Optimal Cache Partition-Sharing*

***Don't ever take a fence down until you know why it was put up.* – Robert Frost**

Jacob Brock, Chencheng Ye[1], Chen Ding
University of Rochester
Rochester, NY, USA

Yechen Li, Xiaolin Wang and Yingwei Luo
Peking University
Beijing, China

***Abstract*— When a cache is shared by multiple cores, its space may be allocated either by sharing, partitioning, or both. We call the last case *partition-sharing*. This paper studies partition-sharing as a general solution, and presents a theory an technique for optimizing partition-sharing. We present a theory and a technique to optimize partition sharing. The theory shows that the problem of partition-sharing is reducible to the problem of partitioning. The technique uses dynamic programming to optimize partitioning for overall miss ratio, and for two different kinds of fairness.**

Finally, the paper evaluates the effect of optimal cache sharing and compares it with conventional solutions for thousands of 4-program co-run groups, with nearly 180 million different ways to share the cache by each co-run group. Optimal partition-sharing is on average 26% better than free-for-all sharing, and 98% better than equal partitioning. We also demonstrate the trade-off between optimal partitioning and fair partitioning.

I. Introduction

There are two popular options for sharing an individual resource such as cache: partitioning and sharing. In partitioning, each core gets a protected portion of the total cache. In sharing, each core has access to the whole cache, and may evict data belonging to another core. Partitioning provides each program protection against aggressive partners, while sharing prevents resources from going unused when they may be needed[2]. If there are only two programs, they may either share or partition any given resource. However, for three or more programs, partitioning and sharing may both be used. For example, two programs may share a partition, while the third program has its own partition. We propose to call this type of scheme *partition-sharing*. Strict partitioning and free-for-all sharing can be seen as opposite edge cases of partition-sharing. In this paper, we investigate partition-sharing as a general problem and give an optimal solution for both throughput and fairness.

The new solution has two components. First, we reduce the problem of partition-sharing to just partitioning, i.e., the optimal solution for partitioning is at least as good as the optimal solution for partition-sharing. The essence of the reduction is the concept of a *Natural Cache Partition*. Given a set of programs sharing a cache, the natural partition is a cache partition such that each program has the same miss ratio in its natural partition as it has in the shared cache. The performance of any shared cache is then equivalent to the performance of naturally partitioned cache. The general problem of finding the best partition-sharing is thus reduced to finding the best partitioning. We give the formal derivation including the precise conditions for optimality.

The second component of the solution is an optimization algorithm to find the best cache partitioning. The previous solution by Stone et al. was designed to maximize performance (minimal group miss ratio) and required the assumption that the individual miss ratio curve be convex [9]. Our algorithm uses dynamic programming to examine the entire solution space. It generalizes the optimization in two ways. First, the miss ratio curve of individual programs does not have to be convex. In fact, it can be any function. Second, it can optimize for any objective function, for example, fairness and quality of service (QoS) in addition to throughput.

For fairness, we define two types of baseline optimization which increase the group performance if no program would perform worse than it is with a baseline partition. We consider two example baselines: the equal partition and the natural partition (i.e., free-for-all sharing).

The main contributions of the paper are:

- *Cache partition-sharing*, for which the existing problems of cache partitioning and sharing are special cases.
- *Natural cache partition*, which gives a theoretical reduction to make the optimization problem solvable.
- *Optimal partitioning algorithm*, which handles all types of programs and objective functions.

*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; 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 organizations.

[1]Chencheng Ye is a visiting student, funded by the Chinese Scholarship Council. His home institution is Huazhong University of Science and Technology, Wuhan, China.

[2]Parihar et al. presented a counter-based hardware mechanism to provide programs the protection of partitioning, without the risk of unused space [4].

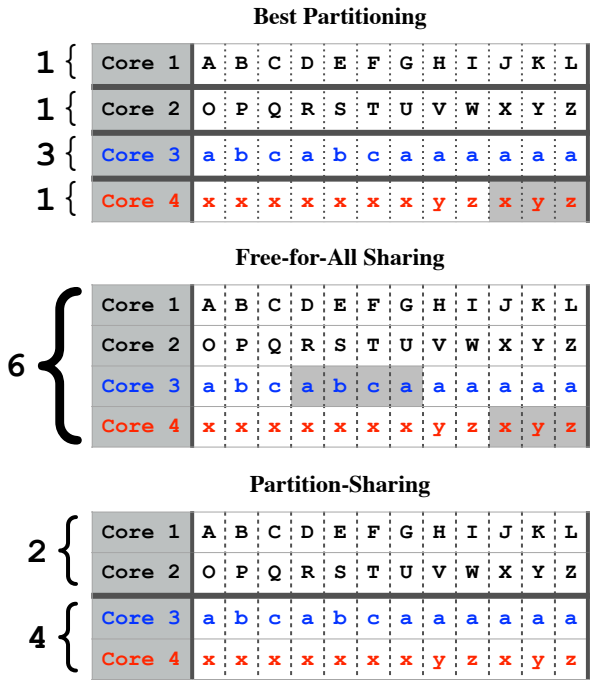

Figure 1: A set of program traces that will benefit from partition sharing. Core 1 and core 2 run streaming programs, so partitioning them off prevents them from polluting the cache. Cores 3 and 4 alternate between having large and small working sets. Sharing a partition allows them each to use more cache when it is needed. Capacity misses are shaded with gray.

- *Evaluation*, which shows the effect of optimization for a set of benchmark programs.

The new solution is useful for a programmer and a computer architect to understand and solve the optimization problem. It fully evaluates the potential of general cache sharing, for both performance and fairness and over existing, narrower solutions.

II. Partition-Sharing Options for Multicore Cache

The general need for partition sharing can be demonstrated with a simple example trace, with 4 cores sharing a cache size of 6. If cores 1 and 2 are running streaming applications, and cores 3 and 4 alternate between large and small working sets (i.e., core 3 needs cache, then core 4 needs cache), cores 1 and 2 should be partitioned to prevent cache pollution, while cores 3 and 4 should share, so that each may use the shared partition while the other does not. The example is illustrated in Figure 1.

In order to understand the complexity of decision making for the partition-sharing, we can divide the problem into 3 sub-problems, illustrated in Figure 2 and described below. In the following descriptions, n_{pr} is the number of programs, n_{pa} is the number of partitions, n_c is the number

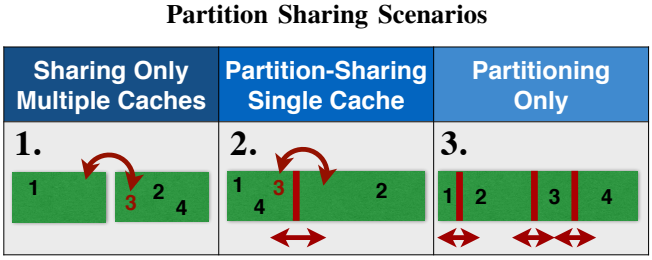

Figure 2: Examples of possible partition sharing scenarios with 4 programs and one or two caches. **1**: Two caches shared by any number of programs. **2**: One cache partitioned for any number of groups of any number of programs (general partition-sharing case). **3**: One cache with dedicated partitions for each program (partitioning-only).

of caches, C is the size of each cache, n_g is the number of groups, and G is the size of each group. The number of possible arrangements (within the constraints) is called the search space size (S). The search space sizes shown below count every possible unique arrangement of programs in caches/partitions.

1. **Sharing, Multiple Caches:** There are multiple caches, but the number of users for each cache may vary. Grouping is still the only variable, but now group sizes are not required to be the same. The problem size here is the number of ways we can separate n_{pr} programs into n_c non-empty groups. This happens to be the well-known *Stirling number of the second kind*:

$$S_1 = \left\{ n_{pr} \atop n_c \right\}. \tag{1}$$

2. **Partition-Sharing, Single Cache:** There is only one cache, which is partitioned. Groups of programs are assigned to partitions. The problem size is the sum of the Stirling numbers multiplied by the number of ways to assign "walls" that partition the cache. That is, for each number of partitions n_{pa}, there are $\left\{ n_{pr} \atop n_{pa} \right\}$ ways to group the programs, and for each of those, there are $\binom{C+n_{pa}-1}{n_{pa}-1}$ ways of assigning cache (balls) to the partitions (bins).

$$S_2 = \sum_{n_{pa}=1}^{n_{pr}} \left\{ n_{pr} \atop n_{pa} \right\} \binom{C + n_{pa} - 1}{n_{pa} - 1} \tag{2}$$

3. **Partitioning Only:** When partitioning alone is used, the number of partitions equals the number of programs. This problem is simply to assign units of cache (balls) to each program (bins). The problem size is

$$S_3 = \binom{C + n_{pr} - 1}{n_{pr} - 1}. \tag{3}$$

As an example, for 4 programs run on an $8MB$ cache that is divisible into $64B$ units, we have $n_{pr} = 4$ and $C = 8MB/64B = 131072$. In this case, $S_2 = 375,368,690,761,743$ and $S_3 = 375,317,149,057,025$. In other words, the solution set of partitioning-only covers 99.99% of the solution set of partition-sharing. This is due to the fact that there are so many more ways to assign cache to 4 partitions than there are ways to assign it to 1, 2, or 3 partitions. Fortunately, we have an algorithm (presented in Section V-B) to find the optimal solution among those 99.99%. We expect the solution in this space to be approach the performance of the optimal partition-sharing solution, particularly for programs without strong phase behavior, and for higher partitioning granularity.

III. The Higher Order Theory of Locality

Relationships between several data locality metrics were developed by Xiang et al. [16] in a theory called the *higher order theory of locality* (HOTL). The metrics gives preliminary definitions, as well as the metrics and their relationships.

Window: A window beginning at the i^{th} element (inclusive) of the trace and continuing for k elements is denoted window (i,k). The windows $(2,6)$ and $(4,2)$ are boxed in Figure 3.

Reuse Pair: A pair of accesses to the same datum, with no other accesses to that datum in-between. Two reuse pairs are highlighted in Figure 3.

Reuse Window: The smallest window containing both members of a reuse pair. Two reuse windows are boxed in Figure 3.

Reuse Time: The *reuse time* for a reuse pair is the length of their reuse window. E.g., for the i^{th} and j^{th} elements of a trace $d_1 ... d_n$,

$$rt(d_i, d_j) = j - i + 1. \tag{4}$$

It is calculated at the second access of the pair, as shown in Figure 3.

The reuse time histogram for a trace can be expressed as a function $freq(rt)$, denoting the number of reuse pairs with reuse time rt.

Footprint: There are two formulations of the footprint function: $WSS(W)$ is the number of distinct data accessed in a trace window W, and $fp(w)$ is the average of $WSS(W)$ for all windows W of length w in a given memory trace. The average footprint function can be approximated from a histogram of reuse times.

$$fp(w) \equiv \frac{1}{n-w+1} \sum_{i=1}^{n-w+1} WSS(i,w) \tag{5}$$

For the rest of the paper, we will only refer to the average footprint function.

Datum	a	a	x	b	b	y	a	a	x	b	b	y
Reuse Dist	-	1	-	-	1	-	4	1	4	4	1	4

Figure 3: An example trace with two reuse windows boxed. Each of the reuse windows decreases every containing window's footprint due to the data redundancy.

Fill Time: The average footprint $fp(c)$ is the average number of distinct data in a window of length c (we use c now because, as will soon be clear, it represents a cache size). The *fill time* is defined as the expected number of accesses it takes to touch a given amount of distinct data, so an equivalent statement is that $ft(fp(c)) = c$. So we have the relationship:

$$ft(c) = fp^{-1}(c). \tag{6}$$

Inter-Miss Time: The fill time for a cache of size $c+1$ is the fill time for a cache of size c plus the average time until the next miss on the size c cache.

$$im(c) = ft(c+1) - ft(c) \tag{7}$$

Miss Ratio: The miss ratio is the reciprocal of the inter-miss time.

$$mr(c) = \frac{1}{im(c)} \tag{8}$$

IV. Miss Ratio Composition Theory

It is necessary to have a theory that can predict the miss ratios of co-run programs using metrics measured only on each program. For a scheduling problem with 20 programs that need to be scheduled on 2 processors sharing a cache, we would like to be able to predict cache performance based on 20 metrics, not 20-choose-2 (and for 4 processors, this becomes 20-choose-4, and so on). In addition to scheduling, another possible application of this would be to monitor performance on-line, and stall individual programs based on the predicted benefit of doing so. For example, if two programs are traversing different 60MB arrays while sharing a 64MB cache, stalling one of them will prevent thrashing, and they may both finish sooner this way.

As with the prior footprint composition method, we begin with the following equation for $fp(w)$, the average footprint of a window of length w, given the access rates ar_1 and ar_2 of each program[3]:

Stretched Footprint: When the memory accesses of two programs interleave, the result is a single trace of memory accesses. Taken as a part of this whole, each program's footprint function is horizontally stretched based on the ratio

[3]The access rate is measured for each program as the length of its memory access trace divided by the number of seconds the program ran for.

of its accesses to the other programs' accesses in any given amount of time (i.e., $\frac{a_i}{a_1+a_1}$). The overall footprint is then the sum of the individual stretched footprints. For two programs, this is

$$\begin{aligned} fp(w, ar_1, ar_2) =& fp_1\left(w * \frac{ar_1}{ar_1+ar_2}\right) \\ &+ fp_2\left(w * \frac{ar_2}{ar_1+ar_2}\right). \end{aligned} \tag{9}$$

To reiterate, this equation states that in an interleaved memory access trace from two non-data-sharing programs, the expected number of *distinct* memory addresses accessed in w total accesses is the sum of the expected number of distinct memory addresses accessed by each program. The latter is calculated given each of their *independently* measured average footprint functions and the number of total accesses belonging to each, based on their access rates ($w * \frac{ar_1}{ar_1+ar_2}$ and $w * \frac{ar_2}{ar_1+ar_2}$).

What we are most interested in, the miss ratio, can be derived from Equations the miss ratio can be calculated using Equations 6, 7 and 8 [16]:

$$mr(c) = fp(w+1) - c, \tag{10}$$

where w is chosen so that $fp(w) = c$. In the scope of co-run programs (using Equation 9), we rewrite this as

$$\begin{aligned} mr(c, ar_1, ar_2) =& fp_1\left((w+1) * \frac{ar_1}{ar_1+ar_2}\right) \\ &+ fp_2\left((w+1) * \frac{ar_2}{ar_1+ar_2}\right) - c, \end{aligned} \tag{11}$$

where w is still chosen so that $fp(w) = fp_1(w * \frac{ar_1}{ar_1+ar_2}) + fp_2(w * \frac{ar_2}{ar_1+ar_2}) = c$. Since the footprint function is monotonic, the appropriate w can be found in $O(log(c))$ time using binary search. If both programs always had the same miss ratio, then the above equation would be a sufficient prediction of the miss ratio. However, since both programs' access rates vary with time, and we cannot predict what they will be at any given moment[4], we must treat the access rates as independent random variables.

V. Cache Partitioning

A. Natural Cache Partition

At a given time, each program sharing a cache will have some quantity of data in the cache. This is called the program's *cache occupancy*, or c_i for program i. In a warm cache, the sum of the cache occupancies equals the size of the cache (i.e., the cache is full). The footprint metric gives us a way to predict the cache occupancies of each program for a cache in a steady state; We call the ordered set of cache occupancies $(c_1, c_2, \cdots)$ the *Natural Cache Partition (NCP)*.

[4]There may be some feedback between the two access rates via misses and cache stalls, but we leave this to future work.

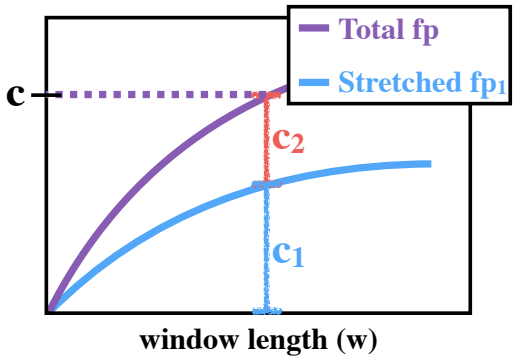

Figure 4: The natural cache partition is defined by the quantity of data in a cache expected to belong to each program. For two programs, the figure shows that the cache occupancy of each program can be predicted to be the individual (stretched) footprint of that program where the total footprint equals the cache size.

The overall footprint and the stretched footprints of a set of co-run programs can be used to determine the NCP. Of course, the individual stretched footprints always add up to the overall footprint. As illustrated in Figure 4, when the overall footprint equals the cache size, each individual footprint indicates that program's cache occupancy.

For programs with no phase behavior, one might expect that a shared cache would provide the same performance as a "naturally" partitioned cache. We call this notion the *Natural Partition Assumption (NPA)*. Since the optimal partition is at least as good as the natural partition (by the definition of "optimal"), insofar as it is correct, the NPA indicates that the optimal cache partition is at least as good as sharing.

If the NPA holds, then every partition-sharing scheme corresponds to some partitioning scheme. And that partitioning scheme may not be the optimal one. Therefore, the optimal partition scheme offers an upper-bound on the partition-sharing options. This implies that if NPA is true, then we should always partition when possible. If not, then partition-sharing may be justifiable, as in the example in Figure 1.

The natural partition is derived using the HOTL theory described in Section III. NPA holds if and only if the HOTL prediction of the miss ratio is accurate. Previous studies have shown that the prediction was accurate as compared to the measurement from simulation and hardware performance counters. We further discuss the validation of the NPA in Section VII-C.

B. Optimal Partitioning

The idea of memory or cache partitioning has been around for a while. In 1992, Stone, et al. proposed cache partitioning as an improvement over sharing with LRU both for

single-threaded programs by partitioning the cache between instructions and data, and for multiprogramming by giving each process a partition [9].

In the way of optimizing their algorithm, they proved that the average miss ratio for two or more uniformly interleaved processes is minimized when the derivative (with respect to partition size) of the miss ratio function for each process is equal. The proof, for only two programs, is straightforward. The total number of misses in some interval T is

$$\left(\frac{1}{2}mr_1(c_1) + \frac{1}{2}mr_2(C - c_1)\right) * T. \tag{12}$$

Assuming that each miss ratio function is convex and decreasing[5], setting the derivative of this function, with respect to program 1's cache size, equal to zero gives

$$\begin{aligned}\frac{dmr_1(c_1)}{dc_1} &= -\frac{dmr_1(C - c_1)}{dc_1} \\ &= \left.\frac{dmr_2(c_2)}{dc_2}\right|_{c_2=C-c_1}.\end{aligned} \tag{13}$$

And it is easy to show for the more general case where program i represents some fraction f_i of the trace, that the optimal partitioning exists when

$$f_1 * \frac{dmr_1(c_1)}{dc_1} = f_2 * \left.\frac{dmr_2(c_2)}{dc_2}\right|_{c_2=C-c_1}. \tag{14}$$

We call this the *Stone, Thiebault, Turek, Wolf (STTW) Cache Partitioning*. The miss ratio derivatives are also equal when there are more than two caches.

Optional Cache Partition Dynamic Programming Algorithm: The optimal partitioning can be determined using a dynamic programming algorithm. Given a set of programs P and a cache of size C, and letting $mc_i(c_i)$ be the miss count (that is, miss ratio times number of memory accesses) of program P_i at a cache size of c_i, the object is to determine the set S of partition sizes so that $\sum_i mc_i(c_i)$ is minimized, and $\sum_i c_i = C$:

$$S \triangleq \underset{(c_1,c_2,\cdots)}{\operatorname{argmin}} \left\{ \sum_i mc_i(c_i) \;\middle|\; \sum_i c_i = C \right\} \tag{15}$$

The dynamic programming algorithm constructs S program-by-program. When each program P_i is added, the new program is assigned the c_i that minimizes the sum of its miss count and the miss count of the optimal partitioning of the first $i-1$ programs with cache size of $C - c_i$. Letting $S_{k,i}$ represent the sub-solution for the first i programs with cache size k and $mc(S_{k,i})$ represent the total miss count for that sub-solution, each step of the algorithm is

$$\begin{aligned} c_i &= \underset{c_i}{\operatorname{argmin}} \{mc(S_{C-c_i,i-1}) + mc_i(c_i)\} \\ S_{C,i} &= S_{C-c_i,i-1} + (c_i). \end{aligned} \tag{16}$$

The time-complexity of this algorithm is $O(PC^2)$. The space complexity is $O(PC)$.

[5]The convexity assumption is something like the law of diminishing returns; the larger the cache, the less a program benefits by adding another byte to the cache. However, miss ratio curves often have drop-offs where a particular working set fits into the cache. The imperfection of this assumption is addressed by Thiébaut et al. (1992) [11]. The assumption that the miss ratio curve is non-increasing is because of the inclusion property of LRU caches.

VI. Baseline Optimization

In this section, we consider the optimization of fair cache sharing. In order to allow some group miss rate reduction without compromising fairness, we define two different types of *baseline optimization*, in which the above dynamic programming algorithm can be used, but with an added constraint at each step:

Equal Baseline: The group's miss ratio is minimized under the constraint that no single program has a higher miss ratio than it would with an equal partition of the cache.

Natural Baseline: The group's miss ratio is minimized under the constraint that no single program has a higher miss ratio than it would with the natural cache partitioning.

Fairness in cache has been studied extensively. There are many definitions. One may define it by sharing incentive, i.e., how the actual performance compares to equal partition. Alternatively, one may define it by interference, i.e., how the actual performance compares to having the full cache. The difference may seem artificial as in the proverbial question whether a glass is half empty or half full. However, the interference is more complex to define, i.e., how much increase in the miss ratio is too much, and whether we count the increase or the relative increase. In this work, we define fairness by the sharing incentive and will use the optimization technique described in Section V to optimize the performance of fair cache sharing.

VII. Experimental Design

In this section, we first describe the methods of evaluation, then evaluate the effect of optimal cache sharing, and finally discuss the issues of validation.

A. Methodology

Following the methodology of Wang et al. [12], we randomly choose 16 programs from SPEC 2006: *perlbench, bzip2, mcf, zeusmp, namd, dealII, soplex, povray, hmmer, sjeng, h264ref, tonto, lbm,omnetpp, wrf, sphinx3* and use the first reference input. The selection is representative, and the number of co-run groups is not too many to visualize. We enumerate all 4-program subsets, which gives us 1820 groups. We model their co-run miss ratio in the 8MB shared cache. The shared cache is partitioned by the unit of 128 cache blocks (8KB). There are a total of 1024 units.

For each set of 4 programs, the total number of partitions is $\binom{1026}{3}$ or $\approx$ 180 million (from Equation 3). The

complexity of dynamic programming is $O(PC^2)$, where $P = 4, C^2 = 1024^2$. PC^2 calculates to about 4 million. The granularity of 8KB is chosen to reduce the cost of dynamic programming, which is $128^2 = 16384$ times smaller when partitioning in 8KB units than in 64-byte cache blocks.

Among the solutions of cache sharing in each group, we model the performance of six solutions:

- *Equal*: each program has 2MB cache.
- *Natural*: the performance when the four programs share the 8MB cache.
- *Equal Baseline*: Group optimization with individual baselines set at the equal partition's performance.
- *Natural Baseline*: Group optimization with individual baselines set at the natural partition's performance.
- *Optimal*: the solution with the lowest group miss ratio.
- *STTW*: the classic solution to minimize the group miss ratio.

Many studies in the past have examined some but not all solutions of cache sharing. Using the terminology of Xie and Loh, Natural is the same as the capitalist cache sharing, and Equal the same as the socialist sharing [17]. The capitalist miss ratio depends on peers (market condition), but the socialist miss ratio is guaranteed at all times. In this study, we are the first to examine the entire solution space of partition sharing.

We show results for all 4-program groups. The exhaustive evaluation is important, since a random subset from these 1,840 groups can mislead since the subset result may differ significantly from other subsets and from the whole set. There is no sure way to choosing a representative subset unless we have evaluated the whole set. The exhaustive evaluation is possible because of the techniques developed in this paper.

The Implementation and the Cost of Analysis: The implementation of dynamic programming is in C++. The scripting is by Ruby. To test the speed, we use a machine with 1.7GHz Intel Core i5-3317U (11 inch MacBook Air, power cord unplugged). For all 1820 groups, it takes 4 minutes 20 seconds real time for optimal partitioning, 5 minutes 8 seconds for natural baseline optimization, and 4 minutes 3 seconds for equal baseline. For each group, the optimizer reads 4 footprints from 4 files. There are 16 footprint files for the 16 programs. The size ranges between 242KB and 375KB. The file size can be made smaller by storing in binary rather than ASCII format. The implementation has not been optimized for speed, but it is fast enough — on average it takes less than 0.21 second to optimize a program group. In comparison, it takes STTW 3 minutes 38 seconds for 1820 groups or 0.11s per group.

The footprint measurement we use is the same as the implementation by Xiang et al. [16] and Wang et al. [12] Xiang et al. reported on average 23 times slowdown from the full-trace footprint analysis. Wang et al. developed a sampling method called adaptive bursty footprint (ABF) profiling, which takes on average 0.09 second per program. To have reproducible results, our implementation uses the full-trace footprint.

B. Optimization Results

The Effect of Cache Sharing — Natural vs Equal: In a co-run group, we call a program a gainer if its miss ratio in the shared cache is lower than in the equal partition, and a loser if the opposite is true. Because of the natural-partition assumption, the comparison can be made between Natural and Equal, as shown in Figure 5.

In *lbm* and *sphinx3*, Natural is nearly always lower than Equal. Hence, the two programs mostly gain from sharing the cache. In *perlbench* and *sjeng*, Natural is almost never lower than Equal. The latter two programs almost never gain from sharing. All programs can have a large relative change in the miss ratio due to cache sharing, although the absolute change may be small if the miss ratio is small to start with.

Sharing has a harmonizing effect to narrow the difference between program miss ratios. The plots in Figure 5 are ordered by the decreasing miss ratio of Equal. A tendency for programs with a high miss ratio (whose plots are at the front of the page) to gain from sharing and programs with a low miss ratio to lose by sharing. The division line is roughly 1.35%. However, the tendency is not strict. The program *perlbench* always loses by sharing, but its miss ratio is higher than two programs, *hmmer* and *tonto*, where sharing is mostly beneficial.

The large variation within a plot and between plots means that it is difficult to classify programs by its behavior in the shared cache. For example, 12 of the 16 programs have mixed gains and losses due to sharing. Classification by the miss ratio is also not effective, since the relative portions of gaining and losing cases do not strictly correlate with the miss ratio. In addition, classification does not solved the problem of deciding which cases gain or lose and by how much.

Effect of Optimization: Table I shows the improvement by Optimal over all the other methods. Optimal improves Natural by 26% on average. It is at least 10% better for 58% of the groups and 20% better for 45% of the groups. The improvement is greater for Equal. The average improvement in all groups is 125%, and 77% and 58% of the groups are improved by at least 10% and 20% respectively.

Unfairness of Optimization: While Optimal makes a group much better overall, it may be unfair and makes a member program worse. Consider the miss ratio of Natural or Equal as the baseline. Optimal is unfair if it increases the miss ratio of a program compared to its baseline. Individually, we see clear evidence of unfairness in Figure 5 — Optimal makes a program worse as often as it makes it better. This is true regardless which baseline we examine.

We may classify a program by how likely it gains or loses ground in Optimal. For example, *sphinx3* is almost

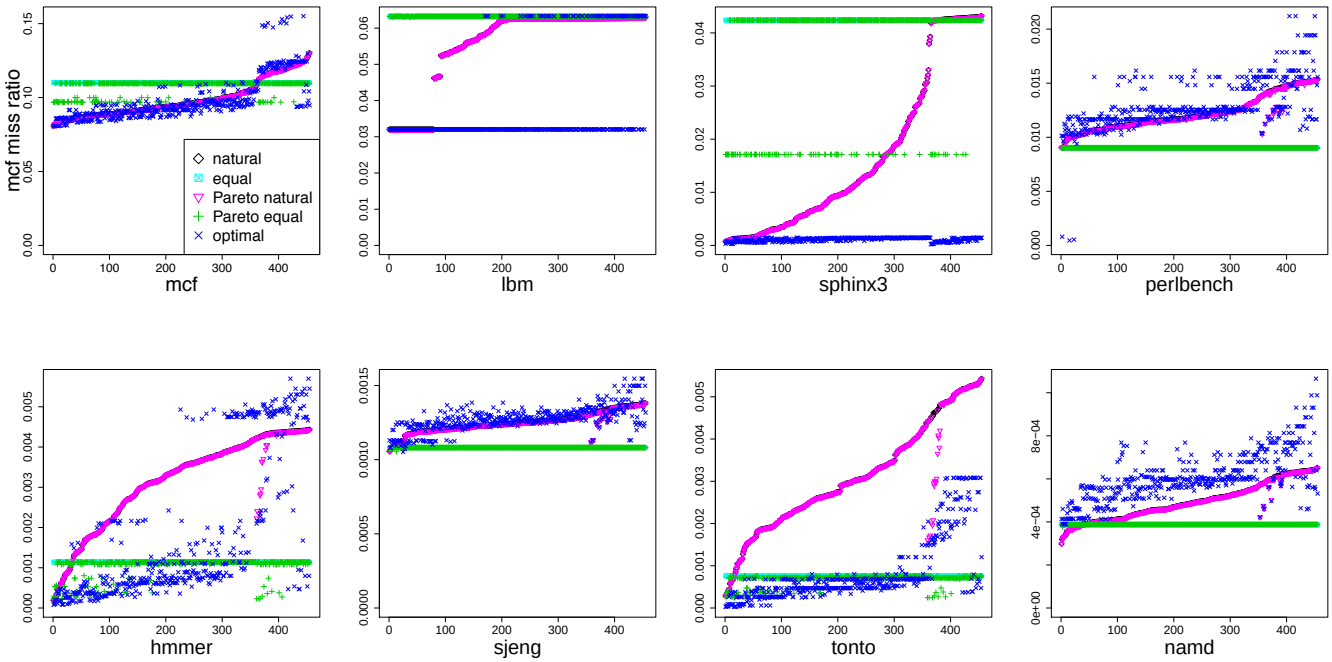

Figure 5: The miss ratio of individual programs running with different peer groups, under natural, equal, natural baseline, equal baseline, and optimal cache partition. Programs are sorted by the (constant) miss ratio in the equal partition. Under the other four schemes, the miss ratios vary with the peer group. Baseline optimization is at least as good as the baseline. Optimal may improve or degrade the individual miss ratio. The group miss ratios are shown in Figure 6 and summarized in Table I. Due to space limitations, we only show 8 representative programs out of the 16 tested. The others have similar behaviors.

always made better in Optimal, and *namd* is almost always made worse. Overall, Figure 5 shows that the optimization attempts to reduce the high miss ratios more than it increases the low miss ratios. However, there are plenty of exceptions. For example, two programs, *hmmer* and *tonto*, have very low miss ratios but gain rather lose from the optimization in most of their co-run groups. In almost all programs, whether to gain or lose depends on its co-run group, so is the degree of gain or loss.

Baseline Optimization: Baseline optimization is more effective for Equal than for Natural. As shown in the individual results in Figure 5 and the group results in Figure 6, Natural Baseline improves upon Natural in only a few cases. Table I shows similar improvements and a similar distribution of improvements for Optimal over Natural and over Natural Baseline. On the other hand, Baseline Equal improves upon Equal in far more cases for both the individual and the group miss ratio. On average, Baseline Equal is better than Equal by nearly 30%. The different improvements from the two baselines make a clear contrast. It shows that there is much more under-utilized cache space in equal partitions than in natural partitions.

To use the terminology of Xie and Loh [17], there is more resource under utilization in the socialist allocation than in the capitalist allocation. Note that this conclusion cannot be made without baseline optimization, which is only possible because of the techniques developed in this paper.

Sampling is Unscientific: The shape and the range of Natural miss ratios in all groups cannot be predicted by taking a few random points. It is almost guaranteed that differently sampled groups have few results in common. If a program always or never gains by cache sharing, sampling can recognize these cases. However, the amount of gains and losses is consistently inconsistent and cannot be fully analyzed by sampling.

Stone-Thiebaut-Turek-Wolf (STTW): The classic solution of Stone et al. minimizes the group miss ratio through convex optimization. We show the group-by-group comparison between STTW and optimal in Figure 7 and the statistical comparison in the bottom row in Table I labeled STTW.

When the convexity assumption holds, STTW is as good as optimal. However, as shown in Table I, in 34% of co-run groups, STTW is at least 10% worse than optimal, showing that the convexity assumption does not hold at least for

Methods of partitioning	Improvement by Optimal			Improved by at least	
	Max	Avg	Median	10%	20%
Equal	4746.43%	125.25%	26.48%	77.08%	57.80%
Equal baseline	2954.52%	97.75%	22.50%	70.27%	52.69%
Natural	266.78%	26.35%	14.51%	57.80%	45.16%
Natural baseline	266.78%	26.21%	14.29%	56.81%	45.10%
STTW	306.55%	33.68%	2.50%	34.39%	33.02%

Table I: The improvement of group performance by Optimal partition over five other partitioning methods. The last two columns show the percent groups that are improved by at least 10% and 20% respectively.

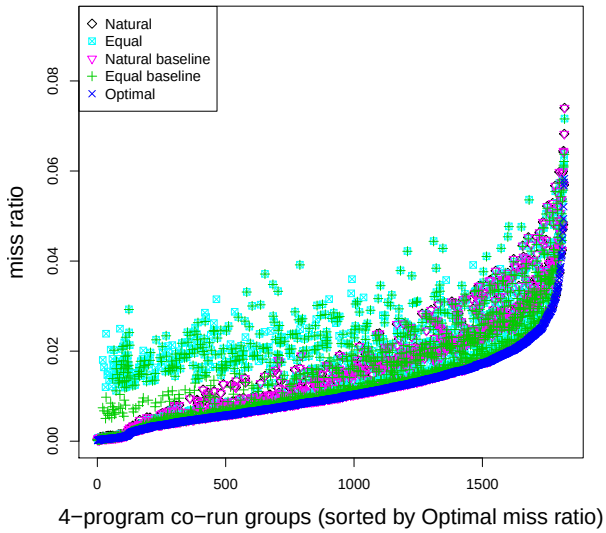

Figure 6: The group miss ratio of the five partitioning methods.

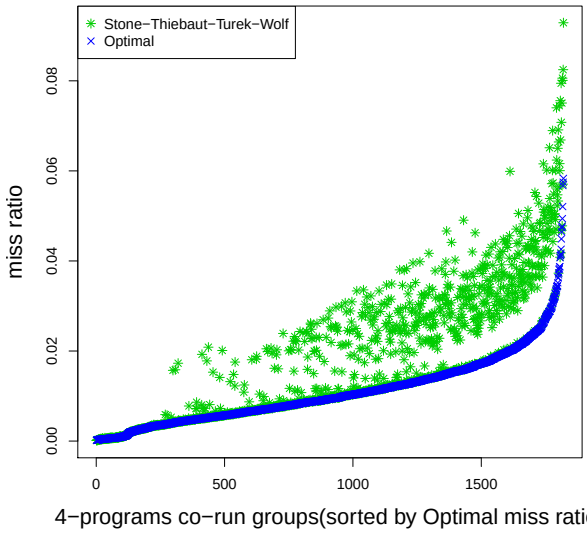

Figure 7: The group miss ratio of Optimal and STTW.

these cases. In most of these groups, STTW is at least 20% worse. For these groups, STTW is actually worse than the natural partition, shown by the higher average improvement of Optimal over STTW (34%) than over Natural (26%).

These results show that the convexity assumption is flawed to the extent that STTW optimization has the worse average performance than the simple free-for-all cache sharing. The problem is exacerbated when more programs share the cache, since a larger group increases the chance of the violation of the assumption by one or more members. In addition, STTW cannot optimize for fairness. The problems of STTW have been solved by our new algorithm, which improves the maximal performance by 34% on average over STTW and enables baseline optimization of fairness.

C. Validation

Previous work has measured the accuracy of the HOTL theory. Xiang et al. compared the prediction for two-program pairs [16]. They tested all 190 pair combinations. For each pair, they repeatedly ran each program so the co-run effect was stable from one test to another,[6] and measured the miss ratio for each program using three hardware counters:

```
OFFCORE_RESPONSE_0.DATA_IN.LOCAL_DRAM
MEM_INST_RETIRED.LOADS
MEM_INST_RETIRED.STORES
```

The first is the miss count, and the sum of the last two is the access count. Their ratio is the miss ratio. Xiang et al. compared the 380 measured miss ratios with the prediction and showed the results in Figure 9 in [16]. The prediction was accurate or nearly accurate for all but two miss ratios. From their results, we draw the conclusion that the natural partition assumption is largely true.

We do not simulate system performance for several reasons. First, our purpose and expertise both lie in pro-

[6]The stable result avoids the problem of performance variation, which can be modeled using Sandberg et al. [5]

gram analysis and optimization, not hardware cache design. Second, our goal is not to maximize performance, which depends on more factors than we can rigorously study in one paper. By focusing on cache and the slowdown-free miss rate, the achieved optimality is actually CPU independent, i.e., optimal regardless of the CPU design.

A simulator, especially for a multicore system, has many parameters. We are not confident that we can produce the same accuracy that we can with locality modeling. Finally, simulation is slow. Most computer architecture studies simulate a small fraction of a program. For example, Hsu et al. used a cache simulator called CASPER to measure the miss ratio curves from cache sizes 16KB to 1024KB in increments of 16KB [1]. They noted that "miss rate errors would have been unacceptably high with larger cache sizes" because they "were limited by the lengths of some of the traces." Our analysis considers all data accesses in an execution.

VIII. Discussion on Optimality

The nature cache partition as we have derived in Section V implies that optimal cache partition is the optimal solution for partition sharing. This optimality requires a number of assumptions, which we enumerate and discuss here.

HOTL Theory Correctness: The HOTL theory assumes the reuse window hypothesis, which means that the footprint distribution in reuse windows is the same as the footprint distribution in all windows [16]. When the hypothesis holds, the HOTL prediction is accurate for fully associative LRU cache. The correctness has been evaluated and validated for solo-use cache, including the two initial studies of the footprint theory [15], [16]. Independent validation can be found in the use of the footprint theory in optimal program symbiosis in shared cache [12], optimal memory allocation in Memcached [2], and a study from the OS community on server cache performance for disk access traces [13].

Random Phase Interaction: We assume that programs interact in their average behavior. Programs may have phases, but their phases align randomly rather than deterministically. An example of synchronized phase interaction is shown at the beginning of the paper in Figure 1. The natural partition does not exist since no cache partition can give the performance of cache sharing. However, synchronized phase interaction is unlikely for independent applications. It is unlikely that they have the same-length phases, and one program always finishes a phase just when a peer program starts another phase. We assume that the phase interaction is random. This assumption is implicit and implicitly validated in Xiang et al., who tested 20 SPEC programs and found that HOTL is acceptable in 99.5% of cases of $\binom{20}{2}$ possible paired running [16].

Fully Associative LRU Cache: The locality theory is partly mathematical, and the mathematics is critical in establishing the formal connection between program- and machine-level concepts, e.g., from the reuse time to the miss ratio. In comparison, the real cache is set-associative, and not all cache sizes are possible. In addition, the replacement policy may be an approximation or improvement of LRU. Past work has validated the theory result by measuring the actual miss ratio (using the hardware counters) on a real system. Xiang et al. showed accurate prediction by the HOTL theory for all three levels of cache on a test machine (Figure 6 in [16]). In addition, the HOTL theory can derive the reuse distance, which can be used to statistically estimate the effect of associativity [8], and as Sen and Wood recently showed, the performance of non-LRU policies [7].

Locality-performance Correlation: A recent study by Wang et al. shows that the HOTL-based miss ratio prediction has a linear relationship between execution time, with a coefficient of 0.938 [12]. They measure execution times and miss ratios of all 1820 4-programs co-run groups from a set of 16 SPEC programs. Thus, reducing execution time can be achieved though reducing same portion of miss ratio.

Practicality: The profiled metrics are average footprint, total number of memory accesses, and solo-run time for each program. Xiang et al. reported on average 23 times slowdown from the full-trace footprint analysis [16]. Wang et al. developed a sampling method called adaptive bursty footprint (ABF) profiling, which takes on average 0.09 second per program [12]. To have reproducible results, our implementation uses the full-trace footprint. We assume that in practice, the data can be collected in real time.

While these assumptions do not always hold, they have been carefully studied and validated through experiments on actual systems. Sections VII-C and IX give more details on some of these studies. In this paper, we use these assumptions to develop optimal partition sharing for general programs on any size cache. For hardware cache design, these assumptions may not be adequate, and careful simulation may be necessary. However, our objective here is narrow and specific, which is a machine-independent strategy for program co-run optimization for cache performance.

IX. Related Work

The Footprint Theory: Optimization must be built on theory, which in this case is the higher-order theory of locality (HOTL) [16]. As a metric, footprint quantifies the active data usage in all time scales. Because the footprint is composable, it can be used to predict the performance of cache sharing. In this paper, we show a consequence of the HOTL theory: the performance of shared cache equals to a particular solution of cache partitioning called the natural partition, which we then use to optimize not just cache sharing, but also cache partitioning and partition sharing.

Three recent studies provide fresh evidence on the accuracy of the footprint analysis in modeling fully-associative LRU cache. Wang et al. tested the analysis on program execution traces for CPU cache [12], Hu et al. on key-value access traces for Memcached [2], and Wires et al.

on disk access traces for server cache [13]. The three studies re-implemented the footprint analysis independently and reported high accuracy through extensive testing. Hu et al. tested the speed of convergence, i.e., how quickly the memory allocation stablizes under a steadystate workload, and found that optimal partition converges 4 times faster than free-for-all sharing [2]. Finally, Wang et al. showed strong correlation (coefficient 0.938) between the predicted miss ratio and measured co-run speed [12]. The correlation means that if we minimize the miss ratio in shared cache, we minimize the execution time of co-run programs.

Cache Partitioning: Cache partitioning is an effective solution to improve performance, which has been shown by many studies including the following two on optimal cache partitioning. Stone et al. gave a greedy solution to allocate cache among N processes, which allocates the next cache block to the process with the highest miss-rate derivative. The allocation is optimal if the miss-rate derivatives are as equal as possible [9]. The optimality depends on several assumptions. One is that the miss-rate derivative must be monotonic. In other words, the MRC must be convex. Suh et al. gave a solution which divides MRC between non-convex points but concluded that the solution "may be too expensive" [10].

The previous optimization is limited to cache partitioning. Here we optimize for both partitioning and sharing. Furthermore, our solution does not depend on any assumption of the MRC curve. It can use any cost function. The generality is useful when optimizing with constraints or for multiple objectives, e.g., both throughput and fairness in elastic cache utility optimization [18].

Concurrent Reuse Distance: Simulation of shared cache is costly, and the result is specific to the cache being simulated. More general solutions are based on the concept of concurrent reuse distance (CRD), which shows the performance of both partitioned and shared cache for all cache sizes [3], [6], [14]. However, CRD is for a given set of programs and must be measured again when the set changes. It cannot derive the optimal grouping of programs, which is needed for partition sharing.

X. Summary

In this paper we presented partition-sharing as a general problem for assigning cache space to programs running on multi-core architectures. We enumerated the search spaces for 5 different sub-problems of partition-sharing, used the Higher Order Theory of Locality to show that the best partitioning-only solution must be the optimal partition-sharing solution, and presented a new algorithm for attaining optimal partitioning solution. In addition, we defined two resource-efficient fairness conditions, *equal baseline* and *natural baseline*, and used a modified algorithm to optimize performance under each one. Experiments show that the baseline optimization can significantly improve equal partitioning but not free-for-all sharing. Each optimization result is obtained from a very large solution space for different ways to share the cache.

This paper is subtitled with a quote from Robert Frost: "Don't ever take a fence down until you know why it was put up". We argue that despite there being a vast number of options for allocating shared cache among programs, the best option is usually to assign a partition to each program. If we are to trust the great American poet, as well as the now-established Higher Order Theory of Locality, we'd better default to leaving up fences between our co-run programs.

References

[1] L. R. Hsu, S. K. Reinhardt, R. R. Iyer, and S. Makineni. Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource. In *PACT*, pages 13–22, 2006.

[2] X. Hu, X. Wang, Y. Li, L. Zhou, Y. Luo, C. Ding, S. Jiang, and Z. Wang. LAMA: Optimized locality-aware memory allocation for key-value cache. In *Proceedings of USENIX ATC*, 2015.

[3] Y. Jiang, E. Z. Zhang, K. Tian, and X. Shen. Is reuse distance applicable to data locality analysis on chip multiprocessors? In *Proceedings of CC*, pages 264–282, 2010.

[4] R. Parihar, J. Brock, C. Ding, and M. C. Huang. Protection and utilization in shared cache through rationing. In *Proceedings of PACT*, pages 487–488, 2014. *short paper*.

[5] A. Sandberg, A. Sembrant, E. Hagersten, and D. Black-Schaffer. Modeling performance variation due to cache sharing. In *Proceedings of HPCA*, pages 155–166, 2013.

[6] D. L. Schuff, M. Kulkarni, and V. S. Pai. Accelerating multicore reuse distance analysis with sampling and parallelization. In *Proceedings of PACT*, pages 53–64, 2010.

[7] R. Sen and D. A. Wood. Reuse-based online models for caches. In *Proceedings of SIGMETRICS*, pages 279–292, 2013.

[8] A. J. Smith. On the effectiveness of set associative page mapping and its applications in main memory management. In *Proceedings of ICSE*, 1976.

[9] H. S. Stone, J. Turek, and J. L. Wolf. Optimal partitioning of cache memory. *IEEE Transactions on Computers*, 41(9):1054–1068, 1992.

[10] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 28(1):7–26, 2004.

[11] D. Thiébaut and H. S. Stone. Improving disk cache hit-ratios through cache partitioning. *IEEE Transactions on Computers*, 41(6):665–676, 1992.

[12] Wang et al. Optimal program symbiosis in shared cache. In *Proceedings of CCGrid*, June 2015.

[13] J. Wires, S. Ingram, Z. Drudi, N. J. Harvey, A. Warfield, and C. Data. Characterizing storage workloads with counter stacks. In *Proceedings of OSDI*, pages 335–349. USENIX Association, 2014.

[14] M. Wu and D. Yeung. Efficient reuse distance analysis of multicore scaling for loop-based parallel programs. *ACM Trans. Comput. Syst.*, 31(1):1, 2013.

[15] X. Xiang, B. Bao, C. Ding, and Y. Gao. Linear-time modeling of program working set in shared cache. In *Proceedings of PACT*, pages 350–360, 2011.

[16] X. Xiang, C. Ding, H. Luo, and B. Bao. HOTL: a higher order theory of locality. In *Proceedings of ASPLOS*, pages 343–356, 2013.

[17] Y. Xie and G. H. Loh. Dynamic classication of program memory behaviors in CMPs. In *CMP-MSI Workshop*, 2008.

[18] C. Ye, J. Brock, C. Ding, and H. Jin. RECU: Rochester elastic cache utility. In *Proceedings of NPC*, 2015.