Optimal Symbiosis and Fair Scheduling in Shared Cache

Xiameng Hu, *Student Member, IEEE,* Xiaolin Wang, Yechen Li, *Student Member, IEEE,* Yingwei Luo, Chen Ding, *Member, IEEE,* Zhenlin Wang

Abstract—On multi-core processors, applications are run sharing the cache. This paper presents optimization theory to co-locate applications to minimize cache interference and maximize performance. The theory precisely specifies MRC-based composition, optimization, and correctness conditions. The paper also presents a new technique called *footprint symbiosis* to obtain the best shared cache performance under fair CPU allocation as well as a new sampling technique which reduces the cost of locality analysis. 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. In an exhaustive evaluation with 12870 tests, the best prior work improves co-run performance by 56% on average. The new optimization improves it by another 29%. Without single co-run test, footprint symbiosis is able to choose co-run choices that are just 8% slower than the best co-run solutions found with exhaustive testing.

Index Terms—locality, working set, cache, performance, optimization

1 Introduction

As multi-core processors become commonplace and cloud computing gains acceptance, applications are increasingly run in a shared environment. Many techniques have addressed the problem of job co-location, including *symbiotic scheduling* [25], co-scheduling [12], contention-aware scheduling [13], [38], task placement [18], and cache-conscious task regrouping [33]. Most techniques use testing and heuristics. Here we use the terminology of Snavely and Tullsen and solve symbiotic scheduling as an optimization problem.

Optimization is difficult given the complexity and dynamics of cache sharing. The traditional metric, the cache miss rate, is measured under fixed cache size. 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.

This paper develops *optimal footprint symbiosis*. It uses the 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.

- *X. Hu, X. Wang, Y. Li and Y. Luo are with Peking University, Beijing, China. Corresponding author: Y. Luo. E-mail: {hxm, wxl, lccycc, lyw}@pku.edu.cn*
- *C. Ding is with University of Rochester, Rochester, NY. E-mail: cding@cs.rochester.edu*
- *Z. Wang is with Michigan Technological University, Houghton, MI. E-mail: zlwang@mtu.edu*

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 four main contributions:

1) **Theory:** A theory of optimal footprint symbiosis to combine non-linear locality composition with linear performance optimization.
2) **Algorithm:** Dynamic programming algorithms to find the optimal co-run schedule that enables optimized sharing of cache and equal use of CPU.
3) **Technique:** Adaptive bursty footprint sampling to enable dynamic co-run optimization with negligible analysis overhead.
4) **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 [24], and as Sen and Wood showed, reuse distance can be used to model the performance of non-LRU policies [23]. Next we introduce the theory of footprint and its relation with reuse distance.

2 Optimal Symbiotic Grouping

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.

2.1 Background: Footprint Theory

Given a window, the *working set size (WSS)* is the amount of data accessed in this 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 equal 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, $t \in [1..n]$.

Figure 1 shows 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.

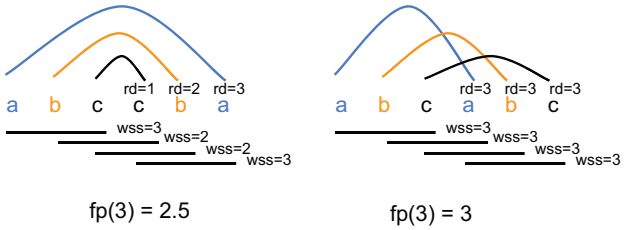


Fig. 1: Reuse distance (rd), working set size (wss) and footprint (fp) of length-3 windows in stack and stream accesses

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 [34]:

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

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 at and only at the next cache miss, when the working set size equals the cache size. For fully-associative LRU cache, the working set fills the entire cache. The time it takes to incur the next miss is the average miss interval (inter-miss time) [34].

Statistically, footprint is the average working set size. The preceding formula equates the growth of the average working set size, $fp(x+1) - fp(x)$, with the average growth of the working set size, i.e. the miss ratio $mr(c)$.

Mathematically, the conversion is well defined. Xiang et al. first showed that the footprint as a function is monotone [32]. Then they proved that it is actually concave [34]. As its "derivative", the miss-ratio function is non-negative and monotone.

2.2 Common Logical Time Miss Ratio

The parameter x in footprint $fp(x)$ represents a timescale. 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. We assume the slowdown of each program in a co-run group is equal. Their relative access ratio in co-run environment follows their solo-run access rates. The *common logical time* (or *common time*) is advanced by accesses from all programs. The footprint is converted from individual time (fp_{it}) into common time (fp_{ct}) as follows:

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

After conversion to common logical time, 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.

2.3 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*. Because the logical co-run miss ratio is composed based on equal slowdown assumption of footprint conversion (Section 2.2). The linearity assumption also clarifies *the slowdown of a co-run group is linear to its logical co-run miss ratio*.

Let g_i be a program in group G, $mr(G, c)$ represents the group co-run miss ratio in shared cache of size c. $t_{so}(g_i, c)$ is the solo-run time of the program with dedicated cache and $t_{co}(g_i, G, c)$ is the co-run time of the program. we define the slowdown of program g_i in the co-run as:

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

and we define the slowdown of group G as:

$$slowdown(G, c) = \sum_i slowdown(g_i, G, c) = |G| + \alpha \times mr(G, c)$$

where α is a constant coefficient that relates co-run miss ratio to co-run group slowdown, and $|G|$ is the size of the group.

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 [26].

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 be linear.

2.4 Optimized Symbiotic Grouping

With the performance ranking model, we now consider program scheduling on multi-core machine. For K programs on P cores, we define a basic symbiotic grouping divides K programs into P-program co-run groups that minimize their co-run slowdown. In this basic problem, we assume P divides K. Symbiosis is communal, not individual. It is sufficient for a program to join a group with the lowest cache contention. However, we must run every task in the program set, so the problem is to run all tasks with the least slowdown.

Given a set of programs $G = \{g_1, g_2...\}$. Let $fp(g_i, x)$ be the common-time footprint. $s = \{G_j\}$ represents a valid grouping, where every program is assigned to a co-run group G_j sharing the same cache. As will become clear in Section 2.6, it is useful to define a metric for a schedule called the *aggregate miss ratio*. It is simply the sum of all co-run group miss ratios of schedule s:

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

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 the metric of merit for comparing the performance between different schedules.

Among all possible groupings, a *symbiotic* grouping is the one that has the lowest aggregate miss ratio. This is the goal of symbiotic optimization.

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 as shown by Kim et al. [39]. This scenario, however, is not consistent with the linearity assumption. However, our evaluation shows individual slowdown differences are insignificant in most groups (discussed in Section 5.4.3). Besides, the intra-group non-uniformity does not affect the efficiency of logical co-run miss ratio as a performance ranking factor (discussed in Section 5.2).

For symbiotic grouping, we use a basic dynamic programming (basic DP) algorithm to find the best P-program groups. The process is to build up solution gradually until a complete grouping is found. Each intermediate step is an optimal partial schedule of the complete schedule. The dynamic programming (DP) formulation is recursive. The intermediate problem is to schedule nP programs, $1 \leq n \leq K/P$. Let A be a set of nP programs. The lowest cost of the set is represented by function F(A). Assuming that the cost of each P-program co-run group G is known as cost(G), e.g. the co-run miss ratio, we specify the lowest cost for A recursively from the lowest cost for all its subsets with $(n-1)P$ programs:

$$\mathrm{F}(A) = \min_{G \subseteq A} \{\mathrm{F}(A - G) + \mathrm{cost}(G)\}$$

Let's consider an example where $K = 6, P = 2$. We want to divide six programs $A = \{g0, g1, \ldots g5\}$ into three pairs that with the lowest cost. There are $\binom{6}{2} = 15$ possible pairs. For each co-run pair G, the right-hand side of the equation computes the cost when G is part of a schedule, which is the cost of G plus the lowest cost for the remaining 4 programs. The 4-program solution is obtained by recursion. In dynamic programming, we record the cost of partial solution to prevent recomputing. Once the lowest cost of the whole set is found, we construct the symbiotic groups by backtracking.

In implementation, we need to enumerate all nP-program groups, $1 \leq n \leq K/P$. We represent a K-element set using a K-bit vector. We call it a *state vector*. When $K \leq 32$, the state vector can be implemented by a 32-bit integer. We enumerate all program sets by traversing the range $[0 \ldots (2^K - 1)]$ and picking out the valid sets, i.e. the number of bits should be a multiple of P. Modern processors have a special instruction to count bits, e.g. `PopCnt` on x86. In our implementation, the validity check takes just three instructions.

2.5 The Performance Ranking Model

The goal of symbiosis is to minimize the contention between programs in shared cache. While the objective is absolute performance, the model is relative: 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 [11]. 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.

2.6 The Case of Multi-threaded Programs

The performance ranking model is not limited to sequential programs. The same symbiosis can be used in the cases where multi-threaded programs are in a co-run group. When a multi-threaded program is sharing LLC with other programs, the data behavior observed in the LLC for all threads become one sequence and thus can be regarded as a sequential flow. Modeling the footprint of a multi-threaded program as a whole is the same as modeling it as a sequential program. Because symbiosis with the existence of multi-threaded programs can be converted to sequential programs, for simplicity, we will only discuss the scheduling of sequential programs in the rest of the paper.

3 Symbiotic Fair Scheduling

With the performance ranking model and symbiotic grouping, we can schedule tasks in a more cache-friendly manner. In this section, we propose a general multi-core scheduler based on the grouping technique discussed in Section 2. We have solved the problem of dividing K programs into P-member symbiotic groups in Section 2. We consider the general problem of fair scheduling in a batch processing system where programs start and finish dynamically while maintain equal CPU time usage. We call this problem *fair scheduling*. Although there are many different ways to achieve fairness, such as fair slowdown proposed by Kim et al. [39]. Choosing CPU fairness as our schedule target is motivated by the Completely Fair Scheduler (CFS) of Linux [15]. Its task picking logic is based on the virtual runtime of each task, which is the actual runtime normalized to the total number of running tasks. CFS always tries to run the task with the least virtual runtime (i.e., the task which has executed least so far), so it balances the expected CPU time each task should get and enable fairness in the task group. Compared to the regular task scheduling policy, SCHED_NORMAL, the batch scheduling policy, SCHED_BATCH, of CFS allows tasks to run in a longer time slice to make better use of caches but at the cost of interactivity. This design is well suited for batch jobs that pay attention to throughput and performance. Our fair scheduling technique adopts the same CPU time fairness guarantee with symbiotic grouping.

3.1 Fair Scheduling

With the basic symbiotic grouping, we can enable the fair scheduling in a batch processing system, which executes a set of G ($|G| = K$) sequential jobs on a processor with P cores sharing the last-level cache (LLC). A fair scheduler has two goals. First, it makes full use of the hardware by keeping all cores running whenever $K \geq P$. Second, it gives the same amount of CPU time to all the programs that are being scheduled. Although a realistic scheduler may consider many other factors, our focus is the shared cache. We limit our concerns to only two factors: full utilization and fairness.

We define a fair schedule as a sequence of co-run groups $\{s_1, s_2, s_3, ...s_L\}$, where each s_i is a subset of G with the size $|s_i| = P$ and the length $L = K/gcd(K, P)$ (gcd is the greatest common divisor). The scheduler cyclically runs each group for T seconds until moving to the next group. When one of the jobs finishes, the job set G is updated, and a new schedule created.

TABLE 1: Number of possible schedules for different K and P

K	P = 3	P = 4	P = 6
6	20	90	1
9	1680	3.150×10^{14}	1680
12	369600	34650	924
18	1.372×10^{11}	$> 10^{15}$	17153136

Consider a group of 4 SPEC CPU2006 programs, G = {lbm, mcf, namd, dealII} on $P = 2$ cores, not all possible pairings are symbiotic. For example, *lbm, mcf* use cache intensely, and *mcf* is sensitive to cache interference. It is not wise to co-run *mcf* with *lbm*. In experiments, the pairing { *[lbm, mcf], [namd, dealII]* } takes 10% more total CPU time than {*[lbm, dealII], [mcf, namd]* }. A symbiotic scheduler starts the execution with the best symbiotic grouping (Section 2.4). When one of them finishes execution, the scheduler has 3 programs left to run on 2 cores. To be fair, it create 3 groups of 2 programs. Each of these 3 programs only appears in 2 different groups. It means each program will run 2 times in every schedule cycle. This guarantees a fair usage of the CPU.

Now we are ready to state the condition for fairness: if we count the cumulative running time of each program, the difference between them can be at most one scheduling quantum (T). In a fair schedule, every program appears the same number of times. More precisely, the schedule includes $L = K/gcd(K, P)$ co-run groups, and each program appears in $U = P/gcd(K, P)$ of them. To ensure fairness, we require that the same program is not run again unless all other programs have been run, so the maximal "unfairness" in CPU time is T.

In some cases such as $K = 3, P = 2$, there is only one fair schedule. In other cases, symbiotic grouping can be used to maximize total co-run performance. Previously in Section 2, we use a K-bit vector to enumerate schedule groups, each element in the state vector is a binary bit (0 or 1) because each program only appears in one group. However, when P does not divide K, each program appears $U = P/gcd(K, P)$ times in the schedule. Here we set each element in the state vector to be an integer with range $[0 \dots U]$.

3.2 Symbiotic Scheduling

If P does not divide K, basic DP is also able to give a fair schedule of optimal performance. The sub-solution in the DP formulation now represents the number of times that every program already appears. It is a state vector with K elements ranging from 0 to $U = P/gcd(K, P)$. However, When P, K are large, the basic DP may be too expensive for online scheduling because its huge solution space. Table 1 shows the number of possible schedules for a set of K, P values. When P does not divide K, the number of possible schedules quickly exceeds 10^{14}. We need to handle this case to implement fair scheduling. Next, we introduce two improvements to reduce the time cost.

3.2.1 Recursive Dynamic Programming

We solve the difficult case (where P does not divide K) faster by recursively solving a series of special cases (where P divides K). We call it *recursive DP* in short. While the previous solution uses base $U + 1$ numbers as state vectors, recursive DP uses only binary bit vectors.

Let $r = K\%P$, r is the smallest value such that P divides $K - r$. We call the general problem a (K, P) problem. Recursive DP transforms the general (K, P) problem into two special problems: $(K - r, P)$ and (K, r).

Take the special case that r divides K. We can apply basic DP (Section 2.4) on this (K, r) problem to find the best schedule, but the cost function of every r-programs should be re-defined. For each group, we take its complement set of $K-r$ programs, run the DP algorithm for $(K - r, P)$ to find the best schedule, and use its cost as the cost of the r-program group. By this definition, the best (K, r) schedule leads to the locally optimal schedule for (K, P) problem. For every group in the best (K, r) schedule, we place the optimal $(K - r, P)$ schedule groups into the local solution of (K, P) problem. A single r-program group now becomes $\frac{K-r}{P}$ groups of P programs each. For example, let $K = 10, P = 4$ and the set of programs be $G = (g_0, g_1, ..., g_9)$. Recursive DP changes the problem to $K = 10, r = 2$ and solves it with basic DP since 2 divides 10. Suppose the solution of $(K = 10, P = 2)$ problem includes the group (g_8, g_9). We can express it as a negation, $(-g_8, -g_9)$, which represents its complement set $(g_0, g_1, ..., g_7)$. Recursive DP replaces$(-g_8, -g_9)$ with the optimal division of $(g_0, g_1, ..., g_7)$ in two 4-program groups. The latter is solved by basic DP since 4 divides 8.

The locally optimal (K, P) schedule is fair. If a program appears in a (K, r) group, it does not appear in its replacement groups. The same frequency of appearance in (K, r) means the same frequency of no-appearance in (K, P) and hence the same frequency of appearance in (K, P).

Now we relax the assumption on r. If r does not divide K, we solve the (K, r) problem through recursion, that is, we split it into two problems: $(K - r_1, r)$ and (K, r_1) where $r_1 = K\%r$. We continue the process $r = K\%P, r_1 = K\%r, \ldots$, until $r_{t+1} = K\%r_t = 0$ and (K, r_t) can be solved by basic DP. This recursion stops after at most $P - 1$ steps. Hence recursive DP always terminates.

The efficiency of recursive DP depends on how basic DP is used to solve $(K - r, P)$. For each r subset, we need to solve $(K-r, P)$, which is costly if we call it for every r subset. Instead, we use DP once for the (K, P) problem. It does not provide a valid solution (when P doesn't divide K), but it provides all solutions of $(K - r, P)$ for all r subsets.

Recursive DP is not global optimal. It splits the general problem (K, P) into (K, r) problems. However, the basic DP of (K, r) problem cannot explore all the possible solutions of (K, P). It has a narrower search space than the whole solution space. Even though (K, r) is optimally solved, it may miss the optimal solution of (K, P). However, searching the complete solution space is time consuming. For example, it takes over 4 days for problem of $(K = 18, P = 6)$, but recursive DP gives a locally optimal solution within a second.

3.2.2 Randomized Dynamic Programming

As we know, the search space grows exponentially with the increasing of K. In the third technique, we sample the job group and then use basic DP to optimize the sample. We call it *randomized dynamic programming* or *randomized DP* for short.

Given K programs, a sample is a random subset of the K programs. Based on sampled subset, we create the shortest fair schedule by basic DP. To make use of basic DP, we create random subsets that contains nP programs. For each subset, we use basic DP to solve (nP, P) problem optimally. To limit the time it takes for basic DP, we set $n = \lfloor min(16, K)/P \rfloor$. For example if $P = 8, K \geq 16$, randomized DP creates random subsets of 16 programs and calls basic DP to schedule each subset to find the optimal solution.

The search space of randomize DP can be as large as needed. The optimality of the solution is monotonically non-decreasing with the growing of search space. Randomize DP is a best-effort algorithm that it gives the best solution under a time constraint. In contrast, recursive DP does not provide a working solution until it runs to a finish. When K is too large for recursive DP, we use randomized DP to limit the time to solution.

4 Adaptive Bursty Footprint Sampling

For shared-cache locality analysis, the original footprint theory is based on full-trace measurement. The time and space overhead may be too high for a task scheduler. Hence, we introduce a footprint sampling technique which is efficient and low-cost.

As we know, the perennial problem of sampling technique is the trade-off between cost and accuracy. The goal is to obtain the best accuracy with the lowest overhead. In footprint sampling, it is achieved by controlling two parameters: the length and frequency of sampling.

In program analysis, a 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 apply bursty sampling to collect sampled access traces to calculate sampled footprint *sfp* and add three novel techniques—limited sensitivity, bounded cost and adaptation. We call it *adaptive bursty footprint sampling* or ABF sampling for short.

The first technique is limited sensitivity. We set the sensitivity threshold h, which is the minimal miss ratio we would predict. The purpose of h is to constrain the length of the sampled access trace, i.e. the burst interval *bi*. Assume that the cache size is c, and the predicted miss ratio is $pmr(c)$. The interval should be long enough to measure the miss ratio equal or higher than h. Here "long enough" means the access trace should be able to fill the cache during the interval *bi*, i.e., $pmr(c) \times bi \geq c$. At the minimal miss ratio h, we have $h \times bi = c$. Hence, the sample length is at most $bi = c/h$.

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 controlling the sampling frequency. We represent the frequency by the ratio of hibernation and 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 checks the actual miss ratio (*amr*) and compares it with the prediction. A new sample is collected if and only if the difference is more than d', which we call the *phase-change threshold*.

The algorithm for ABF sampling is given in Algorithm 1. We use the binary instrumentation Pin tool to sample the access trace online. When bursty sampling is enabled, the procedure is called after each hibernation. The branch at Line 3 tests for phase change 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., so is the use of fork [34]. Such parallel analysis has been pioneered by shadow profiling [20], [28].

Algorithm 1 Adaptive bursty footprint (ABF) sampling

Input: The L2 cache size c; The sensitivity threshold h; The hiberation ratio $r, hi/bi$; The phase-change threshold d'. This procedure is called after each hibernation ($hi = r * bi = r * c/h$ accesses).

Output: The algorithm updates the sampled footprint *sfp* so far. The output is the updated footprint *sfp*.

```
obtain amr(c) using the hardware counter
compute pmr(c) using sfp (Initially pmr(c) = 0)
if |amr(c) − pmr(c)| ≤ d' then
    update sfp using the last sample
else                                        ▷ take a new sample
    pid ← fork()
    if pid = 0 then
        attach Pin and sample for bi (c/h) accesses
        update sfp using the new sample
        exit                                ▷ terminate sampling process
    end if
end if
reset the timer to interrupt after hibernation
```

All sampled access traces are used to calculate *sfp* for miss ratio prediction, i.e. $pmr(c)$ in the algorithm. Because of limited sensitivity, $pmr(c)$ may not be accurate if it is less than h. In this case, its error should be at most h.

In summary, ABF sampling takes 4 parameters. The first two are the cache size c and the sensitive 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 last 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.

5 Evaluation

This section evaluates symbiotic optimization and compares it with alternative solutions.

5.1 Experimental Setup

We use SPEC CPU2006 benchmark suite for program grouping and scheduling. The choice is one of evaluation. Our symbiotic optimization is not limited to compute-intensive programs. In Sections 5.2 and 5.3, we use 16 SPEC CPU2006 programs (arbitrarily chosen): *perlbench, bzip2, mcf, zeusmp, namd, dealII, soplex, povray, hmmer, sjeng, h264ref, tonto, lbm,omnetpp, wrf, sphinx3*. For every program, We only choose 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.

5.2 Linearity Between Miss Ratio and 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 testing optimization.

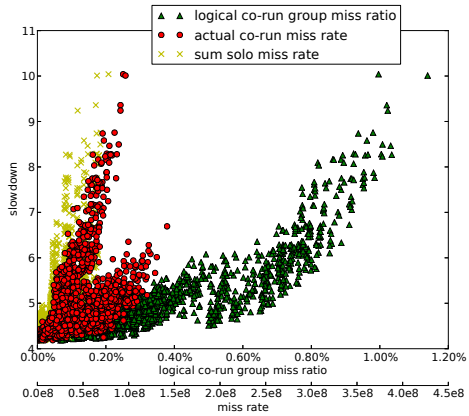


Fig. 2: Correlation with the co-run slowdown by three types of miss metrics: logical co-run miss ratio, actual co-run miss rate, sum of solo-run miss rate

To verify the assumption, We exhaustively test all co-run groups of the 16 SPEC CPU2006 programs.

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

$$\begin{aligned} slowdown(G) &= \sum_{i \in G} slowdown(g_i) \\ &= \sum_{i \in G} \frac{corun(g_i)}{solo(g_i)} \end{aligned} \tag{1}$$

We enumerate all 4-program subsets of the 16 programs, which gives us 1820 co-run groups. Since the test programs in a group have different running times, to measure co-run performance, we run each program repeatedly and measure their slowdown when they are overlapping with each other, a method used in previous work [13], [25], [34], [35]. 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 [21].

Figure 2 plots the performance of 1820 groups (all 4-program subsets of 16 benchmarks), for which the x-axis shows the logical miss ratio of the group (deduced by footprint analysis), 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. By our definition, a slowdown of 4 means no program is affected by co-run, and 16 or larger means that parallel execution of the group is slower than sequential execution.

The logical 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 of them is 0.88. We see two distinct rates in the correlation, divided vertically 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.

The linear relation is not strict: a slightly higher miss ratio does not always mean a greater slowdown. When the miss ratios differ significantly, however, the cache effect becomes dominant. For example, in Figure 2, the best (lowest) slowdown for a group

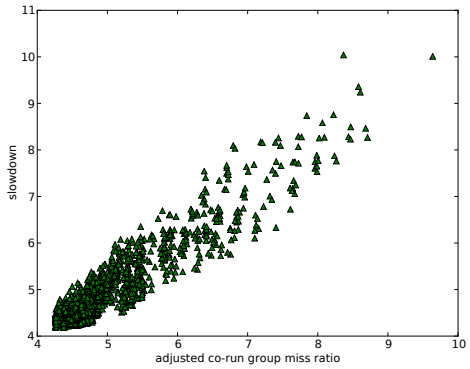


Fig. 3: Correlation between adjusted co-run group miss ratio and co-run slowdown. The correlation coefficient is 0.938.

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. Their correlation coefficients with co-run slowdown are 0.48 and 0.65 respectively. Unlike logical miss ratio, miss-rate correlation shows multiple trends. Unless there is a way to associate the right program with the right trend, we conclude that miss rate is not usable for symbiotic optimization.

The inherent problem of miss rate 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 count all memory transfers (`OFFCORE RESPONSE 0.DATA IN.LOCAL DRAM` used in [34]).

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 2.3 and now has been verified by experiments: although the performance of modern software and hardware is exceedingly complex, the effect of cache sharing is the dominant factor. The linearity assumption here is an observation, and it is the scientific basis for optimal symbiotic scheduling.

5.3 ABF Sampling

Before we present the evaluation of our scheduling technique, we first use the same 16-program set in Section 5.1 to verify the efficiency and accuracy of ABF sampling. In our experiments, we set the phase-change threshold to 1%, the sensitivity threshold to 1%, and hi/bi ratio to 1000. Since the size of shared cache on our test machine is 8MB, there are $8M/64 = 131072$ cache lines. To fill the cache with at least 1% miss ratio, it needs $131072/0.01 = 1.3 * 10^7$ memory accesses, hence we set the length of sampled trace is $bi = 10^7$ and the length of hibernation $hi = 10^{10}$

To compare different parameters, we evaluate under-sampling, where the sample length is 10 times shorter, and over-sampling, where the length is 10 times longer. The parameters of all configurations are shown in Table 2.

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

TABLE 2: Burst/hibernation intervals of 3 sampling methods

Table 3 shows the the sampling cost for the 16 benchmarks, as well as the solo execution time (without sampling) and the number of samples collected by ABF. ABF sampling takes 0.1 seconds or less for all programs except *soplex*, which takes 0.12 seconds. Under-sampling and over-sampling are roughly 10 times faster and slower respectively, since 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 3: Cost (t) and phase count (p) of ABF sampling, compared with under- and over-sampling. Programs are sorted by the solo execution time.

Table 3 also shows the average solo execution time of all programs, which is 257 seconds. For the same benchmark set, Xiang et al. reported the average slowdowns of 38, 153, and 23 times respectively for full-trace simulation (simulating one cache configuration), reuse distance profiling and footprint profiling [34]. A simple extrapolation suggests that for the average running time of around 4 minutes, the average analysis times are 3 hours for simulation, 11 hours for reuse distance and 1.6 hour for footprint analysis.

We compare the miss ratios for all cache sizes up to 8MB (at the increment of one cache block). The collection of miss ratios is called the miss-ratio curve (MRC). Previous techniques had to use much coarser grained MRCs, e.g. cache partitioning [16], [27], when measured using hardware counters or OS support. Footprint and reuse distance MRCs have a much greater resolution and hence support symbiosis at the finest granularity.

We show the comparison for the 16 test programs with one graph each in Figure 4. Each graph shows 6 MRC results. To make it easy to distinguish, points of the same MRC are connected into

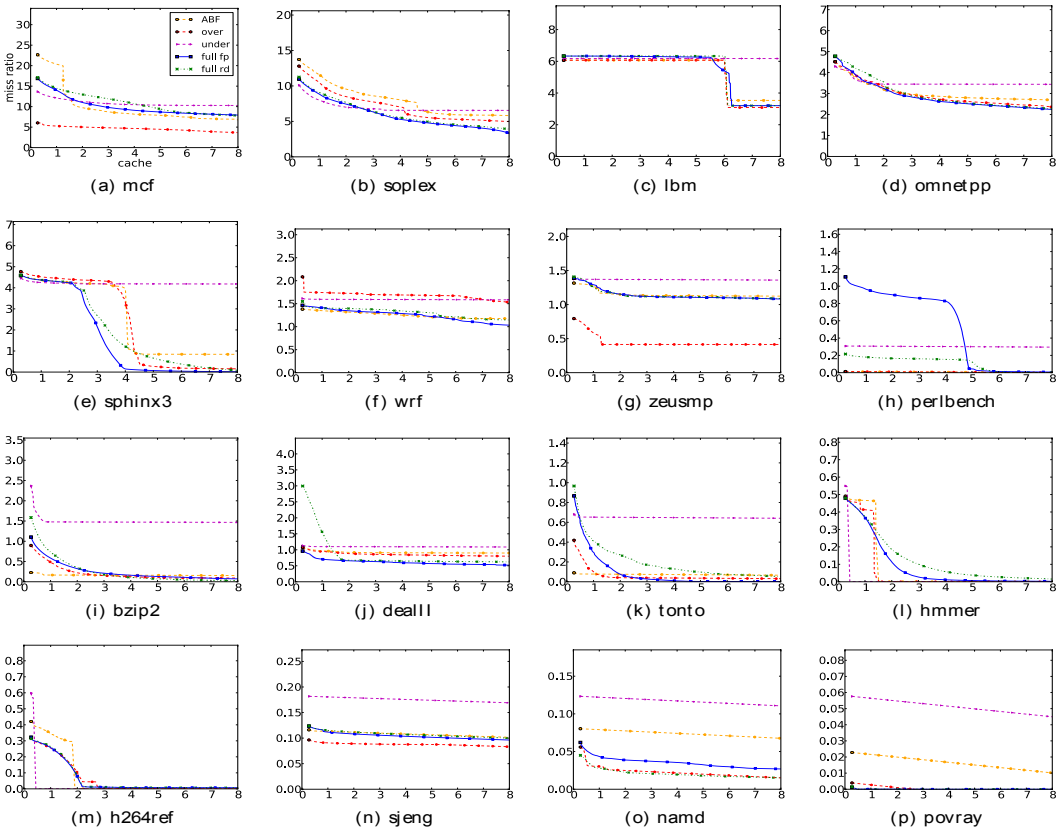


Fig. 4: Miss ratios of the 16 test programs in fully-associative LRU cache of all sizes, at the increment of one cache block, from 256KB to 8MB, including the accurate results by full-trace reuse distance analysis (full rd) and approximate results by full-trace footprint (full fp) and three sampling techniques: ABF, over- and under-sampling. The average overhead per program is 11 hours for reuse distance (full rd), 1.6 hour for footprint (full fp), and 0.09 seconds for ABF (Table 3).

a curve. There are 6 curves in each graph comparing 6 techniques. Two are full-trace analysis, including reuse distance and footprint. As estimated in the last section, their total runtime 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 [34]. The goal of sampling is to approximate the full-trace footprint MRC.

As we can observe, the miss ratios of the first 7 programs are mostly over 1%. ABF sampling produces results close to 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 a gradual decline from 11% to 5%. ABF sampling miss ratios are 3% higher. However, the error is almost constant. The prediction captures the variation almost perfectly.

In *perlbench*, the reuse distance miss ratio is mostly under 0.2%. However, it is a singular case that full-trace footprint mispredicts, showing over 0.8% miss ratios for cache sizes up to 5MB. Interestingly, ABF sampling 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* suggests 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 (relatively) sharp drop of 0.3% 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 predict the (basically) correct miss ratio for larger cache sizes.

The other sampling methods are not as effective as ABF sampling. 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 has a higher precision, i.e. approximation threshold lowered 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 sampling is almost entirely accurate (and finds 17 phases, shown in Table 3).

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

5.4 Symbiotic Optimization

In this section, we evaluate symbiotic grouping and fair scheduling. We have implemented a general fair scheduler based on the combination of recursive DP and randomize DP, which will be tested for footprint symbiosis as well as other co-run optimization techniques for comparison.

5.4.1 DP Algorithms: Cost and Effect

We first evaluate the cost and effect of three DP algorithms. We randomly choose $K = 6, \ldots, 20$ programs from SPEC CPU2006 and use basic DP, recursive DP and randomized DP to find the fair schedule with the minimal average co-run miss ratio on $P = 4, 6$ cores. The two graphs in Figure 5 shows the average logical co-run miss ratios (y-axis) in each schedule of three solutions over different K. For comparison, we also present the best, worst, and average of random solution, we limit the searching time of random solutions to 1 second. The number of processors is $P = 4$ in the left-hand side graph and $P = 6$ in the right-hand side graph. The miss ratio between different Ks is not comparable since it is for different task groups. We connect the points of the same technique for a better (visual) comparison.

As we discussed in Section 3, basic DP is optimal, but after $K = 10$ it becomes too slow. It takes 2.1 seconds for $(K = 10, P = 6)$ problem, over a minute for $(K = 11, P = 6)$ and over 4 days for $(K = 18, P = 6)$. Recursive DP is indistinguishable from basic DP in its optimization result but runs in less than 0.1 seconds for $K \leq 16$ and up to 39 seconds for $K \leq 20$. Randomized DP is set to run for 1 second. It begins to deviate from recursive DP at $K = 17$ in the left-hand graph and $K = 16$ in the right-hand graph. All three DP algorithms are better than the best of 1-second random testing (which examines at least 1 million solution selected at random). The advantage is magnified after $K = 16$.

Hence, in our fair scheduler, we combine recursive DP for $K \leq 16$ for practically optimal results and randomized DP for larger K for best effort results. The cost is less than 0.1 seconds for recursive DP and 1 second for randomized DP.

5.4.2 Group and Scheduling Techniques

We compare our footprint-based symbiosis with four other techniques in two tests. Grouping is to divide 8 programs into two 4-program co-run groups. Scheduling is to execute a set of 20 programs in a fair schedule.

Bounded-Bandwidth Footprint Symbiosis Footprint symbiosis as described in Section 3 finds the best co-run schedule with minimal co-run miss ratio. 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% and then at a much faster rate afterwards. The reason is memory bandwidth, which starts to saturate at that point (37% of the peak memory bandwidth on the test machine). In symbiotic optimization, we want to avoid such group that take much bandwidth. Therefore 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 to 0.6%.

Without the bandwidth ceiling, an optimal schedule may be arbitrarily unbalanced. For example, when dividing 4 programs into two pairs, the solution with minimal total co-run miss ratio may contain the program pair that incurs all the misses and overburden 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 non-linear aggregate effect, imbalance is inevitable in optimization.

Distributed Intensity (DI) Distributed intensity (DI) was developed by Zhuravle et al. [38]. It sorts co-run programs by decreasing solo-run miss rate (misses per million instructions) and assigns them round-robin into each group. The resulting schedule effectively balances the sum of the miss rates in each group. For reproducible results, we use the DI defined miss rate measured in complete solo executions. DI 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.

Multi-threaded Bubble-up Mars et al. used two metrics, pressure and sensitivity, to characterize a program in shared cache [18]. Pressure is how much a program affects the performance of others, and sensitivity is how much the program is affected by the pressure of others. The two metrics are determined by running a program against two probe programs. The bubble program exerts an escalating level of interference. The reporter program measures the performance loss caused at each level of pressure.

The original design was for two-program co-runs [18]. A single bubble program cannot create sufficient pressure on our quad-core system. We implement a multi-threaded bubble (mt bubble) program, which runs the original bubble in $1...P - 1$ threads. Then we test the sensitivity. We measure complete executions of benchmarks to obtain their pressure and sensitivity values, use them to build the mt bubble predictor, and use it in DP to minimize the predicted total slowdown.

The following two are used only in scheduler testing.

SOS SOS creates the fair schedule from 10 random permutations, runs each for 200ms, and then selects the one that has the maximal IPC. Unlike the other methods, whose analysis is all offline, SOS is implemented online with sampling. Since the total running time is in thousands of seconds, the cost of sampling is negligible. SOS uses the maximal IPC and does not need to know the solo time.

Random In fair scheduler testing, we randomly generate 100 permutations of the 20 benchmarks. For each permutation, we rotate to create a fair scheduling. These 100 permutations show the range of performance without optimization.

5.4.3 Symbiotic Grouping

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. As we mentioned earlier, because programs have different execution times, we run each program repeatedly and measure its speed when it is overlapping with other programs.

We simply compute all the results by testing all 4-program co-runs (1820 groups). We use these 4-program groups to evaluate

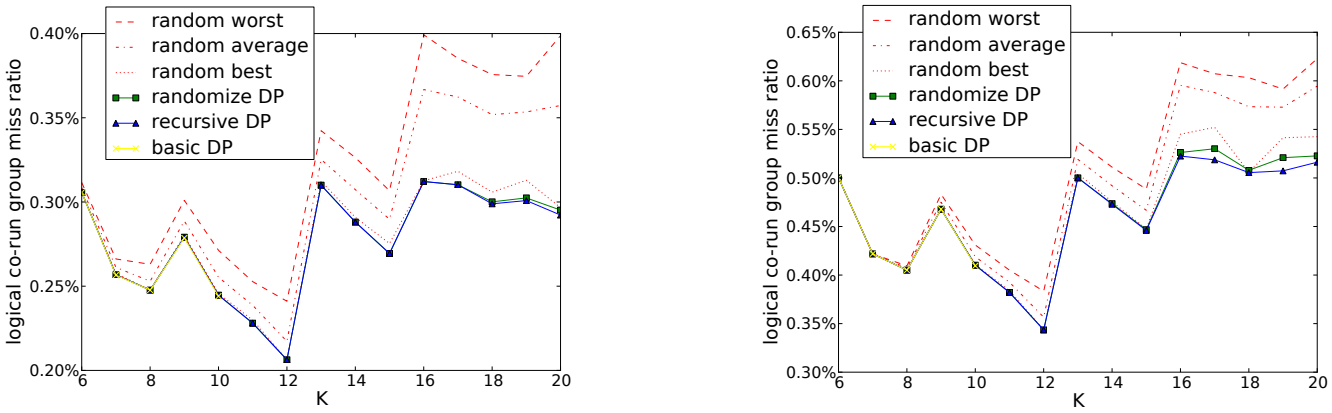


Fig. 5: The miss ratio for $K = 6, \ldots, 20$ and $P = 4$ (Left) and $P = 6$ (Right), produced by DP, recursive DP and randomized DP compared to the best, worst and average of 1-second random testing. DP is at most 2.1 seconds for $K \leq 10$ and recursive DP at most 0.1 seconds for $K \leq 16$. Randomized DP is run for 1 second.

the effect of 8-program symbiotic grouping. Then, we rank all the solutions by their relative slowdown, which is defined as difference of co-run slowdown compared to the best solution. 0 relative slowdown means optimal symbiotic grouping.

Figure 6 shows the (cumulative) distribution of relative slowdowns of different grouping technique 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 footprint 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 second best, equally as good as the full-trace footprint for 90% tests. The deviation for the remaining 10% is not significant.

Unlike the two footprint (full-trace and sampling) techniques, which aim to optimize, DI is a heuristic to seek balance. Among all its solutions, 4.8% is optimal, 38% within 0.1, and 62% within 0.2. mt-bubble is worse than DI, the three percentages are 4%, 27% and 48%. DI is as good as optimization (full-trace footprint and ABF sampling) for half of the tests, which means 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 larger slowdowns. Across all 12870 tests, the arithmetic average relative slowdown is 0.08 for full-trace symbiosis, 0.12 for ABF, 0.24 for DI and 0.44 for mt-bubble. 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 analysis and 1.5% worse than optimal solutions.

The figure also gives the relative slowdown distribution of the median, worst (ranked by relative slowdown) solutions (out of 35 solutions) as well as the average relative slowdown of all solutions of the 12870 tests. The median solution indicates the expected performance of a random solution. The average relative slowdown of median solutions is 0.55. DI reduces the slowdown gap by 56%, while ABF sampling reduces 78%, and full-trace symbiosis reduces 85%.

The median distribution shows that the difference between the optimal 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 other groups are. For the remaining 50%, sensitive groups, the average relative slowdown is 0.14 for full-trace symbiosis, a slight increase from 0.08, and ABF sampling goes from 0.12 to 0.19. DI performs significantly worse, from 0.24 to 0.42, and so does mt-bubble, from 0.44 to 0.79. It shows that optimization is highly beneficial for sensitive tasks.

Worst-case Analysis In our evaluation, we notice that there are six 8-program tests for which symbiosis has over 0.5 relative slowdown. In all of these tests, symbiosis 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 the co-run miss ratio prediction inaccurate.

There are five tests for which DI has 1.8 or higher relative slowdown. In these groups, 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 foresee these cases.

Evenness in Co-run Slowdown Although the performance ranking model is based on even slowdown assumption. The intra-group non-uniform slowdown exits in most cases. To evaluate the degree of uneven slowdown in co-run group, we define the uneveness of group G as:

$$uneveness(G) = \frac{\sum_i |slowdown(g_i) - average(G)|}{average(G) * |G|} \quad (2)$$

where $average(G) = \sum_i slowdown(g_i)/|G|$ is its average slowdown of this group. When everyone's slowdown equals to group average, unevenness is 0. Otherwise, unevenness gives the average relative deviation of each program from the group average.

Figure 7 shows a histogram of unevenness values from all 1820 cases of 4-program co-run groups. As we can observe, 73.7% of them have 10% or less deviation from the group average. 92% of them are below 20%. The most uneven groups are 9 cases (0.5%) whose deviation is between 30% and 35%. One of them is the group *bzip2, zeusmp, lbm, omnetpp*, mentioned previously in the worst case analysis. Their co-run slowdowns are 1.71, 1.40, 1.29,

and 2.11. Their group average slowdown is 1.62, and the average deviation, i.e. unevenness, is 17%. Although our performance ranking model is based on even slowdown assumption. The intra-group non-uniform slowdown truly exits in most groups. But the unevenness in most cases is insignificant. The results in Figure 6 verify the efficiency of the performance of ranking model even though there is certain level of non-uniform slowdown in a group.

5.4.4 Symbiotic Scheduling

In this test, the fair scheduler (Section 3.1) takes a set of 20 programs (randomly chosen from SPEC CPU2006). At each moment, it runs the next K unfinished programs based on the best schedule found by recursive DP or randomize DP. The scheduling quantum T is set to 10 seconds, which is the same order of magnitude of the CFS scheduler under batch scheduling policy (1.5 seconds). Every time a benchmark finishes, the optimal schedule is re-computed. When the remaining tasks are fewer than P, it stops scheduling and waits for the last program to finish and compute the elapsed total wall-clock time.

Fair scheduling gives repeatable running times. For the same input sequence and the same scheduling method, different runs have almost the same total wall-clock time, at most five seconds difference for the totals in thousands of seconds. We test four symbiotic methods: footprint symbiosis, DI, mt-bubble, and SOS.

Figure 8 shows the finish times of different method for ($K = 16, P = 4$). It also shows the results of 100 random scheduling in the sorted order (on x-axis) from fastest, 6284 seconds, to slowest, 6947 seconds (11% difference). There is no randomness in the symbiotic methods. Their times are shown by horizontal lines.

Footprint symbiosis performs the best, which is expected from exhaustive testing results. It takes 6147 seconds, faster than the best heuristic solution DI by 179 seconds (2.9%) and random average by 344 seconds (5.6%) respectively. What's remarkable is that it is better than random best by 137 seconds or 2.2%. The reason for this is that a scheduler test involves multiple scheduling decisions, so footprint symbiosis has more opportunities to optimize while other techniques have a higher risk of choosing a schedule far from the best.

In Section 5.4.3, we have analyzed footprint symbiosis, DI, and mt-bubble. Here we examine SOS. As K becomes larger, the number of possible schedules increases exponentially. *SOS* uses the same number of samples but can still improve performance

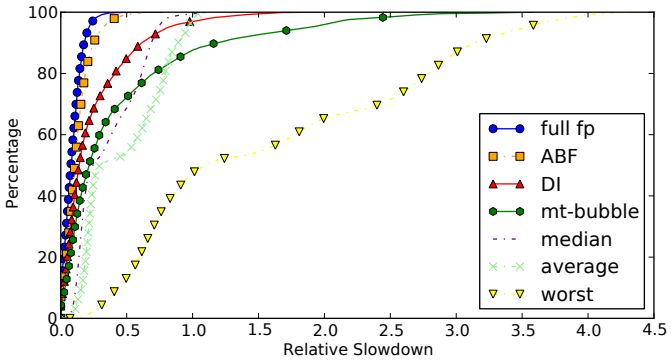


Fig. 6: Distribution of relative slowdowns (compared to best schedule) by optimization using full-trace footprint and ABF, compared with balancing by DI, pressure-sensitivity by mt-bubble, and the median, average and worst of 100 random groupings.

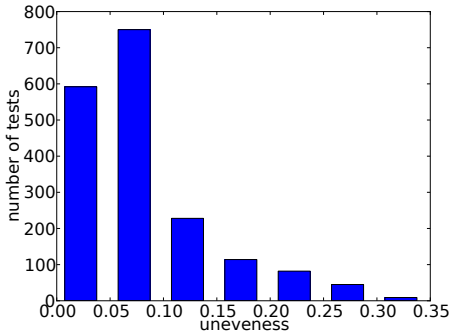


Fig. 7: Unevenness of 1820 co-run slowdowns, measured by the deviation from the average slowdown

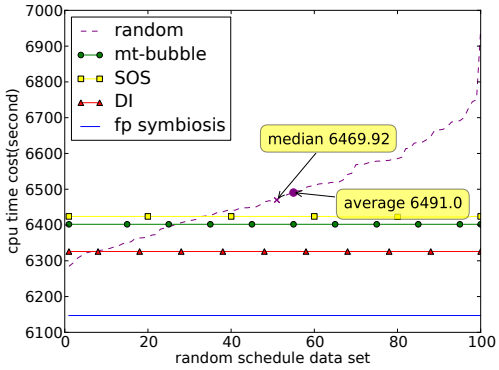


Fig. 8: Finish times of different methods for $K = 16, P = 4$

as K increases from 12 to 20. However, further increasing the number of samples from 10 to 100 does not further improve performance. More sampling incurs extra cost. More importantly, the search space is too large for the additional 90 samples to make a significant difference.

Figure 8 shows footprint symbiosis at its best for this test. At the other K values, footprint symbiosis takes a longer time, shown in Figure 9. The x-axis shows K values from 6 to 20. Just three results, minimal, median, and maximal, of random for $K = 20$ are shown. It shows the performance of a fifth method, cache-conscious task regrouping [33], in the curve marked *CCTR*. Footprint symbiosis is consistently the best and except for $K = 10$, outperforms the best of 100 random schedules, finishing under 6284 seconds, at least 73 seconds faster than the random best. The performance increases in larger K, slower than 6220 seconds when $K \leq 14$ but faster than 6184 seconds when $K \geq 16$. DI and mt-bubble also show a greater benefit for larger K.

As we showed in Figure 6, the majority of programs are not affected significantly by co-run scheduling, but some programs are. While a better scheduler can improve these co-run sensitive programs, the overall improvement is much smaller because it is averaged over all programs. This is shown by both DI and footprint symbiosis. If we compare DI with the median of random in Figures 8 and 9, we see that the average improvement by DI is around 2% compared to the median of random schedules. Although the average improvement is small, the individual improvements are much larger for the portion of programs that are co-run sensitive. As shown in Figure 6, footprint symbiosis makes further improvements, reducing the slowdown (compared to best)

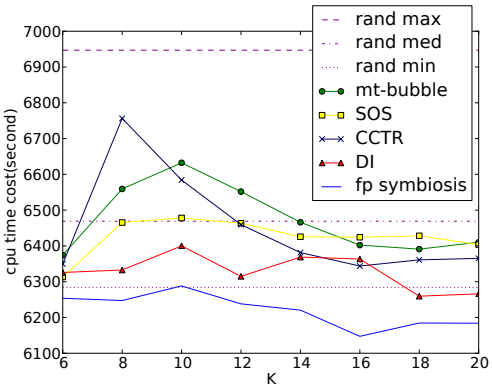


Fig. 9: Finish times of different methods for $K = 6 \ldots 20, P = 4$

by a factor of two or more over DI, for over 30% of program groups. In Figures 8 and 9, we see the effect of these additional improvements on overall performance. We have two benefits from the additional complexity of optimization. In practice, it doubles the overall improvement of the best prior method. In theory, it shows how much further improvement is possible through co-run scheduling, which was an important open question left by prior work. Furthermore, footprint symbiosis is efficient. Table 3 shows that the runtime overhead for each program is less than 0.1 seconds. With randomize DP, the optimal solution searching can be done in 1 second. The runtime overhead of footprint symbiosis is comparable to DI.

6 Related Work

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

The Footprint Theory The footprint theory was built on earlier models of shared cache, Suh et al. [41] for time-sharing systems and Chandra et al. [40] for multicore processors. The footprint was estimated but not measured, a problem that was later solved in the higher-order theory of locality (HOTL) [34]. 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 [31] but also concave [34]. 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 [34]. Second, the footprint can be used to compute other locality metrics including the miss ratio curve and the reuse distance [34]. 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 by Denning et al. for primary memory [9], [10] and the theory of stack algorithms by Mattson et al. for cache memory [19]. 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 confirm 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 [29].

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 [38]. Mars et al. developed Bubble-up [18] to improve QoS in addition to throughput. The model is based on execution time. 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 [33]. The relation between miss ratio and performance was qualitative rather than quantitative. Some techniques, e.g. DI and Bubble-up, require 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 (except for measuring the access rate).

Footprint Sampling Xiang et al. gave the first technique for footprint sampling [34]. 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.

Reuse Distance Sampling Zhong and Chang [36] 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 [37] 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 [22]. Wu et al. called them private and concurrent reuse distances (PRD/CRD) [30]. 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 [22].

Reuse distance sampling causes a program to slow down by at least a 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.

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 measured 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 [27] and by the HPCToolkit for locality optimization, e.g. array regrouping [17]. These techniques are fast but require hardware or OS support. In addition, the metric measured, especially reuse distance, is not composable, so it would require co-run testing to optimize shared cache symbiosis [11].

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 slowdown to only a factor of 5 and the space overhead to within 250MB of 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.

7 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. We have developed the footprint symbiosis technique based on the linearity assumption as well as a fair CPU scheduler for program scheduling on multi-core machine. In addition, we propose ABF sampling to bound the analysis error and the total cost of footprint analysis. Experimental evaluation shows that ABF sampling takes less than 0.1 seconds per program. Through exhaustive and scheduler testing, we show that footprint symbiosis is on average 8% away from optimal co-run (for 8 programs), it can outperform the best possible system co-run (for 20 programs), and it improves the previous techniques in both performance and robustness.

Acknowledgments

The research is supported in part by the National Science Foundation of China (Contract No. 61232008, 61272158, 61328201, 61472008 and 61672053); the 863 Program of China under Grant No.2015AA015305; the National Science Foundation (Contract No. CCF-1629376, CNS-1319617, CSR-1618384, CSR-1422342); IBM CAS Faculty Fellow program; and a grant from Huawei. 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 organization

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] CFS Scheduler, https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt, 15 4 2016.

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

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

[37] 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.

[38] 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.

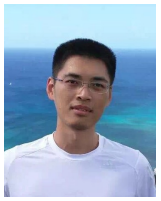
[39] S. Kim, and D. Chandra and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 111–122, 2004.

[40] 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*, pages 340–351, 2005.

[41] G. E. Suh, S. Devadas, and L. Rudolph. Analytical cache models with applications to cache partitioning. In *Proceedings of the International Conference on Supercomputing*, pages 1-12, 2001.

Xiameng Hu received his B.S. degree from Tianjin University in 2013. He is currently working towards the PhD degree in the School of Electronics Engineering and Computer Science at Peking University. His research interests include distributed computing, memory system optimization and data locality theory etc.

Xiaolin Wang received his B.S. and Ph.D. degrees from Peking University in 1996 and 2001 respectively. He is an associate professor in Peking University. His research interests include system software, virtualization technologies and distributed computing, etc.



Yechen Li received his B.S. and M.S. degree from Peking University in 2012 and 2015 respectively. He is now working at Google Inc.

Yingwei Luo received his B.S. degree from Zhejiang University in 1993, and his M.S. and Ph.D. degrees from Peking University in 1996 and 1999 respectively. He is a professor in Peking University. His research interests include system software, virtualization technologies and distributed computing, etc.

Chen Ding holds the rank of Professor of Computer Science at University of Rochester and conducts research in locality theory and optimization and safe program parallelization.

Zhenlin Wang is currently a professor of the Department of Computer Science at Michigan Technological University. His research interests are broadly in the areas of compilers, operating systems and computer architecture with a focus on memory system optimization and system virtualization.