-locality profiling of native execution," in *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, 2005, pp. 169–180.
- [13] D. Eklov and E. Hagersten, "StatStack: Efficient modeling of LRU caches," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, 2010, pp. 55–65.
- [14] D. Eklov, D. Black-Schaffer, and E. Hagersten, "Fast modeling of shared caches in multicore systems," in *Proceedings of the International Conference on High Performance Embedded Architectures and Compilers*, 2011, pp. 147–157, *best paper*.
- [15] X. Xiang, B. Bao, and C. Ding, "Program locality sampling in shared cache: A theory and a real-time solution," Department of Computer Science, University of Rochester, Tech. Rep. URCS #972, December 2011.
- [16] C. Ding and T. Chilimbi, "All-window profiling of concurrent executions," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2008, *poster paper*.
- [17] P. Denning, "Working sets past and present," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 1, Jan. 1980.
- [18] R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM System Journal*, vol. 9, no. 2, pp. 78–117, 1970.
- [19] A. J. Smith, "On the effectiveness of set associative page mapping and its applications in main memory management," in *Proceedings of the 2nd International Conference on Software Engineering*, 1976.
- [20] M. D. Hill and A. J. Smith, "Evaluating associativity in CPU caches," *IEEE Transactions on Computers*, vol. 38, no. 12, pp. 1612–1630, 1989.
- [21] G. Marin and J. Mellor-Crummey, "Cross architecture performance predictions for scientific applications using parameterized models," in *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, 2004, pp. 2–13.
- [22] G. E. Suh, S. Devadas, and L. Rudolph, "Analytical cache models with applications to cache partitioning," in *Proceedings of the International Conference on Supercomputing*, 2001, pp. 1–12.
- [23] C. Ding and T. Chilimbi, "A composable model for analyzing locality of multi-threaded programs," Microsoft Research, Tech. Rep. MSR-TR-2009-107, August 2009.
- [24] Y. Jiang, E. Z. Zhang, K. Tian, and X. Shen, "Is reuse distance applicable to data locality analysis on chip multiprocessors?" in *Proceedings of the International Conference on Compiler Construction*, 2010, pp. 264–282.
- [25] E. Z. Zhang, Y. Jiang, and X. Shen, "Does cache sharing on modern CMP matter to the performance of contemporary multithreaded programs?" in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2010, pp. 203–212.
- [26] Y. Zhong, X. Shen, and C. Ding, "Program locality analysis using reuse distance," *ACM Transactions on Programming Languages and Systems*, vol. 31, no. 6, pp. 1–39, Aug. 2009.
- [27] 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*, Chicago, Illinois, Jun. 2005.
- [28] K. Singh, M. Curtis-Maury, S. A. McKee, F. Blagojevic, D. S. Nikolopoulos, B. R. de Supinski, and M. Schulz, "Comparing scalability prediction strategies on an SMP of CMPs," in *Proceedings of the Euro-Par Conference*, 2010, pp. 143–155.
- [29] J. C. Sancho, M. Lang, and D. J. Kerbyson, "Analyzing the trade-off between multiple memory controllers and memory channels on multi-core processor performance," in *Proceedings of the LSPP Workshop*, 2010, pp. 1–7.
- [30] S. Hao, Z. Du, D. A. Bader, and M. Wang, "A prediction based CMP cache migration policy," in *Proceedings of the IEEE International Conference on High Performance Computing and Communications*, 2008, pp. 374–381.
- [31] M. Schulz, J. Galarowicz, D. Maghrak, W. Hachfeld, D. Montoya, and S. Cranford, "Open—SpeedShop: An open source infrastructure for parallel performance analysis," *Scientific Programming*, vol. 16, no. 2-3, pp. 105–121, 2008.
- [32] J. Mellor-Crummey, R. Fowler, G. Marin, and N. Tallent, "HPCView: A tool for top-down analysis of node performance," *Journal of Supercomputing*, pp. 81–104, 2002.
- [33] L. Adhianto, S. Banerjee, M. W. Fagan, M. Krentel, G. Marin, J. M. Mellor-Crummey, and N. R. Tallent, "HPCToolkit: tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.
- [34] 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*, 2008, pp. 220–229.
- [35] X. Shen, J. Shaw, B. Meeker, and C. Ding, "Locality approximation using time," in *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2007, pp. 55–61.
- [36] K. Z. Ibrahim and E. Strohmaier, "Characterizing the relation between Apex-Map synthetic probes and reuse distance distributions," *Proceedings of the International Conference on Parallel Processing*, vol. 0, pp. 353–362, 2010.
- [37] L. He, Z. Yu, and H. Jin, "FractalMRC:online cache miss rate curve prediction on commodity systems," in *Proceedings of the International Parallel and Distributed Processing Symposium*, 2012.
- [38] D. K. Tam, R. Azimi, L. Soares, and M. Stumm, "RapidMRC: approximating L2 miss rate curves on commodity systems for online optimizations," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009, pp. 121–132.
- [39] X. Liu and J. M. Mellor-Crummey, "Pinpointing data locality problems using data-centric analysis," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2011, pp. 171–180.