Optimal Footprint Symbiosis in Shared Cache

Xiaolin Wang, Yechen Li,
Yingwei Luo and Xiameng Hu
Peking University
Email: wxl@pku.edu.cn

Jacob Brock and Chen Ding
University of Rochester
Rochester, New York

Zhenlin Wang
Michigan Technological University
Houghton, Michigan

***Abstract*—On multicore processors, applications are run sharing the cache. This paper presents online optimization to co-locate applications to minimize cache interference to maximize performance.**

The paper formulates the optimization problem and solution, presents a new sampling technique for locality analysis and evaluates it in an exhaustive test of 12,870 cases. For locality analysis, previous sampling was two orders of magnitude faster than full-trace analysis. The new sampling reduces the cost by another two orders of magnitude. The best prior work improves co-run performance by 56% on average. The new optimization improves it by another 29%. When sampling and optimization are combined, the paper shows that it takes less than 0.1 second analysis per program to obtain a co-run that is within 1.5% of the best possible performance.

I. Introduction

As multi-core processors become commonplace and cloud computing is gains acceptance, applications are increasingly run in a shared environment. Many techniques have addressed the problem of job co-location to minimize interference and maximize performance, including *symbiotic scheduling* [24], co-scheduling [12], contention-aware scheduling [13], [37], task placement [17], and cache-conscious task regrouping [32]. Most techniques use testing and heuristics. In this work we use the terminology of Snavely and Tullsen, call the technique symbiotic scheduling, and solve it as an optimization problem, in part to explore the remaining potential for improvement.

Optimization is difficult given the complexity and dynamics of cache sharing. The traditional metric, the cache miss rate, is measured for a single cache size and not for all sizes. It is insufficient for shared cache because the portion of cache used by a program may be arbitrary. Furthermore, the miss rate does not always correlate with cache contention. Once the memory bandwidth is saturated, the miss rate will stop increasing even though the cache contention can still increase when more programs join a co-run. Finally, the miss rate is not composable. We cannot compute the co-run miss rate from solo-run miss rates.

The research is supported in part by the National Science Foundation (Contract No. CNS-1319617, CCF-1116104, CCF-0963759); IBM CAS Faculty Fellow program; the National Science Foundation of China (Contract No. 61232008, 61272158, 61328201, 61472008 and 61170055); the 863 Program of China under Grant No.2012AA010905, 2015AA015305; the Research Fund for the Doctoral Program of Higher Education of China under Grant No.20110001110101; and a grant from Huawei. Zhenlin Wang is also supported by NSF Career CCF0643664. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding organizations.

This paper develops *optimal footprint symbiosis*. It uses a recent footprint theory to predict aggregate locality of a group of programs from their individual footprints. To enable optimization, it defines a *logical miss ratio* based on a standardized logical time called *common logical time*. It formulates the linearity assumption, which states that co-run slowdown is linearly proportional to the common-logical-time miss ratio. Then optimization is to find the co-run grouping that minimizes total logical miss ratio.

Optimal footprint symbiosis depends on footprint measurement, which is costly. The paper develops *adaptive bursty footprint (ABF) sampling*, which minimizes the cost of measurement for a given precision threshold. The paper makes three main contributions:

1) **Theory:** A theory of optimal footprint symbiosis to combine non-linear locality composition with linear performance optimization.
2) **Technique:** Adaptive bursty footprint sampling to enable dynamic co-run optimization with negligible analysis overhead.
3) **Evaluation:** Evaluation of the cost of sampling and the performance benefit of optimal symbiosis, compared with best previous approaches.

The study has limitations: We only treat cache sharing for applications that do not share data, and the derivation of optimal symbiosis assumes fully-associative LRU cache, even though hardware replacement implementations usually employ set associativity and a pseudo-LRU policy. The second limitation is mitigated, however, by the fact that modern 8-way or higher associativity designs give effectively the same performance as a fully associative cache [5]. In addition, reuse distance can be used to statistically estimate the effect of associativity [23], and as Sen and Wood showed, reuse distance can be used to model the performance of non-LRU policies [22]. Next we introduce the theory of footprint and its relation with reuse distance.

II. Footprint-based Optimal Symbiosis

In this section, we introduce the footprint theory, which we use to compute shared-cache locality. Then we formalize the linearity assumption, which relates shared-cache locality to shared-cache performance. Finally we describe the symbiotic optimization.

A. Background: Footprint Theory

Given a window, the *working set size (WSS)* is the amount of data accessed in the window, i.e. the size of the "active"

data. The working set size may change from window to window. To be deterministic, we define the *footprint* as the average size for all windows of the same length. Given a data access trace of length n, the footprint $fp(t)$ is the average working set size for all windows of length t, $t \in [1..n]$.

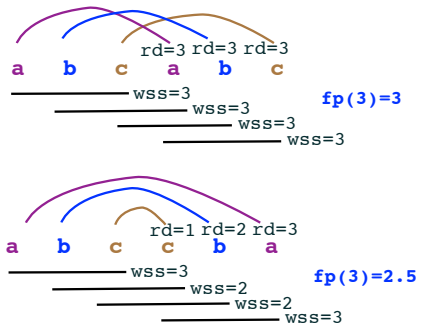

Fig. 1: The reuse distance (rd), working set size (wss) and footprint (fp) of length-3 windows in stack and stream accesses. Both rd and fp quantify locality and show stack accesses have better locality. The difference is that fp is composable across programs while rd is not.

The footprint theory gives the conversion between footprint $fp(x)$ and miss ratio $mr(c)$, where x is the logical time and c is the cache size, using the following formula [33]:

$$mr(c) = fp(x+1) - fp(x)$$

where $c = fp(x)$.

Intuitively, the conversion equates the miss ratio of the cache with the growth rate of the working set, i.e. the working set grows if and only if the cache misses. This is not always true but it is true at the point where the working set size equals the cache size. For fully-associative LRU cache, the working set fills the entire cache. Then the next miss increases the working set by one.

Statistically, footprint is the average working-set size. The difference between footprints, $fp(x+1) - fp(x)$, is the growth of the average working-set sizes. The conversion equates the growth of the average working-set size with the average growth of the working-set size, hence the miss ratio.

Mathematically, the conversion is well defined. Xiang et al. first showed that the footprint as a function is monotone [31]. Then they proved that it is actually concave [33]. As its "derivative", the miss-ratio function is not just non-negative but actually monotone.

B. Common Logical Time Miss Ratio

The parameter x in footprint $fp(x)$ represents time. It can be logical, e.g. memory accesses, or physical, i.e. seconds. For symbiosis, we use the logical time of memory accesses.

The primary benefit of footprint is composability: the footprint of a co-run can be computed from the footprint of solo-runs. However, we must first normalize the logical time of participating programs.

Let g_i be a program in group G, and the access rate ar_{g_i} be the average number of accesses g_i makes per second in a solo-run. In the group run, the access rates are combined. The *common logical time* (or *common time*) is measured in accesses from all programs. The footprint is converted into common time (fp_{ct}) from individual time (fp_{it}):

$$fp_{ct}(g_i, x) = fp_{it}(g_i, \frac{x \times ar_{g_i}}{\Sigma_i\ ar_{g_i}})$$

After conversion, the group footprint is simply the sum of individual footprints.

$$fp(G, x) = \Sigma_i\ fp_{ct}(g_i, x)$$

The co-run miss ratio $mr(G, c)$ is the derivative of the group footprint, using the formula given earlier. The only difference is that the logical time is changed from individual to common time.

C. From Miss Ratio To Execution Time

The symbiotic optimization assumes that *a program's co-run slowdown is linear to the group's logical co-run miss ratio*, where the logical co-run miss ratio is based on common logical time. We call this the *linearity assumption*.

Let g_i be a program in group G, $mr(G, c)$ the group co-run miss ratio in shared cache of size c, $t_{so}(g_i, c)$ the solo-run time of the program with dedicated cache, $t_{co}(g_i, G, c)$ the co-run time of the program, and α a constant, we assume:

$$slowdown(g_i, G, c) = \frac{t_{co}(g_i, G, c) - t_{so}(g_i, c)}{t_{so}(g_i, c)} = \alpha \times mr(G, c)$$

The linearity assumption enables performance ranking. Intuitively, the miss ratio measures cache contention. The higher the miss ratio is, the greater the cache contention is, and the slower a program executes because of the contention.

The miss ratio $mr(G, c)$ is logical. It can increase without a bound, so can the co-run time $t_{co}(g_i)$, e.g. when too many programs overtax the cache. In contrast, a miss rate in physical time, i.e. misses per second, has an upper bound set by the machine memory bandwidth. Once the memory bandwidth is saturated, the miss rate will stop increasing even though the cache contention can still increase when more programs join a co-run.

Consider a program which chooses one of the groups to join. The linearity assumption states that the best choice is the group that has the lowest co-run miss ratio, because the lowest miss ratio implies the lowest co-run slowdown.

The assumption simplifies the complex phenomenon of performance, to which many factors contribute. We model shared cache by the miss ratio, but different misses have different time costs. On modern processors, the timing effect is increasingly complex. Sun and Wang categorized a large number of factors including pipelining, prefetching, multi-bank cache, non-blocking cache, out-of-order execution, branch prediction, and speculation [25].

The linearity assumption classifies performance factors in two types. The first type is unaffected by cache sharing; for example, instruction parallelism and the CPU clock frequency. The other type depends linearly on cache sharing. For example, the contention due to memory bandwidth is assumed to grow linearly with the cache contention.

A linear model is a simplification, but it has important benefits for optimization. For example, all performance factors are considered: it is fine that we do not know all linear factors because we do not need the precise co-efficient of the linear relation. For optimization, it is sufficient that the aggregate effect is linear.

D. Optimal Symbiosis

Optimal symbiosis divides a set of programs into co-run groups to minimize their slowdown. Symbiosis is communal, not individual. Individually, it is sufficient for a program to join a group with the lowest cache contention. Given a set of tasks, we must run every task, so the problem is to run all tasks with the least slowdown.

Given a set of programs $G = \{g_i\}$. Let $fp(g_i, x)$ be the common-time footprint. Let $s = \{G_j\}$ be a valid schedule, where every program g_i is assigned to a co-run group G_j sharing the same cache. As will become clear in Section II-E, it is useful to define a quantity for a schedule called the *aggregate miss ratio*. This is simply the sum of all co-run group miss ratios in that schedule:

$$mr(s, c) = \Sigma_j mr(G_j, c)$$

.

For example, if a schedule has two co-run groups, the aggregate miss ratio is the sum of the two co-run miss ratios. Note that the aggregate miss ratio is not a *miss ratio* in the strict sense; it can be greater than 1. It is simply an indicator metric for comparing the performance between different schedules.

Among all possible schedules, a *symbiotic* schedule is the one that has the lowest aggregate miss ratio. This is the goal of symbiotic optimization. The groups in a symbiotic schedule are symbiotic groups.

The optimality is a conjecture. It depends on the linearity assumption. Moreover, it has to deal with the following problems.

Non-uniform Slowdowns: There are two types of non-uniform slowdown: Inter-group non-uniform slowdown, and intra-group non-uniform slowdown.

In inter-group non-uniform slowdown, all programs within the same group have the same slowdown, but different groups have different slowdowns. By the linearity assumption, optimality holds in this case: if the sum of the miss ratios $\Sigma mr(g_i)$ is minimal, the total slowdown $\Sigma \alpha mr(g_i)$ is minimal, even though each group g_i has a different co-run miss ratio and a different slowdown.

In intra-group non-uniform slowdown, different programs in the same group have a different slowdown. This scenario, however, is not consistent with the linearity assumption.

Perturbation Free Testing: In real uses, we may not be able to measure individual footprint in a solo execution. When tested in a co-run environment, the access rate of a program is affected by its peers Because of the linearity assumption, we can compute the solo-time access rate from the co-run access rate. In this paper, however, we use the access rate from solo-executions.

E. The Performance Ranking Model

The goal of symbiosis is to optimize the interaction between programs in shared cache. While the objective is absolute performance, the model is relative performance. Instead of predicting the performance directly, the model predicts the ranking of performance. We call it the *ranking model*.

It is well established that the effect of shared cache interaction is asymmetrical and non-linear. The ranking model divides this complexity into two parts. The first is non-linear miss-ratio composition, building on the footprint theory and extending it with the common-time logical miss ratio. The second is linear performance correlation. The following summarizes the key components and their relations:

- *Common logical time*, which allows us to compute (add) and compare logical miss ratios.
- *Logical miss ratio*, which includes co-run miss ratio (per group) and aggregate miss ratio (summed over multiple groups). Co-run miss ratio represents cache contention. Aggregate miss ratio represents the performance of co-run grouping.
- *Footprint theory*, which allows us to compute the co-run and aggregate miss ratio from individual footprints without parallel testing.
- *Linearity assumption*, which establishes the relation between co-run performance and the logical miss ratio.
- *Footprint sampling*, which will measure individual footprint efficiently in real time.

III. Adaptive Bursty Footprint (ABF) Sampling

The perennial problem of a sampling method is the tradeoff between the cost and accuracy. The goal is to obtain the best accuracy with the least cost. That is achieved by controlling two parameters: the length and frequency of sampling.

In program analysis, the common technique is called *bursty sampling* [1], [4], [7]. Each sample is a burst, and the dormant period between two samples is called hibernation. The cost of a sampling depends on the length of a burst and the length of hibernation between bursts. We use the terms *burst interval* and *hibernation interval* and symbolize them as *bi* and *hi*.

We build our solution on bursty sampling and add three novel techniques—approximation, bounded cost and adaptation, all designed to specialize for a cache environment. We call it *adaptive bursty footprint sampling* or ABF sampling for short. It collects the sampled footprint *sfp*.

ABF sampling approximates the miss ratio for the target cache. Let the cache size be c. Applying the conversion in Section II-A, we take the sampled footprint *sfp* and predict

the miss ratio $pmr(c)$. The prediction approximates the actual miss ratio $amr(c)$. We set an approximation threshold h, which is the bound on the prediction error:

$$|amr(c) - pmr(c)| \leq h$$

The threshold is absolute, not relative. For example, if $h = 0.01$, the error is 1% miss ratio not 1% of the miss ratio.

The threshold h is the minimal miss ratio we would predict, i.e. $pmr(c) \geq h$. The limit in accuracy allows us to set a limit on the length of a sample, i.e. the burst interval *bi*, in number of accesses. The interval is just long enough to measure the miss ratio of h or higher. Here "long enough" means to fill the cache during the interval with cache misses, i.e. $pmr(c) \times bi \geq c$. At the minimal miss ratio h, we have $h \times bi = c$. Hence, we set the sample length:

$$bi = c/h$$

The second technique is bounded cost. The bound is relative, i.e. no more than 1% of the time of the execution. This is achieved by not sampling too frequently. We represent the frequency by the ratio of hibernation to burst interval, $\frac{hi}{bi}$. If the hi/bi ratio is 1000, the sampled execution is about 0.1% of the total execution. In ABF sampling, we use the hi/bi ratio to bound the cost. The higher the ratio is, the lower is the maximal cost.

The third technique is adaptive sampling. After each hibernation, ABF sampling checks the actual miss ratio and compares it with the prediction. A new sample is collected if and only if the difference is no more than h', which we call the *phase-change threshold*.

The algorithm for ABF sampling is given in Algorithm 1. As in bursty sampling, the procedure is called after each hibernation. The branch at Line 3 tests for the prediction error and decides whether to take a new sample or not. Sampling is done by forking a second process and attaching the Pin tool to it. Footprint analysis is the same as Xiang et al. [33], so is the use of fork. Such parallel analysis has been pioneered by shadow profiling [19], [27]. There are problems of safety, e.g. the forked process should not perform I/O, and concurrency, when sampling threaded code, but these problems are orthogonal to ABF sampling.

In summary, ABF sampling takes 4 parameters. The first two are the cache size c and the approximation threshold h, which are used to set the sample length bi. The third is the hi/bi ratio, which is used to set the length of hibernation hi. The fourth is the phase-change threshold for adaptive sampling. The total cost is bounded by the hi/bi ratio, but may be much lower because of the adaptation. We will show that most of test programs need just one sample.

IV. Evaluation

The preceding sections have presented symbiotic optimization, including the model of shared-cache performance and the technique of footprint sampling. This section evaluates the model, the technique, and the effect of optimization, comparing them with alternative solutions of locality profiling and co-run optimization.

Algorithm 1 Adaptive bursty footprint (ABF) sampling

Require: This procedure is called after each hibernation. The L2 cache size is c. The phase-change threshold is h'. Initially $pmr(c) = 0$. The output is sampled footprint sfp.

```
1:  obtain amr(c) using the hardware counter
2:  compute pmr(c) using sfp
3:  if pmr(c) > 0 and |amr(c) − pmr(c)| ≤ h' then
4:     update sfp using the last sample
5:     { return to hibernation }
6:  else
7:     { take a new sample }
8:     pid ← fork()
9:     if pid = 0 then
10:       attach Pin and sample for bi accesses
11:       update sfp using the new sample
12:       exit {terminate sampling process}
13:    end if
14: end if
15: reset the timer to interrupt after hibernation
```

A. Experimental Setup

We exhaustively test the co-run schedules for 16 programs (arbitrarily chosen) from SPEC 2006: *perlbench, bzip2, mcf, zeusmp, namd, dealII, soplex, povray, hmmer, sjeng, h264ref, tonto, lbm,omnetpp, wrf, sphinx3*. We use the first reference input provided by SPEC.

We use an Intel(R) Core(TM) i7-3770 with four cores, 3.40GHz, 25.6GB/s bandwidth, 256KB private L2, and 8M shared LLC, with prefetching enabled. It runs Fedora 18 and GCC 4.7.2.

We enumerate all 8-program subsets, which gives us 12870 subsets. Each subset is divided into two co-run groups in the way that maximizes the performance of the 4-core machine. In implementation, we do not need to run 12870 tests. Instead, we only test all 1820 of the 4-program groups, and record their performance. Then for each 8-program scheduling, we simply compute the performance of different scheduling choices.

The test programs have different running times. To measure co-run performance, we run each program repeatedly and measure its speed when it is overlapping with other programs, a method used in previous work [13], [24], [33], [34]. The method produces stable results and avoids the problem of run-to-run performance variation. This variability can be predicted using the method of Sandberg et al. to model the effect of different overlappings of applications' phases [20]

Let $G = \{g_i\}$ be a set of co-run programs. Let the co-run and sequential times be $corun(g_i), solo(g_i)$. We define the co-run slowdown of the group by the *sum* of the individual slowdowns:

$$slowdown(G) = \sum_{i \in G} slowdown(g_i) = \sum_{i \in G} \frac{corun(g_i)}{solo(g_i)}$$

B. Linear Relation Between Miss Ratio and Co-run Slowdown

Symbiotic optimization is formulated assuming a linear relation between the common logical time miss ratio and

co-run performance. We evaluate this assumption before the optimization.

Figure 2 plots 1820 co-run groups (all 4-program subsets of 16 benchmarks). Each group is a point, for which the x-axis shows the logical miss ratio of the group, and the y-axis shows the co-run slowdown of the group. The miss ratio ranges from 0% to 1.2%. The slowdown ranges from 4 to 11. A slowdown of 4 means no program is affected by co-run, and 16 or larger means that parallel execution is slower than sequential execution.

Co-run group miss ratio shows a consistent correlation with co-run slowdown: the higher the miss ratio, the greater the slowdown. The correlation coefficient is 0.88. We see two distinct rates in the correlation, divided horizontally at $x = 0.6\%$. We run linear fitting in both groups and combine them into an adjusted relation. The adjusted correlation has a correlation coefficient of 0.938 and is almost linear, as shown in Figure 3.

When the miss ratios are similar, a higher miss ratio does not always mean a greater slowdown. When the miss ratios differ significantly, the cache effect becomes dominant. For example, in Figure 2, the best (lowest) slowdown for a group with 0.8% miss ratio is higher than the worst (highest) slowdown at 0.6% miss ratio. In Figure 3, the worst slowdown at $x = 5$ is better than the best slowdown at $x = 6.5$.

Weak Correlation with Co-run Miss Rate: Miss rate is based on physical time, i.e. misses per second. The x-axis in Figure 2 is overloaded with the actual miss rate. The figure plots the 1820 co-run groups with their co-run miss rate, and with the sum of their solo-run miss rates. The correlation coefficients are 0.48 and 0.65 respectively. Unlike logical miss ratio, miss-rate correlation shows multiple trends (models). Unless there is a way to assign the right program to the right model, we conclude that miss rate is not usable for symbiotic optimization.

The inherent problem is the physical time. While co-run slowdown is unbounded, the miss rate is bounded (by the memory bandwidth). One may argue that we can use the hardware counters to measure the miss ratio in real-time, but it defeats the purpose since our goal is to optimize the miss ratio without exhaustive testing.

We do not show measured miss ratios in the plots because our machine has prefetching, and no hardware counter that can measure ever memory transfer.

Solo-run Miss Rates Cannot Compose Co-run Miss Rate: It is clear that we cannot predict the co-run miss rate by the sum of the solo-run miss rate. In a recent survey paper, Ding et al. showed that miss rate is not composable [11]. Here is an empirical confirmation that solo-run miss rate is not usable for symbiotic optimization.

Linearity Assumption Validated: The correlation started as a conjecture in Section II-C and now has been verified by experiments. It shows that although the performance of modern software and hardware is exceedingly complex, the effect of cache sharing will become the dominant factor, as more programs share cache. When it does, its effect is largely linear. The linearity assumption is now an observation, and it is the scientific basis for the subsequent optimization.

C. Footprint Sampling

In all experiments, we set the approximate threshold to 1% and hi/bi ratio to 1000. Since the size of shared cache on our test machine is 8MB, the correct parameters are 10^7 accesses per sample and 10^{10} hibernation between samples. In more detail, there are $8M/64 = 131072$ cache lines. To fill the cache with at least 1% miss ratio, we need $131072/0.01 = 1.3 * 10^7$ memory accesses, hence the choice of $bi = 10^7$. The phase-change threshold h' is 1%.

To compare with alternatives, we evaluate under-sampling, where the sample length is 10 times shorter, and over-sampling, where the length is 10 times longer. The three methods are shown in Table I.

configurations	*bi* (accesses)	*hi* (accesses)
ABF sampling	10^7	10^{10}
under-sampling	10^6	10^9
over-sampling	10^8	10^{11}

TABLE I: Burst/hibernation intervals of 3 sampling methods

Table II shows the the sample cost for the 16 benchmarks, as well as the solo execution time (without analysis) and the number of samples collected by ABF. ABF sampling takes 0.1 second or less for all programs except for one, *soplex*, which takes 0.12 second. Under-sampling and over-sampling are roughly 10 times faster and slower respectively, because the cost of analysis is linear to the length of the sample.

bench-mark	t-solo (sec)	over-sampling		ABF		under-sampling	
		p	t (sec)	p	t (sec)	p	t (sec)
h264ref	51	1	0.77	1	0.076	1	0.0082
bzip2	78	1	0.78	1	0.075	16	0.0081
soplex	121	1	1.2	1	0.12	10	0.01
povray	137	1	0.85	1	0.083	1	0.0082
perlbench	148	1	0.76	1	0.073	3	0.0097
hmmer	179	1	0.76	1	0.074	1	0.0078
lbm	214	1	0.88	1	0.086	9	0.0089
mcf	232	1	1.3	7	0.102	15	0.011
dealII	242	3	0.82	5	0.08	23	0.0084
omnetpp	279	1	1.01	1	0.1	11	0.01
zeusmp	322	1	0.78	17	0.08	17	0.0081
namd	323	1	0.86	1	0.085	1	0.008
sjeng	423	1	0.89	1	0.087	1	0.0098
wrf	431	4	0.79	6	0.079	52	0.008
sphinx3	461	1	0.86	1	0.087	11	0.0087
tonto	485	5	0.89	5	0.09	46	0.01
arith avg	257	1.56	0.88	3.18	0.086	13.62	0.0089

TABLE II: Cost (t) and phase count (p) of ABF sampling, compared with under- and over-sampling. Programs are sorted by the solo execution time.

Table II shows the average solo execution time of 260 seconds. For the same benchmark set, Xiang et al. reported in 2013 average slowdowns of 38, 153, and 23 for full-trace simulation (of one cache configuration), reuse distance and footprint analysis [33]. A simple extrapolation suggests that for an average program, the average analysis times are 3 hours for simulation, 11 hours for reuse distance and 1.6 hours for footprint.

The sampling technique by Xiang et al. had 0.5%, or 1.3 seconds visible cost and 19%, or 50 seconds total (not hidden by parallel profiling) [33]. It was the best solution,

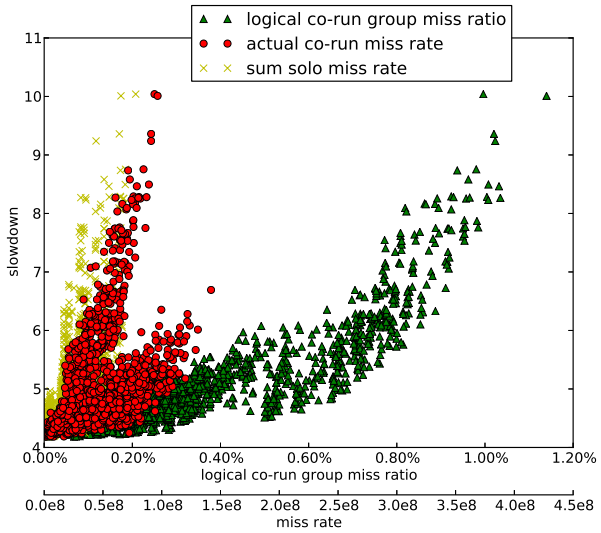

Fig. 2: Correlation with the co-run slowdown by three types of miss metrics: logical co-run miss ratio, actual co-run miss rate, sum of solo-run miss rate

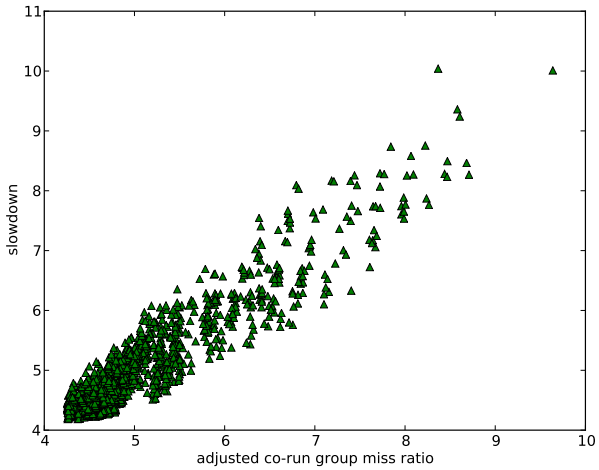

Fig. 3: Correlation between adjusted co-run group miss ratio and co-run slowdown. The correlation coefficient is 0.938.

outperforming the fastest full-trace analysis by two orders of magnitude. Using a total of less than 0.1 seconds on average, ABF sampling is by far the fastest, ushering in another two orders of magnitude improvement.

We compare the miss ratio for all cache sizes up to 8MB (at the increment of one cache block). Since the test machine has 8MB cache, it is sufficient if we can predict the miss ratio for these cache sizes. The collection of miss ratios is called the miss-ratio curve (MRC). Measurement based solutions, e.g. cache partitioning [15], measure the MRC for a handful of values. Footprint and reuse distance MRCs have a much greater resolution.

We show the comparison for the 16 test programs with one graph each in Figure 4. Each graph shows 5 MRC results. To make it easy to distinguish, points of the same MRC are connected into a curve. There are 5 curves in each graph comparing 5 techniques. Two are full-trace analysis, including reuse distance and footprint. As estimated in the last section, their overheads are 11 hours and 1.6 hour respectively. Reuse distance MRC is completely accurate (for fully associative LRU cache). Footprint MRC requires the reuse hypothesis to be correct [33]. The goal of sampling is to obtain the full-trace footprint MRC.

The miss ratios of the first 7 programs are mostly over 1%. ABF sampling produces close results as full-trace footprint analysis. In *lbm*, sampling captures the near vertical drop of miss ratio from over 6% to under 4% when the cache size is around 6MB. In *sphinx3*, full-trace analysis shows a steep (but not vertical) drop of miss ratio from over 4% to near 0% when the cache size increases from 2MB to 8MB. ABF sampling shows a vertical drop from 4% to 1% when the cache size is 4MB. The errors at larger cache sizes are within 1%. The worst error happens in *soplex*. The miss ratio follows an even decline from 11% to 5%. ABF miss ratios are 3% higher. However, the error is almost constant. The prediction captures the variation almost perfectly.

The sharp decline in miss ratio at a narrow range of cache sizes, e.g. over 2% in *lbm* and 4% in *sphinx3*, is the cause of dramatic performance fluctuation programmers observe in shared-cache co-runs. It is important for an online analysis to identify such cases, which ABF sampling does.

In *perlbench*, the reuse distance miss ratio is mostly under 0.2%. However, it is a singular case that full-trace footprint mis-predicts, showing over 0.8% miss ratios for cache sizes up to 5MB. Interestingly, ABF shows near 0 miss ratios for all cache sizes, which is relatively more accurate than full-trace footprint. The reason is that the conversion from footprint to miss ratio is usually but not always accurate. The results of *perlbench* suggest that the accuracy may be improved through sampling.

The miss ratios in the other 8 programs are mostly under 1%. ABF sampling is not configured to give accurate results. Still, the sampling results are mostly accurate, including the capture of 0.3% sharp drop of miss ratio in *h264ref* at 2MB cache size, 0.5% drop of miss ratio in *hmmer*, and near perfect prediction in *sjeng* in all cache sizes. In *bzip2* and *dealII*, ABF sampling does not detect the miss ratio drop but instead predict the (basically) correct miss ratio for larger cache sizes.

The other sampling methods are not as cost effective. Over-sampling is more accurate when miss ratios are small, e.g. around 0.4% in *namd* and near 0% in *povray*. This is expected because over-sampling is the same as higher precision, i.e. setting approximation threshold to 0.1%. However, the improved accuracy, 0.4% in *namd* and 0.2% in *povray*, comes at 10 times the cost. Furthermore, since the sample length is 10 times greater, the hibernation length is also 10 times longer. As a result, over-sampling performs much worse on all programs with phase behavior, *mcf, wrf, zeusmp*. In *zeusmp*, over-sampling predicts almost 1% below the actual miss ratio, while ABF is almost entirely accurate (and finds 17 phases, shown in Table II).

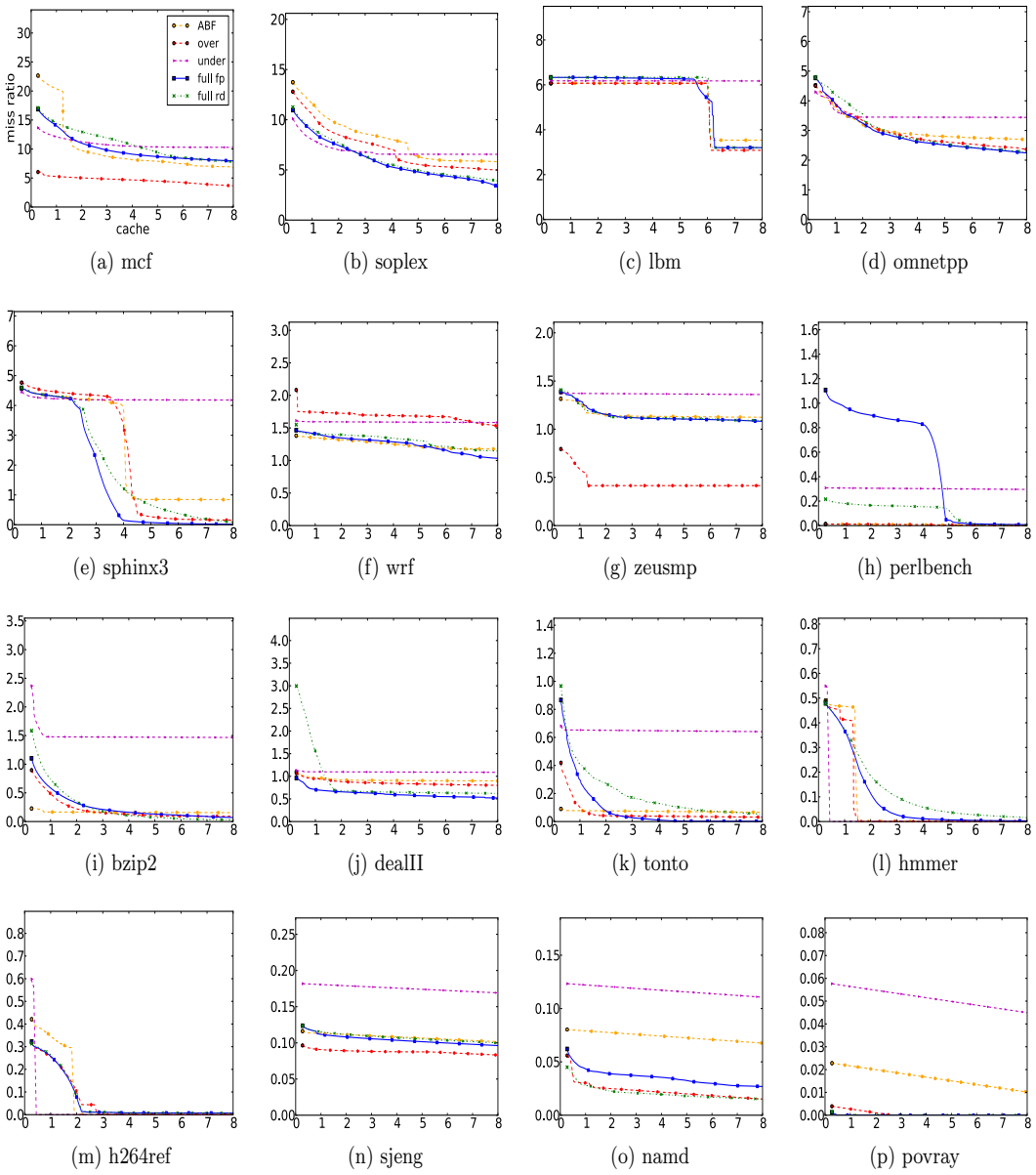

Fig. 4: Miss ratios of the 16 test programs in fully-associative LRU cache of all sizes from 256KB to 8MB, including the accurate results by full-trace reuse distance analysis (full rd) and approximate results by full-trace footprint (full fp) and three sampling techniques: ABF, over- and under-sampling. The effect of cache size is often non-linear, which ABF captures the non-linear locality with 0.09 second overhead (Table II).

Under-sampling is faster than ABF but the precision is significantly worse. The results give strong evidence that ABF sampling is the minimal-cost solution for precise prediction.

D. Symbiotic Optimization

We first describe a set of optimization methods and then compare their performance.

1) Bounded-Bandwidth Symbiosis: Footprint symbiosis as described in Section II-D finds the co-run schedule whose total miss ratio is minimal. In practice, however, memory bandwidth plays a decisive role as shown earlier in Figure 2. The co-run slowdown increases at slower rate when the miss ratio is below 0.6% (when the bandwidth starts to saturate) and then at a much faster rate afterwards. In symbiotic optimization, we set an upper bound on the miss ratio. We first require that no co-run miss ratio exceed the threshold in the symbiotic schedule. If no such schedule exists, we gradually lift the threshold until we find a schedule. Based on the results in Figure 2, we set the initial threshold, which is 37% of the peak memory bandwidth on the test machine.

Without the bandwidth ceiling, an optimal schedule may be arbitrarily unbalanced. For example, when dividing 4 programs into two pairs, the total miss ratio may be the lowest but one pair incurs all the misses and over-burden the memory bus. However, upper bounding differs from balancing. Under the

bandwidth ceiling, the best schedule may still be unbalanced. This is an important consequence of non-linearity when optimizing for shared cache. Because the aggregate effect is non-linear, imbalance is required for optimization.

2) Distributed Intensity (DI): Distributed intensity (DI) was developed by Zhuravle [37]. It sorts co-run programs by decreasing solo-run miss rate and assigns them round-robin into groups. The resulting schedule effectively balances the sum of the miss rates in each group. For reproducible results, we use the miss rate measured in complete solo executions. As discussed earlier, solo-run miss rates are not composable. However, this is not a serious problem for DI because it does not compute the co-run miss rate. Its goal is to balance rather than to optimize. It banishes imbalance but does not rank the remaining schedules (by shared-cache performance). For example, in group $G = (mcf, mcf, libquantum, dealII)$, mcf has the highest solo-run miss rate. When dividing them into two pairs, grouping $mcf, libquantum$ gives 3% worse performance than grouping $dealII, libquantum$, but DI cannot differentiate between them.

3) Co-run Optimization: There are 12870 subsets of 8 programs in our test suite of 16 programs. We take each 8-program set as a scheduling problem: how to divide the 8 programs to run on our 4-core test machine to minimize the total slowdown. Because of the testing strategy (i.e. running each program multiple times), the length difference among test programs does not matter.

For each test, there are 35 possible schedules.[1] We rank them and for each one, calculate the relative slowdown as the difference with the best schedule. Symbiotic optimization is to choose the schedule with 0 relative slowdown.

Figure 5 shows the (cumulative) distribution of relative slowdowns in all 12870 tests, with the slowdown on the x-axis and the cumulative percentage on the y-axis. An (x, y) point means that y portion of relative slowdowns is less than or equal to x.

Full-trace symbiosis has the best performance. 11% of its relative slowdowns are 0 (optimal), 62% within 0.1, and 92% within 0.2. ABF sampling is the next best, equally as good as the best for 90% of tests. The deviation for the remaining 10% is small.

DI shows the performance of balancing. Among all tests, 4.8% is optimal, 38% within 0.1, and 62% within 0.2. DI is as good as optimization for half of tests, which means that balancing is sufficient for 50% of the cases (with no or little relative slowdown), but optimization can further improve performance for the other 50%, especially the ones with a large slowdown. Across all 12870 tests, the arithmetic average is 0.08 for full-trace symbiosis, 0.12 for ABF, and 0.24 for DI. The difference between symbiotic optimal and actual optimal is 0.08 for 8 programs or just 0.01 per program. ABF sampling is per program 0.5% worse than full trace and 1.5% worse than optimal.

The figure shows the relative slowdown of the median solution (out of 35 solutions) for each of the 12870 tests. It indicates the expected performance from a random choice. The average relative slowdown is 0.55. DI reduces this slowdown gap by 56%, ABF sampling 78%, and full-trace symbiosis 85%.

The median distribution shows that the difference between the best and the median solutions is less than 0.1 in 50% of cases but increases quickly and dramatically to over 50% in the remaining cases. It means that the first 50% of the groups are not very sensitive to co-run scheduling, but the others are. For the 50% that are sensitive groups, the average relative slowdown is 0.14 for full-trace symbiosis, a slight increase from 0.08. ABF goes from 0.12 to 0.19. DI performs significantly worse, from 0.24 to 0.42. It shows that optimization is most beneficial for sensitive tasks.

Worst-case Analysis: There are six tests for which symbiosis has over 0.5 relative slowdown. In all of them, it made the wrong decision by picking the group *bzip2, zeusmp, lbm, omnetpp*. The error occurs because the slowdown is uneven within the group. It makes footprint composition inaccurate.

There are five tests for which DI has 1.8 or higher relative slowdown. In the five, DI picks the group *bizp2, soplex, lbm, sphinx3*, and their individual slowdowns are: 1.97, 2.4, 1.63, 2.22. They are among the most cache-sensitive programs but all have a low solo-run miss rate. However, the miss rate increases dramatically when they co-run with others. Since DI assumes the same miss rate in co-run as in solo run, it cannot anticipate the change.

Evenness in Co-run Slowdown: We define $uneveness(G) = \sum_i |slowdown(g_i) - average(G)|/average(G)/|G|$, where G is a co-run group and $average(G) = \sum_i slowdown(g_i)/|G|$ is its average slowdown. When everyone has the group average, unevenness is 0. Otherwise, unevenness gives the average relative deviation by each program from the group average.

Figure 6 shows a histogram of unevenness values from all 1820 cases of 4-program co-runs. 73.7% of them have 10% or less deviation from the group average. 92% have 20% or less. The most uneven are 9 cases (0.5%) whose deviation is between 30% and 35%. One of them is the group, *bzip2, zeusmp, lbm, omnetpp*. Their co-run slowdowns are are 1.71, 1.40, 1.29, and 2.11. The group average is 1.62, and the average deviation, i.e. unevenness, is 17%.

V. Related Work

This section reviews mainly techniques of program symbiosis and locality sampling.

a) The Footprint Theory: Optimization must be built on theory, which in this case is the higher-order theory of locality (HOTL) [33]. As a metric, footprint quantifies the active data usage in all timescales. The HOTL theory shows that as the window size increases, the footprint is not only monotone [30] but also concave [33]. Hence it is strictly increasing unless it reaches a plateau. If it reaches a plateau, it will stay flat. The concavity is useful in two ways. First, the inverse function is well defined, which is the data residence time in cache [33]. Second, the footprint can be used to compute other locality metrics including the miss ratio curve

[1] Instead of running $35 \times 12870 = 450450$ tests, we can simply compute all the results by testing all 4-program co-runs, which means 1820 tests for 16 programs.

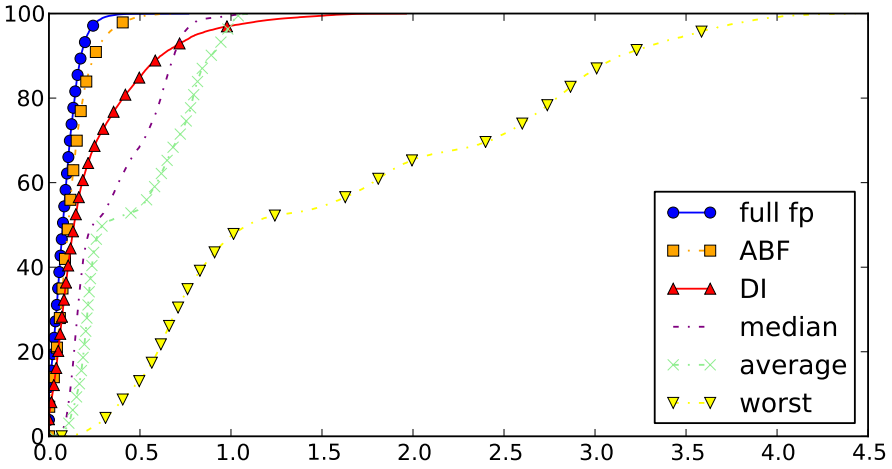

Fig. 5: Distribution of relative slowdowns (compared to best schedule) by optimization using the full trace and the ABF sampling technique, compared with balancing by DI and the median, average and worst of all schedules.

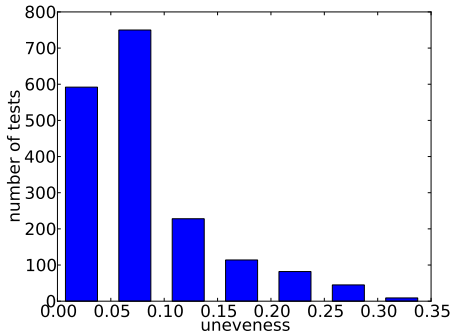

Fig. 6: Unevenness of 1820 co-run slowdowns, measured by the deviation from the average slowdown

and the reuse distance [33]. In HOTL conversion, the concavity of the footprint ensures the monotonicity of the miss ratio. The HOTL conversion connects the two classic theories of locality: the working set theory Denning et al. for primary memory [9], [10] and the theory of stack algorithms by Mattson et al. for cache memory [18]. These and related theories are recently surveyed in [11]. This paper extends the theory with the notion of common logical time. Furthermore, it defines a linear model between locality and performance. These extensions are necessary for symbiotic optimization.

In this study, we have re-implemented the footprint analysis, and the results confirmed that the footprint theory is largely accurate (Figure 4). Through another independent implementation, Wires et al. recently showed that the footprint analysis can be used for disk access traces to predict the server cache performance with a high accuracy [28].

b) Program Symbiosis in Shared Cache: Zhuravlev et al. developed distributed intensity (*DI*) scheduling, which equalizes the sum of solo-run miss rates in each group [37]. Many other techniques address the problem of contention in a shared multicore processor through data-driven analysis, e.g. machine learning [8]. They do not aim to optimize co-run performance. Jiang et al. showed that optimal group scheduling is NP-complete [14]. They gave an exact solution based on integer programming. Such a solution requires knowing the co-run performance beforehand. Xiang et al. composed cache sharing performance using reuse distance and footprint and showed the benefit of cache-conscious task regrouping in two tests [32]. The relation between miss ratio and performance was qualitative rather than quantitative. Mars et al. developed Bubble-up [17] to improve QoS in addition to throughput. The model is based on execution time. It requires a dedicated environment for training, and new training is needed for each machine. Our model is based entirely on footprint. It is machine independent and does not require solo testing.

c) Footprint Sampling: Xiang et al. gave the first technique for footprint sampling [33]. It sets the sample length and frequency as follows. Once a sample is started, the sampling continues until the analysis has seen the amount of data equal to the cache size. This length is the cache lifetime, which is the resident time of an accessed cache block, i.e. between the last time of access and the time of its eviction. Xiang et al. set the hibernation interval to be 10 seconds. The sampling time ranges between 0% and 80% of the original run time, with an average of 19%. Although 18% is hidden by shadow profiling, the interference between the sampling task and the original program can slow down a program by as much as 2.1%. Xiang et al. used sampling to predict solo-run miss ratios. ABF sampling has four differences. The first is approximate prediction. As a result, the sample length is bounded for a given cache size, while the lifetime in Xiang et al. is unbounded, e.g. when the working set of a program is smaller than cache. Second, the total cost is also bounded in ABF sampling (by the hi/bi ratio) but not in Xiang et al. Third, ABF sampling does not collect a sample unless a program has multiple phases, while Xiang et al. always takes a sample. Finally, ABF sampling is used for symbiotic optimization, while Xiang et al. evaluated only miss ratio prediction.

d) Reuse Distance Sampling: Zhong and Chang [35] adopted the approach of bursty sampling [1], [4], [7] to measure reuse distance. An execution is divided into occasional

sampling "bursts" separated by long hibernation periods. The scale-tree algorithm of reuse distance analysis [36] is adapted to use a single node for a hibernation period. They found that sampling was 99% accurate and reduced the measurement overhead by as much as 34 times and on average 7.5 times.

In multicore reuse distance, Schuff et al. modeled locality in multi-threaded code for both private and shared cache [21]. Wu et al. called them private and concurrent reuse distances (PRD/CRD) [29]. Schuff et al. combined the sampling technique of Zhong and Chang with parallel analysis to reduce the measurement overhead to the level of the fastest single-threaded analysis [21].

Reuse distance sampling causes a program to slowdown by at least integer factor, while the cost of ABF sampling is mostly less than 1%. There are two main reasons for the difference. Asymptotically, reuse distance takes more than linear time to measure but footprint takes linear time. Second, the hibernation period in reuse distance sampling is still instrumented and its memory accesses analyzed, but the hibernation period in ABF sampling has zero overhead.

e) Address Sampling: Using the virtual memory paging support, StatCache collects the access trace to sampled addresses to estimate the cache miss ratio [2]. As part of the IBM framework for continuous program optimization (CPO), Cascaval et al. sampled TLB misses to approximate reuse distance [6]. IBM PowerPC has hardware support so a program can track accesses to specific memory locations. They studied the relation between the sampling rate and the accuracy. They found that marking every fiftieth instruction gathers about every thousandth address. The measure reuse distances are treated as a probability density function. The accuracy is defined by Hellinger Affinity Kernel (HAK), which gives the probability that two density functions are the same. Similar hardware support of address sampling is used by Tam et al. to estimate the miss ratio curve [26] and by the HPCS tool for locality optimization, e.g. array regrouping [16]. These techniques are fast but require hardware or OS support. In addition, the metric measured, especially reuse distance, is not composable, so it cannot optimize shared cache symbiosis [11].

f) Time Sampling: Beyls and D'Hollander developed efficient sampling in the SLO tool for program tuning [3]. A modified GCC compiler is used to instrument every reference to arrays and heap data. To uniformly select samples, it skips every k accesses before taking the next address as a sample. To track only reuses, it keeps a sparse vector 200MB indexed by lower-order bits. When being sampled, the index of the chosen address is inserted into the vector. A full check is called whenever the same index is accessed. The overhead comes from two sources. The first is when a full check is called, but it is infrequent. The algorithm uses reservoir sampling which means that the frequency in theory is constant regardless of the length of the execution. The second is address and counter checking, which are two conditional statements for each memory access. Using sampling, they reduced the analysis overhead from 1000-fold slow-down to only a factor of 5 and the space overhead to within 250MB extra memory [3]. Using the tool, they were able to double the speed of five already hand optimized SPEC2000 benchmarks. The SLO tool measures reuse time which is not a direct measure of locality or cache usage.

VI. Summary

In this work, we have defined common logical time and co-run miss ratio based on the common logical time. We have validated the linearity assumption that co-run performance correlates linearly with the logical miss ratio. It enables for the first time to minimize co-run slowdown without co-run testing. In addition, we have developed ABF sampling to bound the analysis error and the total cost. Experimental evaluation shows ABF sampling takes less than 0.1 second per program, and ABF symbiosis is on average 0.12 relative slowdown from optimal co-run (for 8 programs). We conclude that ABF sampling and miss-ratio minimization gives a cost-effective solution for optimal program symbiosis in shared cache.

References

[1] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 168–179, Snowbird, Utah, June 2001.

[2] E. Berg and E. Hagersten. StatCache: A probabilistic approach to efficient and accurate data locality analysis. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, pages 20–27, 2004.

[3] K. Beyls and E. H. D'Hollander. Discovery of locality-improving refactoring by reuse path analysis. In *Proceedings of High Performance Computing and Communications. Springer. Lecture Notes in Computer Science*, volume 4208, pages 220–229, 2006.

[4] M. D. Bond, K. E. Coons, and K. S. McKinley. PACER: proportional detection of data races. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 255–268, 2010.

[5] J. F. Cantin and M. D. Hill. Cache performance for SPEC CPU2000 benchmarks. *http://www.cs.wisc.edu/multifacet/misc/spec2000cache-data*.

[6] 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*, pages 339–349, 2005.

[7] T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–209, 2002.

[8] C. Delimitrou and C. Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 77–88, 2013.

[9] P. J. Denning. The working set model for program behaviour. *Communications of the ACM*, 11(5):323–333, 1968.

[10] P. J. Denning and S. C. Schwartz. Properties of the working set model. *Communications of the ACM*, 15(3):191–198, 1972.

[11] C. Ding, X. Xiang, B. Bao, H. Luo, Y. Luo, and X. Wang. Performance metrics and models for shared cache. *J. Comput. Sci. Technol.*, 29(4):692–712, 2014.

[12] 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*, pages 220–229, 2008.

[13] Y. Jiang, K. Tian, and X. Shen. Combining locality analysis with online proactive job co-scheduling in chip multiprocessors. In *Proceedings of the International Conference on High Performance Embedded Architectures and Compilers*, pages 201–215, 2010.

[14] Y. Jiang, K. Tian, X. Shen, J. Zhang, J. Chen, and R. Tripathi. The complexity of optimal job co-scheduling on chip multiprocessors and heuristics-based solutions. *IEEE Transactions on Parallel and Distributed Systems*, 22(7):1192–1205, 2011.

[15] L. Liu, Y. Li, Z. Cui, Y. Bao, M. Chen, and C. Wu. Going vertical in memory management: Handling multiplicity by multi-policy. In *Proceedings of the International Symposium on Computer Architecture*, pages 169–180, 2014.

[16] X. Liu, K. Sharma, and J. M. Mellor-Crummey. ArrayTool: a lightweight profiler to guide array regrouping. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, pages 405–416, 2014.

[17] J. Mars, L. Tang, K. Skadron, M. L. Soffa, and R. Hundt. Increasing utilization in modern warehouse-scale computers using bubble-up. *IEEE Micro*, 32(3):88–99, 2012.

[18] R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM System Journal*, 9(2):78–117, 1970.

[19] T. Moseley, A. Shye, V. J. Reddi, D. Grunwald, and R. Peri. Shadow profiling: Hiding instrumentation costs with parallelism. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 198–208, 2007.

[20] A. Sandberg, A. Sembrant, E. Hagersten, and D. Black-Schaffer. Modeling performance variation due to cache sharing. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, pages 155–166, 2013.

[21] 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, pages 53–64, 2010.

[22] R. Sen and D. A. Wood. Reuse-based online models for caches. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, pages 279–292, 2013.

[23] A. J. Smith. On the effectiveness of set associative page mapping and its applications in main memory management. In *Proceedings of the International Conference on Software Engineering*, 1976.

[24] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 234–244, 2000.

[25] X.-H. Sun and D. Wang. APC: a performance metric of memory systems. *SIGMETRICS Performance Evaluation Review*, 40(2):125–130, 2012.

[26] 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*, pages 121–132, 2009.

[27] S. Wallace and K. Hazelwood. Superpin: Parallelizing dynamic instrumentation for real-time performance. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 209–220, 2007.

[28] J. Wires, S. Ingram, Z. Drudi, N. J. Harvey, A. Warfield, and C. Data. Characterizing storage workloads with counter stacks. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, pages 335–349. USENIX Association, 2014.

[29] M.-J. Wu, M. Zhao, and D. Yeung. Studying multicore processor scaling via reuse distance analysis. In *Proceedings of the International Symposium on Computer Architecture*, pages 499–510, 2013.

[30] 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*, pages 91–102, 2011.

[31] 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*, pages 350–360, 2011.

[32] X. Xiang, B. Bao, C. Ding, and K. Shen. Cache conscious task regrouping on multicore processors. In *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid*, pages 603–611, 2012.

[33] X. Xiang, C. Ding, H. Luo, and B. Bao. HOTL: a higher order theory of locality. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 343–356, 2013.

[34] X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the EuroSys Conference*, pages 89–102, 2009.

[35] Y. Zhong and W. Chang. Sampling-based program locality approximation. In *Proceedings of the International Symposium on Memory Management*, pages 91–100, 2008.

[36] Y. Zhong, X. Shen, and C. Ding. Program locality analysis using reuse distance. *ACM Transactions on Programming Languages and Systems*, 31(6):1–39, Aug. 2009.

[37] 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*, pages 129–142, 2010.