Cache Exclusivity and Sharing: Theory and Optimization

A problem on multicore systems is cache sharing, where the cache occupancy of a program depends on the cache usage of peer programs. Exclusive cache hierarchy as used on AMD processors is an effective solution to allow processor cores to have large private cache while still benefit from shared cache. The shared cache stores the "victims", i.e. data evicted from private caches. The performance depends on how victims of co-run programs interact in shared cache.

This paper presents a new metric called victim footprint. It is measured once per program in its solo execution and can then be combined to compute the performance of any exclusive cache hierarchy, replacing parallel testing with theoretical analysis. The paper tests victim footprint on parallel mixes of sequential programs, compares the accuracy of the theory with hardware counter results, and evaluates the benefit of exclusivity-aware analysis and optimization.

Additional Key Words and Phrases: Cache Analysis; Cache Sharing; Exclusive Cache

ACM Reference format:
. 2016. Cache Exclusivity and Sharing: Theory and Optimization. 1, 1, Article 1 (January 2016), 23 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In an *exclusive cache hierarchy*, the lower level stores the evictions of the upper level, and the content of the two levels has no intersection. The exclusive cache hierarchy has long been the choice of AMD multicore processors. It has several important benefits over an inclusive hierarchy.

First, there is no data duplication, and the cache space is better utilized. We call this effect *deduplication*. Second, the lower level cache stores evicted data. It may be called the *victim cache*. The sharing of the victim cache is between the evicted data from multiple programs. We call the second effect *filtering*.

This paper presents a new theory to model an exclusive cache hierarchy. It first formalizes an abstract cache architecture called a split LRU stack. Then it defines a new measure of program locality called *victim footprint* and shows how it can predict the performance of the program on a split LRU stack. With the performance prediction, it shows how to optimize the sharing of exclusive cache.

The new theory of victim footprint solves mainly two problems in performance modeling. The first is composable locality analysis. The victim footprint is measured once for each program and can then be composed to compute the combined locality for any program group. The second is portable prediction. The combined victim footprint can be used to predict the miss ratio in any shared cache without rerunning these programs.

The new theory assumes fully associative LRU cache. With this assumption, the composition and prediction techniques are entirely mathematical. Such mathematical operations are simple and precise to present. In fact, the effect of sharing will be entirely shown in mathematical terms for single-level (Eq. 13) and multi-level (Eq. 14) sharing.

The mathematical operations are parameterized across all program groups and cache sizes. Different generations of multicore processors from different vendors can be modeled by simply changing

New Paper, Not an Extension of a Conference Paper.

2016. XXXX-XX/2016/1-ART1 $15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

these parameters but using the same victim footprint. The simplicity and portability make experimental results more readily reproducible by others. More importantly, we show that the new theory produces realistic results.

Cache modeling is an important but long standing problem to quantitatively explain and improve cache performance. A common strategy is direct measurement, e.g. through hardware counters. For shared cache, direct execution results for one program group on one machine are not predictive of other groups or other machines. To be composable and portable, more theoretical models have been developed. Early models are based on reuse distance [5, 41, 50], which was costly to measure. More recent models, in particular StatStack [14, 15] and HOTL (higher order theory of locality) [53], use reuse time, which can be measured much faster. The new theory of victim footprint is an extension of the HOTL theory.

The paper first introduces the HOTL theory as the background (Section 2) and then presents the new theory and its evaluation:

- *Victim footprint formulation*, which sets up the necessary formalism, including the definitions of split LRU (Section 3.1), victim footprint and victim cache fill time, and their properties especially the uniqueness theorem (Section 3.2).
- *Composable analysis and portable prediction*, which uses victim footprint to model data sharing in different types of CPU cache designs such as AMD exclusive caches, Intel cache allocation technology (CAT), and IBM transient loads (Section 3.3).
- *Evaluation*, which tests the victim footprint on three types of AMD and Intel machines for performance prediction, optimization, and comparison with the previous theory and heuristic solutions (Section 4).

The study has three more limitations. First, it uses the theory in postmortem analysis with the full execution trace of each program. In practice, sampling can enable on-line analysis (shown by prior work discussed in Section 5). Second, it models cache performance, i.e. the number of accesses in main memory, not running time. It is oblivious of the processor architecture or the effect of prefetching. Finally, while the HOTL theory (Section 2) is recently extended to model data sharing [28], in this paper we assume no data sharing between co-run programs.

2 BACKGROUND: HOTL THEORY

The higher-order theory of locality (HOTL) defines a set of metrics and uses them to compute the miss ratio in shared cache [53]. The most important metric is a locality profile called the *footprint*. The footprint is defined by an integer function, and cache modeling consists of mathematical operations on such functions. Next we review these definitions and operations.

The working set is the classic locality model defined by Denning [8]. In an execution trace, each time window is represented by (t, x), where t is the end position and x the window length. The number of distinct elements in the window is the *working-set size* $\omega(t, x)$. For each x, $fp(x)$ is the average working-set size of all windows of length x, i.e. the total working-set size divided by the number of length-x windows as shown by the following equation:

$$fp(x) = \frac{1}{n - x + 1} \sum_{t=x}^{n} \omega(t, x) \tag{1}$$

where the parameter x is an integer time scale $0 \le x \le n$, and n the trace length.

From the footprint, the HOTL theory computes two metrics for fully-associative LRU cache of size c:

- *miss ratio* $mr(c)$, computed as the derivative of the footprint.

- *fill time* $ft(c)$, which is the average amount of time for a program to access data equal to cache size c, computed by the inverse of the footprint.[1]

Mathematically, we have:

$$mr(c) = fp(x+1) - fp(x) \text{ where } c = fp(x) \tag{2}$$

$$ft(c) = x \iff c = fp(x) \tag{3}$$

The two metrics are mappings between the cache size and the window length, which Figure 1

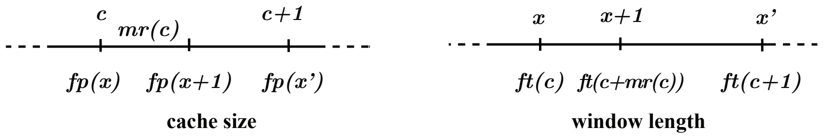


Fig. 1. Illustration of the footprint and fill time defined in the HOTL theory (Eq. 1 and 3). The left line shows the cache size and the footprint, and the right line the window length and fill time. The former also shows the mapping from time to space, and latter from space to time, with the mapping functions underneath.

The second line in Figure 1 shows the window length and equivalently the fill time, i.e. the average time it takes to fill an empty cache. It is the same as the average eviction time, i.e. the average time before a data block is evicted from cache after its last access [22].

The miss ratio is the increase of the footprint from $fp(x)$ to $fp(x+1)$. Given a window of length x, its working set is expanded at the next access if and only if it is a miss. The footprint increase from x to $x+1$ is the increase of the average working-set size of all length-x windows. The HOTL theory states that this footprint increase is the probability of a miss, i.e. the miss ratio. Formally, the miss ratio is the derivative of the footprint, as stated by Eq. 2. This is a key link between locality and cache performance, the one we will use to develop the victim footprint.

3 VICTIM FOOTPRINT THEORY

This section first defines cache exclusivity using an abstract cache architecture called the split LRU stack and then develops the theory of victim footprint.

3.1 Split LRU Stack

An LRU stack is the classic cache model defined by Mattson et al. [31] A *split LRU stack* divides an LRU stack into two parts, the upper partition, H, which includes the first h stack positions (storing more recently accessed data), and the lower partition, L, which includes the rest of the stack. Figure 2(a) shows a split LRU stack.

Data evicted from the H partition are called victims. A victim is first stored in the lower partition until it is accessed again, when it is moved to H, or it is evicted. Since the L partition stores victims, we call it the *victim cache*.

The victim cache may be shared. Fig. 2 (b) shows an example where the lower partition serves the victims from two programs. On an AMD multicore processor, programs have private L1, L2 caches

[1]Xiang et al. called it volume fill time $vt(c)$ [53].

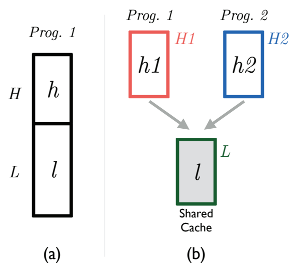


Fig. 2. Illustration of split LRU stacks. Two-level exclusive cache hierarchy used by (a) one program with a private victim cache (b) two programs sharing a victim cache. The symbols *H,L* mean physical caches and *h,l* their size.

but share one L3 cache [1]. The private higher partitions would be the L2s, and the victim cache would be the shared L3.

Fig. 3 simulates an LRU stack for a given access trace. The stack positions are numbered top down, with the top position storing the most recently accessed datum. At each position, the sequence of accesses form a trace.

Trace		a	b	c	d	d	c	b	a
LRU Stack	(1)	a	b	c	d	d	c	b	a
	(2)		**a**	**b**	**c**	c	**d**	**c**	**b**
	(3)			**a**	**b**	b	b	**d**	**c**
	(4)				**a**	a	a	a	**d**

Fig. 3. LRU stack simulation for a given access trace. The sequences at stack positions 2,3,4 are victim traces, with evictions marked in red (when viewed in color).

Generally in the split LRU stack, each row is a trace. The trace at the top position is the *access trace*. All others, the bottom three rows in Fig. 3, are *eviction traces* or *victim traces*. Each row shows the "access" to the victim cache for $h = 1, 2, 3$ respectively. In the victim cache, the content changes when there is an eviction. All evictions are marked in red in Fig. 3.

The split LRU stack defines the abstract cache architecture that we will model. In this abstract definition, all two-level, exclusive caches are specified by two parameters: the sizes of the two partitions h, l. Either can be any integer that is zero or larger.

In this section, we have defined the exclusive cache as the split LRU stack. Next, we solve the problem of modeling split LRU. First, we define the victim footprint for a victim trace. Second, we compute the effect of sharing in the victim cache.

3.2 Victim Footprint

The section defines the victim footprint and shows its relation with the miss ratio.

3.2.1 Victim Footprint and Victim Fill Time. The victim footprint is defined from the footprint $fp(x)$ and with one extra parameter h, the size of the H partition. Mathematically, the victim footprint is $vfp(h, x)$ for $h, x \geq 0$,

$$vfp(h, x) = fp(x_h + x) - h \text{ where } fp(x_h) = h \tag{4}$$

and its inverse function *victim fill time* $vft(h, c)$ for victim cache size c.

$$vft(h, c) = x \text{ when } vfp(h, x) = c \tag{5}$$

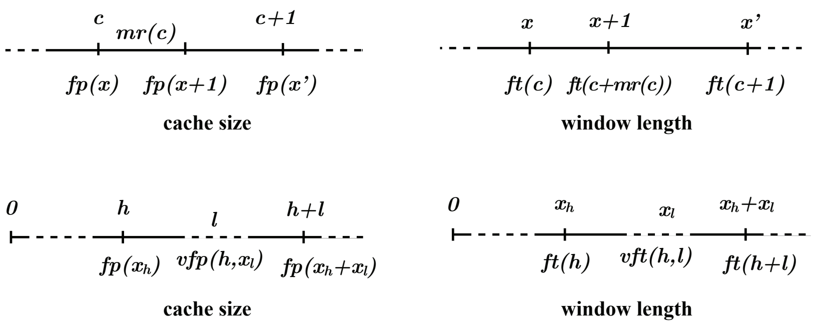
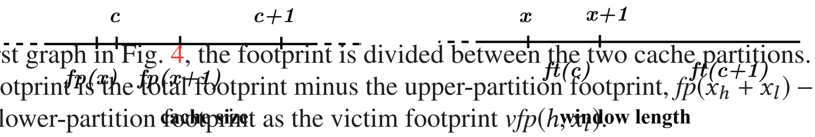


Fig. 4. The illustration of the victim footprint and victim fill time. They are similar to the footprint and fill time in Fig. 1 except that the two lines are divided into the segments corresponding to the split LRU stack, and the victim metric is the second segment on each line.



In the first graph in Fig. 4, the footprint is divided between the two cache partitions. The lower-partition footprint is the total footprint minus the upper-partition footprint, $fp(x_h + x_l) - fp(x_h)$. We define this lower-partition footprint as the victim footprint $vfp(h, x_l)$.

In the second graph in Figure 4, the fill time is divided into the time to fill the upper partition and the time to fill the lower partition. We define the latter time as the victim cache fill time, or formally $vft(h, x_l) = ft(h + l) - ft(h)$.

From Figure 4, a reader can observe the inverse relation between l and x_l in the two victim metrics, i.e. $l = vfp(h, x_l)$ on the left and $x_l = vft(h, l)$ on the right.

As a derived function, the victim footprint differs from the footprint in two aspects. The first change is the function value. The victim footprint is l, while the footprint is $h + l$. The second change is the function parameter. The time window of the victim trace is x_l, while the time window of the access trace is $x_h + x_l$.

The metrics are defined for all $h, l \geq 0$ and therefore all split LRU stacks. If we generalize and model any lower partition by using x instead of x_l and recognize $fp(x_h) = h$, we obtain the definition in Eq. 4. The parameters in $vfp(h, x)$ represent the general case when h is given, and l is unknown.

3.2.2 *Victim Cache Miss Ratio.*

Consider a split LRU stack with two partitions H, L of sizes h, l. We define $vmr(h, l)$ as the miss ratio of the lower partition:

- *The Victim cache miss ratio $vmr(h, l)$* is the number of misses in the lower partition divided by the number of *all* accesses.

Note that the denominator of $vmr(h, l)$ counts not just the misses of the higher partition but all accesses (to the higher partition).

Following the HOTL theory that the miss ratio is the increase of the footprint, we compute the victim cache miss ratio $vmr(h, l)$ as the increase of the victim footprint:

$$vmr(h, l) = vfp(h, x + 1) - vfp(h, x) \tag{6}$$

where h is the size of H, $vfp(h, x)$ is the victim footprint, and $vfp(h, x) = l$, the size of L. The equation is identical to Eq. 2, except that we substitute the cache size by l and footprint by victim footprint. The computed miss ratio is the fraction of all accesses in the trace.

3.2.3 *Correctness and Uniqueness.*

We formalize the requirement for any model of the split LRU stack and then derive the victim footprint as the one and only solution that satisfies this requirement.

Consider the split LRU stack H, L of sizes h, l. When both partitions are used by a single program and not shared, the miss ratio of the split LRU stack should be identical to a single cache of the combined size $h + l$. Since both partitions serve the same program, they should behave collectively as a single, unified cache.

We state the *Victim Cache Requirement* (VCR) as follows. When only one program uses the exclusive cache hierarchy, the miss ratio of the victim cache must equal to the would-be miss ratio of a single cache with the combined space. Mathematically, the requirement means that

$$vmr(h,l) = mr(h+l) \tag{7}$$

for all $h, l \geq 0$. We call Eq. 7 the *Victim Footprint Requirement Equation* (VCR Equation). The VCR equation is a single equation, but it is parameterized — the equality must hold for every pair of $h, l \geq 0$.

We prove the *VFP Theorem*, which states that the victim footprint is not only correct, i.e. it satisfies the VFP Equation, but also unique, i.e. no other solution exists that satisfies the VFP equation.

THEOREM 3.1. (VFP Theorem) *The victim footprint in Eq. 4 is a unique solution that satisfies the victim cache requirement in Eq. 7.*

PROOF. We prove that any other solution $vfp'(h,x)$ must be equal to $vfp(h,x)$, by induction. In the base case, when the victim-trace window size is 0, both of vfp and vfp' naturally should be 0 :

$$vfp(h,0) = vfp'(h,0) = 0$$

For the inductive case, given that $vfp(h,x) = vfp'(h,x)$ for some $x \geq 0$, we show that $vfp(h,x+1) = vfp'(h,x+1)$.

Let $l = vfp(h,x) = vfp'(h,x)$. According to VCR, the derivative of $vfp'(h,x)$ must be equal to the miss ratio of the unified cache of size $h + l$. Hence, we have:

$$vfp'(h,x+1) - vfp'(h,x) = mr(h+l) \tag{8}$$

From Eq.4, we have:

$$\begin{aligned} vfp(h,x+1) - vfp(h,x) &= (fp(x+1+t_h) - h) - (fp(x+t_h) - h) \\ &= fp(x+1+t_h) - fp(x+t_h) \\ &= mr(fp(x+t_h)) \end{aligned}$$

Again from Eq.4, we have $fp(x+t_h) = vfp(h,x) + h = l + h$, so we have

$$\begin{aligned} vfp(h,x+1) - vfp(h,x) &= mr(fp(x+t_h)) \\ &= mr(h+l) \end{aligned} \tag{9}$$

Combining Eq.8, Eq.9, and the inductive assumption, we see that the inductive case holds and therefore $vfp'(h,x) = vfp(h,x)$ for all $x \geq 0$, and the solution to the VCR Equation is unique. □

The proof of the VFP Theorem is effectively the construction of $vfp(h,x)$ from $x = 0$. It can be equally used as the derivation of Eq. 4.

3.3 Composition Analysis of Cache Sharing

When sharing the cache, a set of co-run programs interact. We want a composable model to derive the composite effect from individual solo-run locality. We first consider the sharing of victim cache on AMD processors and then two other types of sharing on Intel and IBM processors.

3.3.1 Sharing Exclusive LLC. On AMD processors, each program has a private L2, and they all share the L3 cache (last-level cache LLC). L3 is the victim cache of all the L2s. Let the L2 size be c_2 and L3 size be c_3. Let the co-run group be $g = \{1, \ldots, p\}$ and their victim footprints be $vfp_i(h, x)$, $i = 1 \ldots p$. In this paper, we consider only independent programs that do not share data.

To explain the derivation we start with a symmetric case, where a group G consists of p identical co-run programs are executed together with uniform interleaving. The original footprint is defined by the *individual logical clock*, where a program makes a data access at every time tick. We first normalize the time to the *co-run logical clock*, where each program makes a data access at one out of every p ticks.

Because of the time change, the co-run victim footprint of each program is the original victim footprint "stretched" by a factor of p. For program i, the co-run footprint is $vfp_{G_i}(c_2, x) = vfp_i(c_2, \frac{x}{p})$. For the rest of this section, we use the letter G in the subscript position to symbolize a footprint in the co-run logical clock, i.e. after "stretching."

After normalizing the time, the aggregate victim footprint, $vfp_G(c_2, x)$, is simply p times the individual victim footprint:

$$vfp_G(c_2, x) = p\ vfp_{G_i}(c_2, x)$$

The victim-cache (L3) miss ratio is the derivative of the group victim footprint.

$$vmr_G(c_2, c_3) = vfp_G(c_2, x + 1) - vfp_G(c_2, x) \tag{10}$$

where $vfp_G(c_2, x) = c_3$.

We now consider a general group G, which differs from a symmetric group in two ways. First, each program i may have a different victim footprint $vfp_i(h, x)$. Second, in the parallel execution, each program may have a different access rate ar_i. An *access rate* is the number of loads and stores per second.

Group Miss Ratio. The aggregate access rate is $ar_G = \sum_{i \in G} ar_i$. The victim footprint of program i is "stretched" by $\frac{ar_i}{ar_G}$:

$$vfp_{G_i}(c_2, x) = vfp_i(c_2, \frac{x\ ar_i}{ar_G}) \tag{11}$$

The group victim footprint is their aggregation:

$$vfp_G(c_2, x) = \sum_{i \in G} vfp_{G_i}(c_2, x) \tag{12}$$

Finally, the group miss ratio is computed the same way as in Eq. 10.

The way the access rate is used here is the same as in StatStack developed by Eklov et al. [14]. Brock et al. used to term *stretched footprint* [4] and called co-run logical clock *common logical time* [46].

Individual Miss Ratio. The miss ratio of individual programs in the shared cache is the derivative of the individual victim footprint taken at the point where *the group victim footprint is the cache size*.

$$vmr_{G_i}(c_2, c_3) = vfp_{G_i}(c_2, x + 1) - vfp_{G_i}(c_2, x) \tag{13}$$

where $vfp_G(c_2, x) = c_3$.

In the shared cache, a program is affected by its peers. We can now show the effect of this interaction in precise mathematical terms. The miss ratio of the solo-use exclusive cache was given in Eq. 6. Comparing this earlier equation with Eq. 13, we see that both take the derivative of the individual victim footprint. The difference is where this derivative is taken. The solo-use cache stores only self data, so differentiation happens when the individual victim footprint equals to the cache size. The shared cache is shared, so differentiation happens when the *group* victim footprint equals to the cache size.

Composition Invariance. The group miss ratio can be computed in two ways: directly from the victim footprint of the group (Eq. 10) or indirectly as the sum of individual miss ratios (Eq. 13). Composition invariance states that the two results be the same:

$$vmr_G(c_2, c_3) = \sum_{i \in G} vmr_{G_i}(c_2, c_3)$$

The proof of composition invariance is straightforward with simple re-arrangement of the terms in Eqs 10, 12 and 13. It relies on a mathematical property of victim footprints, which is that the two steps, taking the sum (composition) and taking the derivative (conversion to miss ratio), are commutative.

In the symmetric case, the composition invariance means that p identical programs sharing the cache is the same as them evenly partitioning the cache.

Early composable models used reuse distance and footprint and had only one way to compute the group miss ratio [5, 41, 51, 52]. Recent models are composition invariant, first from the higher order theory of locality (HOTL) [4, 46, 53] and recently based on the average eviction time (AET) [22]. The effect of cache sharing is computed differently. Previously in HOTL and presently in VFP, it is computed as the group footprint equals to the cache size. In AET, the quality means that all co-run programs have the same average eviction time.

3.3.2 Multi-level Sharing. AMD Bulldozer processors use cluster-based multi-threading (CMT), where each core (or module) has two clusters. The new Zen processors use simultaneous multi-threading (SMT) [32]. The two have the same effect of multi-level cache sharing. For example, a quad-core AMD processor can run 8 programs in 4 pairs where each pair share a separate L1 and L2, and all pairs share L3.

Victim footprint can be used to model multi-level sharing as follows. The sharing of L1 is solved by HOTL (Section 2). In particular, we use the recent result by Brock et al., who showed that HOTL implies the existence of a cache partition, called the *natural cache partition (NCP)*, whose partitioned-cache performance equals to that of cache sharing [4]. The natural partition of a program is its effective occupancy in the shared cache.

Consider the two-program group $G = \{a, b\}$ that share L1 and L2, where L2 is the victim cache of L1. According to Brock et al., the effect of L1 sharing is equivalent to a partition of L1 into two parts of size $c_{1,a}, c_{1,b}$, where $c_{1,a} + c_{1,b} = c_1$ the L1 size.

To compute the effect in L2, we use the solution of Section 3.3.1. It has three steps: footprint stretching in Eq. 11, footprint composition in Eq. 12 and miss ratio conversion in Eq. 10. The only change is the second step. Previously, each sharer program has the full (private) L2. Now, a, b have their natural partition of L1 as $c_{1,a}, c_{1,b}$. The second step then becomes:

$$vfp_G(c_1, x) = vfp_{G_a}(c_{1,a}, x) + vfp_{G_b}(c_{1,b}, x) \tag{14}$$

Finally, Eq. 10 computes the co-run miss ratio from this group footprint.

In multi-level sharing, the program interaction at one level affects their interaction at the next level. We can now show this effect in precise mathematical terms. The footprint aggregation of single-level sharing was given in Eq. 12. Comparing this earlier equation with Eq. 14, we see that the difference lies in the first parameter of the victim footprint, the size of the upper partition. In single-level sharing, the upper-level cache is private, so the size is constant in the earlier equation. In multi-level sharing, the upper-level cache is shared, so the size is a fraction of the total size. In the victim footprint, this first parameter can be any non-negative integer, so it can model any effect of sharing in the previous level.

We model cache sharing in any number of levels by applying the equations level by level. On AMD processors, all three levels of cache are exclusive and can be shared. We compute the natural cache partition of L1,L2. Then L3 sharing can be modeled the same way as L2 has been in this section.

Mathematically, the footprint is the victim footprint whose first parameter is 0, i.e. $fp(x) = vfp(0, x)$. Hence, the equations of the victim footprint give the general solution for any cache hierarchy, inclusive or exclusive.

Last, the Victim Cache Requirement (VCR) in Section 3.2.3 applies here. Since a, b share L1, L2, their co-run miss ratios in L2 should be the same as if they share just one cache of the combined size. Using the formalism of natural cache partition [4], we can prove that this multi-level VCR is satisfied by the victim footprint.

3.3.3 Cache-way Sharing. In set-associative LRU cache, each cache set is an LRU stack. We can collect the per-set footprint (and hence victim footprint) by taking the sequence of accesses for each cache set. The split LRU stack defined in Section 3.1 can be used to model two additional cache designs.

- **Intel's cache allocation technology (CAT):** Data of a program may be restricted (allocated) to use just a continuous group of cache ways [24]. When two CAT allocations partially overlap, there are three segments, as shown in Fig. 5(a). The first two segments are the two partitions of the split LRU for the first program, and the last two segments are the two partitions of the split LRU stack for the second program. The first L partition and the second H partition share the same cache space.
- **IBM's transient loads:** Memory loads have two types: normal loads and transient loads [23]. The data loaded normally go to the MRU (most recently accessed) position, while the data of transient loads go to the LRU position, as shown in Fig. 5(b). The design can be viewed as a case of CAT, where normal loads are allocated all cache ways, and transient loads only the last c

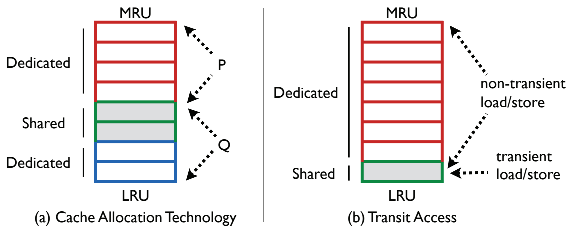


Fig. 5. Two designs of cache-way sharing:(a) cache-way allocation technology (CAT) on recent models of Intel processors, and (b) transient loads on IBM Power processors. In (a), the middle two ways are shared between programs P, Q. In (b), the last cache way is shared between non-transient loads/stores and transient loads/stores of the same program.

Victim footprint can be extended to model both types of cache-way sharing. Here we sketch only the basic ideas:

- **Intel's CAT:** In Fig. 5(a), the middle segment is shared between two programs. The interaction happens between the victim footprint of the first program and the footprint of the second program. The natural cache partition (NCP) is computed for the middle segment, and the effective occupancy for the second program determines the h parameter for its victim footprint for the third segment.

- **IBM's transient loads:** The last cache way in Fig. 5(b) is a special case of the middle segment in Fig. 5(a), which can then be solved accordingly.

3.4 Correctness

As formalized in the HOTL theory, the conversion from footprint to miss ratio by Eq. 2 is not always correct. The correct miss ratio is given by reuse distance, i.e. for cache size c, the proportion of reuse windows whose reuse distance $d > c$. The footprint conversion is correct if the *reuse window hypothesis* holds, which states: the footprint in all reuse windows has the same distribution as the footprint in all windows, for every window length [53].

As the correctness condition for the HOTL theory, the reuse window hypothesis is the condition of correctness for victim footprint when modeling split LRU.

The reuse window hypothesis may or may not hold in practice. The accuracy of footprint conversion has been tested in empirical studies for the miss ratio of CPU caches [53], storage caches [13, 48], and the partitioning of software cache called Memcached [21]. These studies show that footprint is accurate in most of these widely used benchmarks.

One known problem of the footprint conversion is program phases. The miss ratio prediction can be misguided when taking the average footprint across phases with radically different locality. Drudi gave an example trace that has distinct phases, for which the average footprint gives the incorrect miss ratio on some cache sizes [13]. A solution is to detect locality phases (e.g. using reuse distance [37]) and then use the footprint analysis in each phase. The interaction between program phases in a co-run group depends on the alignment of co-run executions. This type of performance variation has been modeled by Sandberg et al. [34]

In practice, online footprint analysis and composition may be necessary to analyze cache sharing. Since victim footprint is mathematically derived from footprint, they have identical measurement costs, which can be made very low by sampling (0.1 second overhead per program as reported in [46, 53]). The remaining question is whether the victim footprint can accurately model cache exclusivity. This section gives the theoretical answer. In the evaluation section, we will measure the accuracy of the victim footprint on real machines.

4 EVALUATION

The VFP theory makes the idealistic assumption of fully-associative LRU cache, so it is vital to validate its real relevance. This section presents a lengthy evaluation with 5 experiments: 1) the accuracy of VFP prediction of exclusive cache sharing, 2) comparisons between VFP and heuristics, 3) symbiotic scheduling for the exclusive cache, 4) main causes of VFP prediction errors, and 5) the effect of non-LRU replacement policy. All experiments use actual machines. There is no simulation.

4.1 Methodology

An unusual aspect of the experimental design is shown in Table 1, where VFP is measured on an old model Intel machine (first release 2013) and then used to predict performance on three other machines, two models of AMD (server and high-end desktop, both 2011) and a recent model from Intel (released mid-2016), all with different processor architectures and cache hierarchies. The AMD cache is exclusive, while the Intel cache (on a Broadwell processor) is inclusive. The cross-machine prediction demonstrates the generality and machine independence of the victim footprint.

Table 1. Machines used: one VFP is used to predict for all machine types

Intention	Profiling	2 benchmark co-run	3/4 benchmark co-run	CAT prediction
Vendor	Intel	AMD	AMD	Intel
Model	i7-4770	Opteron 4226	FX-8120	E5-2630 V4
OS	Fedora 17	Ubuntu15.04	Ubuntu14.04	Ubuntu 16.04
GCC	4.7.2	5.1.0	4.8.4	5.4.0
L2	256KB	2M	2M	256KB
L3	8M	8M	8M	25MB

Implementation. VFP is derived from footprint. To measure the footprint, we profile an execution trace using the binary rewriter Pin [27] and collect three histograms: the inter-reference time of all reuses and the time of first and last access of all data blocks. The three histograms are then used to compute the footprint using the Xiang formula [52]. The measurement is identical to the one used by HOTL and has similar costs, i.e. on average 20 times slowdown [53]. Several techniques can measure the footprint in near real time using sampling [22, 46, 53]. We do not use sampling since it complicates the experimental setup, and the results would have depended on sampling parameters.

Xiang et al. showed that cache conflicts could be estimated using the Smith formula [20, 29, 38] but observed that the effect is negligible on modern large, highly associative CPU caches (for the benchmarks we describe next) [53]. In all experiments, we use the prediction for fully associative cache and ignore set-associativity in all prediction calculations.

Benchmarks. The evaluation uses SPEC CPU2006 (version 1.1) benchmarks [40], which is as far as we know the benchmark suite that has the most diverse representation of the cache behavior in sequential programs. We do not use *dealIII*, because it cannot be successfully compiled. The remaining 28 programs are all used. The same input, *reference*, is used in both profiling and testing. Postmortem analysis avoids the inaccuracy of sampling. However, it takes the average across the whole trace and therefore does not distinguishing phases. For programs with strong phase behavior, periodic sampling can predict the miss ratio more accurately [46].

We classify the benchmarks into 3 categories of roughly equal size based on their *memory intensity* measured on the Opteron machine. The memory intensity is measured as the L3 miss ratio (misses per load/store). The miss ratios are 2.5% to 8.0% for 9 memory intensive benchmarks, 0.05% to 2.5% for 10 memory moderate benchmarks and below 0.005% for 9 memory light benchmarks. The definition of memory intensity follows Zhuravlev et al. who defined cache intensity as L2 misses per million instructions [59].[2] Table 2 shows the three memory-intensity categories.

Exhaustive Co-run Testing. For complete testing and to avoid any bias due to sampling, we test all 2-program (duet) runs, which is numbered $\binom{28}{2} = 378$. For 3- and 4-program (triplet and quartet) runs, we select 4, 4 and 2 benchmarks respectively from the three categories and in each category select the programs in alphabetic order. The selection favors intensive and moderate programs, so cache is a more significant factor in performance. Table 2 shows the names of these 10 programs in bold font. The total numbers of triplet and quartet tests are $\binom{10}{3} = 120$ and $\binom{10}{4} = 210$ respectively.

Co-run benchmarks are bound to cores that do not share the L2 cache, more specifically, cores #0 and #2 in 2 benchmark co-runs, and #4, #6 in 3 and 4 benchmark co-runs.

[2] We have ranked the programs in two other ways. The classification is exactly the same when using miss rate (misses per second) and differs by 2 programs when using L3 MPKI.

Table 2. 28 SPEC CPU 2006 benchmarks grouped by memory intensity (L3 miss ratio). 10 programs (name in **bold**) are used in 3- and 4-program tests.

Intensive (9 Programs)				
bwaves	**GemsFDTD**	**lbm**	**leslie3d**	libquantum
mcf	milc	soplex	omnetpp	
Moderate (10)				
astar	**cactusADM**	**calculix**	**gcc**	gobmk
sjeng	sphinx	wrf	xalancbmk	zeusmp
Light (9)				
bzip2	**gamess**	gromacs	h264ref	hmmer
namd	perlbench	poveray	tonto	

The benchmarks have different running times. In co-run tests, we run each program repeatedly for 20 minutes and measure the performance events when it overlaps with the partner programs. The method has been used in previous work [26, 39, 53, 56]. It produces stable results and avoids the problem of run-to-run performance variation. Such variability has been shown by Sandberg et al.[34]

Metrics and Measurement. The miss ratio is defined as the number of LLC cache misses divided by the number of memory accesses(unfiltered by L1/L2 caches). LLC cache misses and memory accesses are measured by the hardware counters as reported by the lightweight performance monitoring tool `likwid` [44]. The following events are counted. *They include both on-demand accesses and prefetches*. We do *not* disable hardware prefetcher in experiments, in order to evaluate the model in a fully realistic environment.

Event	Event ID	Mask
`DATA_CACHE_ACCESSES`	0x040	N/A
`UNC_L3_CACHE_MISS_ALL`	0x4E1	0xF7

The miss ratio is computed as follows[12]:

$$MissRatio = \frac{\text{UNC\_L3\_CACHE\_MISS\_ALL}}{\text{DATA\_CACHE\_ACCESSES}}.$$

VFP predicts both individual and group miss ratios in the shared cache. We test the accuracy of only the total miss ratio of a co-run group (UNC_L3_CACHE_MISS_ALL), which is per socket and includes the events of all the cores of the socket.[3]

4.2 VFP Prediction Accuracy

This section evaluates co-run groups by their intensity categories. Each category is labeled `mInMkL` and includes all co-run groups with *m* memory intensive, *n* moderate and *k* light programs in Table 2. The lexicographical order of the category label coincides with the order of intensity. Of duet-run categories for example, `0I0M2L` is least intense, and `2I0M0L` most intense. For lack of space, only duet and triplet tests are shown. Quartet results will be shown in the next section.

Fig. 6 shows measured and predicted miss ratios and the absolute errors of prediction for duet groups (upper graph) and triplet groups (lower graph). The graphs divide 378 duet groups into 6 categories and 120 triplet groups into 9 categories and show the average in each category. The average miss ratio increases from 0.029% to 4.90% in duet categories and from 0.24% to 4.34% in triplet categories. The increase is monotonic in the order of group intensity.

[3]Per-program prediction can be tested when per-core counters are available, as it has been done in previous work [53].

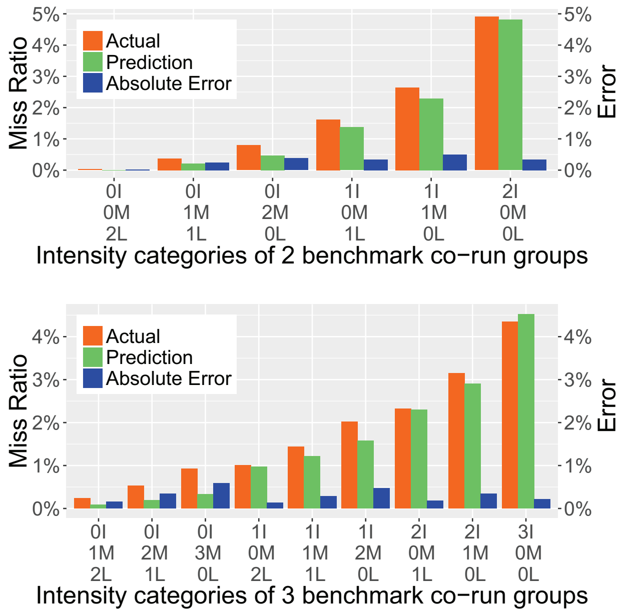


Fig. 6. Actual and predicted miss ratios and the absolute error for each category ordered by intensity. *The absolute error is lower than 0.5% in all categories, except 0.58% at* `0I3M0L`.

In all categories except the most intense `3I0M0L`, the average predicted miss ratio is lower than the actual, showing that VFP under predicts more than it over predicts. VFP under predicts in 281 out of 378 duet groups and over predicts in the remaining 97 groups. It under predicts in 100 out of 120 triplet groups and over predicts in the rest. The tendency does not depend on the miss ratio. The average miss ratio of the under predicted groups is lower in duet groups (1.38% vs. 2.37%) but higher in triplet groups (2.27% vs. 1.85%).

Prefetching is a reason for the tendency of VFP to under predict. A prefetcher estimates likely misses and may prefetch unnecessarily. A second reason is the effect of phases, which we will discuss in Section 4.5.

The absolute error of VFP prediction is lower than 0.50% in all categories, except for 0.58% at `0I3M0L`. A relative error is not as important in practice, i.e. when a miss ratio is very small. Still, to fully evaluate the theory, we should examine the relative error.

Fig. 7 shows the average relative error in all duet and triplet categories ordered (together in one graph) by intensity. The first 6 least intense categories, i.e. those with 0 intensive programs, have the highest relative errors, ranging between 44.7% to 61.5%. Their miss ratios are low, on average 0.03% and 0.55% for duet and triplet groups respectively, so cache performance is unlikely the performance bottleneck, and these high relative errors are unlikely a problem in practice.

In the 4 most intense categories, the relative error is within 10% except for 10.7% in *2I2M0L*. The most intense category has the lowest relative error of 4.7%, corresponding to an absolute error of 0.2% (out of 4.34% shown in Figure 6).

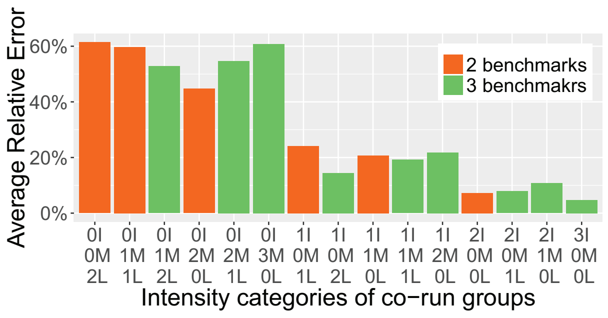


Fig. 7. Relative prediction errors of all categories, ordered by memory intensity.

4.3 Theory vs. Heuristics

As discussed at the outset, the exclusivity has two effects. The first is deduplication which increases the effective size of the shared cache. The second is interaction of the "victims" in the shared cache. Both effects are magnified by large private L2s on AMD, 2MB L2 vs. 8MB L3.[4]

We compare VFP theory with HOTL theory (exclusivity unaware) and three heuristics. All five methods are analytical and can compute the cache performance for all program groups and all shared cache sizes without exhaustive testing.

(1) HOTL, which models sharing of a single LLC of the combined size including all private caches, i.e. 12MB, 14MB, 16MB in 2, 3 and 4 benchmark co-runs. HOTL models the deduplication effect but not victim-cache sharing.
(2) Even, which assumes each co-run program uses an equal partition of the combined size. HOTL is used to compute the miss ratio. Chen et al. developed this heuristic for GPU caches [6].
(3) Proportional(MissRatio) and Proportional(MissRate), which are similar to Even but the cache occupancy is proportional to its solo-run miss ratio (misses per hundred access) and miss rate (misses per second) respectively.

Fig. 8 compares the cumulative distribution (CDF) of absolute prediction errors for all duet, triplet and quartet tests, one in each graph. From the results, we draw the following conclusions.

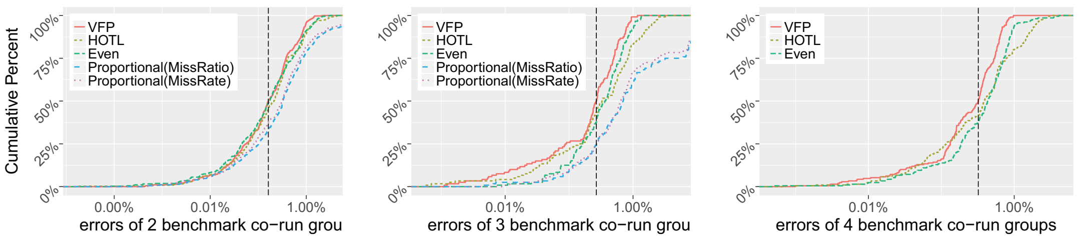


Fig. 8. The cumulative distribution of absolute prediction errors of 378 duet groups (left), 120 triplet groups (middle) and 210 quartet groups (right). The dotted lines show the median errors (50%) of VFP: 0.16%, 0.27% and 0.32% respectively.

[4]L1,L2 are also exclusive, so the effective L2 size is 2MB + 64KB.

HOTL is clearly worse than VFP, which shows that *an exclusive cache hierarchy is not equivalent to a single, larger LLC*. The average errors of HOTL are high and grow rapidly, 37%, 76% and 85% higher than those of VFP, for duet, triplet, and quartet tests. The increasing errors are due to the increasing combined cache size used by HOTL (because of deduplication). It is clear that victim cache sharing differs from cache sharing.

Even partitioning is an effective heuristic. It is highly accurate in our tests. The average errors are 0.36%, 0.43% and 0.53%, and the median errors are 0.17%, 0.37% and 0.43%. Still, VFP reduces them by 17%, 23%, 38%, 6%, 27%, and 26%. Even partitioning is a simple heuristic, although it still needs the footprint.

VFP is most accurate, and its advantage increases with the group size. This can be seen visually in Fig. 8, where VFP is the curve that is highest, i.e. left-most, and its gap with other curves increases with the group size. The average errors are 0.30%, 0.33% and 0.33%, and the medians are smaller 0.16%, 0.27% and 0.32%.

Proportional partitioning is incorrect when understanding cache sharing. The two heuristics are worst performing, the one based on miss rate marginally better than the other based on miss ratio. Beckmann et al. observed that the access to shared cache by a program is effectively random due to the filtering by private cache. Models of cache have been developed with different assumptions of randomness ([2, 3, 18, 38, 47] as discussed in [11]). Our results suggest such effect but no obvious solution: even partition works well, but proportional partitioning does not.

4.4 Program Symbiosis in Victim Cache

The term "symbiotic scheduling" was coined by Snavely and Tullsen [39]. We use the methodology developed by Wang et al. to evaluate the quality of symbiosis [46]. In this test, a symbiotic scheduler evenly divides a set of $2p$ programs into 2 co-run groups to run on a p-core processor. In each group, each program i runs repeatedly, long enough to obtain a stable average of co-run time, $t_{group}(i)$.

The performance is measured by the aggregate slowdown defined as follows. It is the sum of the slowdown of each program i, which is the extra time due to co-run divided by the sequential running time $t_{solo}(i)$. The average of this aggregate, $\frac{s}{2p}$, is equivalent to Average Normalized Turnaround Time (ANTT) [16].

$$s = \sum_{i=1}^{2p} \frac{t_{group}(i) - t_{solo}(i)}{t_{solo}(i)} \tag{15}$$

We compare the following four techniques:

- OPT : Test the running time of all $\binom{2p}{p}$ group assignments and select the best.
- VFP : Compute the miss ratio of all group assignments using VFP and choose the best.
- HOTL by Wang et al. [46]: Similar to VFP but using the HOTL model.
- DI (Distributed Intensity) by Zhuravlev et al. [59]: Sort programs by their solo-run miss rates (misses per 1 million instructions) and assign programs round-robin into groups. DI was shown to be effective and robust on both Intel and AMD processors. The benefits on Intel processors have been independently confirmed [46].

We first examine $p = 3$ and test all $\binom{10}{6} = 210$ problems for triplet symbiosis. Each problem has $\frac{\binom{6}{3}}{2} = 10$ ways to divide 6 programs into two groups. The left graph of Fig. 9 shows the cumulative distribution (shown on y-axis) of the 210 aggregate slowdowns (shown on x-axis) obtained by each technique.

The median of the 210 aggregate slowdowns is 129.2% for VFP, shown by the vertical line in Fig. 9. The median is 126.8% for OPT, 133.3% for HOTL, and 138.1% for DI. If we use these median to measure how the three techniques compare to OPT, DI is 9% worse than OPT, and HOTL

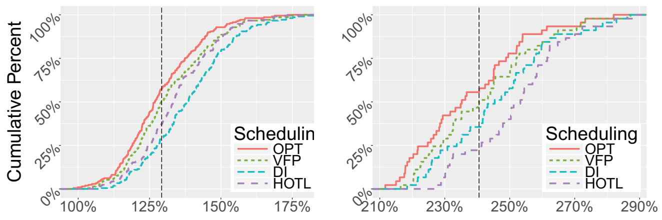


Fig. 9. The cumulative distribution of aggregate slowdowns of (left) 210 tests of triplet symbiosis and (right) 45 tests of quartet symbiosis by three techniques and optimal. The dotted lines show the median slowdowns (50%) of VFP: 129% and 240% respectively.

and VFP are 5% and 1.9% worse. Hence, HOTL closes half of the gap between DI and OPT, and VFP closes more than half of the remaining gap between VFP and OPT. The numbers of optimal symbiosis out of 210 problems are 12 for DI, 24 for HOTL, and 50 for VFP. HOTL is optimal twice as often as DI, and VFP is optimal more than twice as often as HOTL.

VFP outperforms HOTL for 60% of the 210 problems and DI for 74% and ties with HOTL and DI for 14% and 12% of the problems respectively.

4-Program Co-runs. We also examine $p = 4$ and test all $\binom{10}{8} = 4$ problems for quartet symbiosis. Each problem has $\frac{\binom{8}{4}}{2} = 35$ ways to divide 8 programs into two groups. The fact that an exclusive cache hierarchy cannot be modeled as an inclusive cache hierarchy is shown most clearly by the results of quartet symbiosis in Fig. 9. HOTL, which was the second best in triplet symbiosis, becomes the worst in quartet symbiosis, because it fails to model the filtering effect. DI, however, performs well when scheduling programs under high cache contention.

Private Cache vs. Memory Bandwidth vs. Shared Cache. There is much greater contention when the group size increases from 3 to 4, evident by the greater scale in the x-axis of the right-hand graph of Fig. 9. The aggregate slowdowns are nearly doubled when triplets become quartets. This suggests a saturation of memory bandwidth and controller resources, which Zhuravlev et al. showed as having a greater effect than cache sharing [59]. DI was developed as a heuristic solution (to balance the "intensity"). The cache models, VFP and HOTL in this case, predict the miss counts and enable optimization. The quartet results show that while the heuristic of DI is especially effective at high contention, model-based optimization still improves over the heuristic. But to do so, the model must be exclusivity aware.

Heuristic vs Theory. For co-run grouping for throughput, DI is effective and robust. However, DI does not predict the co-run miss ratio, i.e. not applicable in the evaluation in Section 4.3 especially Fig. 8. In comparison, VFP predicts individual co-run miss ratios, which is more useful for providing QoS guarantee beyond finding the best symbiosis. In addition, it is unclear how DI should be applied in multi-level sharing as described in Section 3.3.2, where the "intensity" of co-run programs in LLC depends on how the higher level is shared. In comparison, VFP predicts the effect of multi-level cache sharing.

The overhead of prediction is small, $O(P * loglogN)$ for exclusive cache, where P is the number of co-run programs and N is the length of the trace. In our experiments, it takes 0.67 seconds for all 378 duet tests in exclusive cache and 0.22 seconds for all 120 triplet tests.

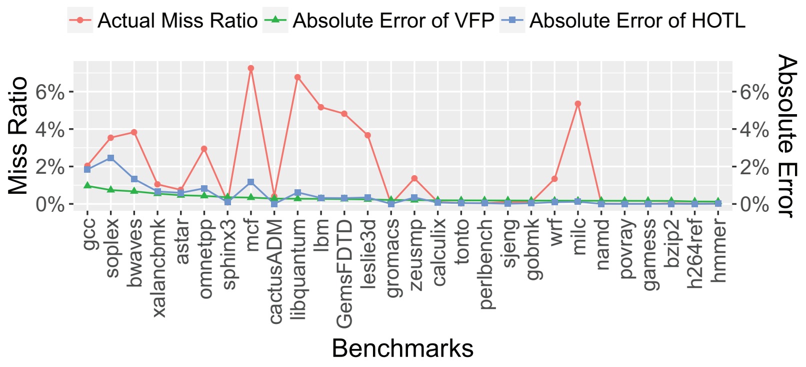


Fig. 10. Average co-run miss ratio, VFP co-run prediction error, and HOTL solo-run prediction error for each program. The VFP co-run prediction error correlates with the HOTL solo-run prediction error but not with the co-run miss ratio.

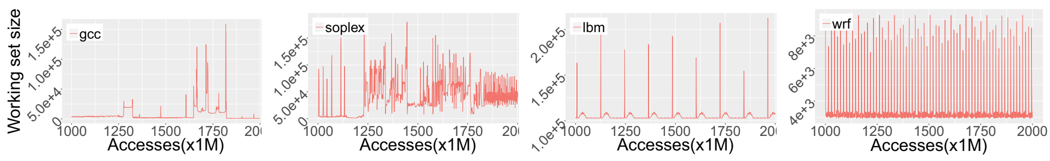


Fig. 11. Phase variation in working-set size of four programs in the period between 1 and 2 billion-th accesses. The regularity of phase variation correlates with the VFP prediction accuracy shown in Figure 10.

4.5 VFP Error Analysis

The error of VFP is small, 0.30% 0.33% and 0.33% on average across three group sizes. The primary source of this error comes from the HOTL theory. We demonstrate this using the results of duet tests.

Figure 10 shows the *per-program* the error of VFP prediction for each of the 28 programs in all its 27 duet groups. For ease of viewing, Figure 10 connects discrete data points into curves. The 28 programs are sorted in the descending order by the absolute error of VFP, starting from the least accurate programs which are the focus of this analysis. In addition to VFP errors, the figure shows the average miss ratio of each program in its duet groups and the error of the prediction by HOTL for a *solo* program execution.

The VFP error correlates more with the solo-run HOTL error than with the co-run miss ratio. The linear correlation co-efficient (r) is 88.83% between VFP co-run error and HOTL solo-run error.

The main reason for HOTL errors is program phases. HOTL measures the average working-set size (Eq. 1). When a program has different phases, the overall average differs from per-phase average. This has been shown by a contrived example by Drudi [13] and empirical results by Wang et al. on SPEC benchmarks especially *mcf* [46]. However, our results indicate that the effect of phases is more subtle — the error of HOTL (and VFP) is caused not so much by phases but by the irregularity of phases.

To visualize phase behavior, we plot the working set size (WSS) in every 1 million accesses in the period between 1 billion-th and 2 billion-th instructions. Fig. 11 shows the WSS in four benchmarks. The four graphs are ordered by the per-program (in)accuracy as in Fig. 10. The first two, *gcc* and *soplex*, have highest errors, 1.84% and 2.46%. It is visually striking that the first two graphs are highly irregular. In comparison, the next two programs, *lbm* and *wrf*, have large but regular phase variation. Their per-program errors are much smaller, 0.32% and 0.09%. Hence, phase behavior itself does not necessarily reduce VFP accuracy.

4.6 Prediction of Intel CAT

We evaluate the prediction on an Intel machine with an E5-2630V4 (codenamed “Broadwell-EP”) processor and 64GB memory. It has a unique cache hierarchy, shown in Table 1. Most distinct is the Intel Cache Allocation Technology [19], which we use to run a program with 20 cache sizes, from 1.25MB to 25MB at 1.25MB increments.

We use events `MEM_UOPS_RETIRED:ALL_LOADS`, `MEM_UOPS_RETIRED:ALL_STORES` and `OFFCORE_RESPONSE_1:ANY_REQUEST:L3_MISS`. The first two counts the number of accesses of loads and stores respectively, the last one counts the number of L3 misses. All of them count both on-demand accesses and prefetches. Note that the new measurement is entirely for validation and not for prediction, which uses the same footprint as used in previous experiments.

Figure 12 shows the measured and predicted miss ratios (using HOTL) for 28 SPEC CPU 2006 benchmark programs for all 20 cache sizes. The logarithmic scale is used since the 560 miss ratios span many orders of magnitude. The average prediction error is 0.52% in (absolute) miss ratio. The highest happens in *gcc* with 1.25MB cache. The measured miss ratio is 5.46%, while the prediction is 0.77%. For a single program across all 20 cache sizes, the highest average error (2.20%) happens at *libquantum*, where the measured miss ratios are from 4.2% to 5.4%, and the prediction is from 7.39% to 7.40%. The only other program with 2% or higher average error is *milc*, which is just 2.0%.

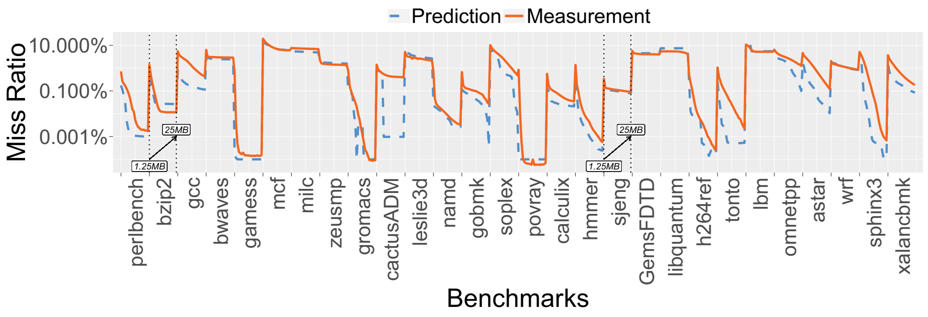


Fig. 12. Predicted and measured miss ratios of 28 programs each on 20 cache sizes from 1.25MB to 25MB, with 1.25MB increments, on an Intel Broadwell processor.

Modern Intel processors manage their caches using an adaptive replacement policy similar to Re-Reference Interval Prediction (RRIP) [25] and Dynamic Insertion Policy (DIP) [33]. Both RRIP and DIP can detect the effect of streaming accesses and modify LRU replacement to cache none or only a part of the streaming data.

For similar cache sizes, 7.5MB and 8.75MB on Intel and 8MB on AMD, the miss ratios on two machines are similar for most programs, but there are several differences, and they suggest RRIP/DIP like benefits. For *libquantum*, HOTL predicts 7.4%, while the actual is 6.8% on AMD but between 4.2% and 5.4% on Intel. HOTL over predicts *bzip2* and *leslie3d* on Intel but not on AMD, likely because they have streaming accesses, and caching is improved by an adaptive insertion policy. For other programs, the Intel results are similar to AMD. On both machines, the largest error happens on *gcc* because of the irregular phase behavior as explained before. For *cactusADM*, HOTL predicts relatively well on the first 5 cache sizes but around 10^{-5} miss ratio at 7.5MB or higher, while the actual miss ratio is around 0.4%, on both Intel and AMD.

As Figure 12 shows, the miss ratios of 560 SPEC CPU 2006 executions span multiple orders of magnitude by different programs. Overall, the HOTL prediction tracks the reality closely for most programs in most cache sizes.

4.7 Review of Results

The section has presented five experiments. The first evaluated VFP in modeling the shared exclusive cache, including 378 duet, 120 triplet, and 210 quartet tests on two types of AMD processors. It requires 28 sequential runs (to obtain all predictions) and 708 parallel runs (for validation). In all test categories, the absolute error in miss ratio is within 0.6% (or 0.5% except for one category). The absolute error does not increase for programs with a higher miss ratio, so the relative error decreases. In the most memory intensive category, the relative error is within 5%, corresponding to an absolute error of 0.2%.

The second experiment compared VFP with HOTL and three heuristics. It tested 4 additional modeling methods for each of the 708 co-run tests from the first experiment. Ranked by accuracy, we have VFP > the best heuristic > HOTL > and the other two heuristics. The theory considering exclusivity is the best. The average prediction error of HOTL (which models deduplication but not filtering) is near twice that of VFP in triplet and quartet tests. Interestingly, the best heuristic also outperforms HOTL, although other heuristics perform poorly.

The third experiment is exclusivity-aware program co-location for shared cache. For 210 problems of triplet and 45 problems of quartet symbiosis, it compared VFP, HOTL, a heuristic model DI, and optimal which required 2,100 triplet runs and 1,575 quartet runs for exhaustive testing. HOTL surpasses over DI at low contention, and DI over HOTL at high contention. VFP improves over both, closing near 80% of the gap between the prior work and the optimal. Furthermore, VFP predicts individual miss ratios (and hence can ensure QoS), but DI cannot. In both cases, exclusivity modeling is necessary.

The fourth experiment showed that the error of VFP largely comes from HOTL. It showed high correlation between VFP prediction errors in 708 co-run tests and HOTL prediction errors in 28 sequential tests. The last experiment was a first study of HOTL on Intel CAT and tested 20 cache sizes each for 28 programs. The main sources of error in HOTL are two: the weakness of average program statistics and the lack of support for non-LRU replacement policies. However, among the 28 programs tested, the limitations impact just 3 or 4 programs each, and most of these errors are marginal. Furthermore, if HOTL becomes more accurate, VFP will be more accurate. The HOTL problems are orthogonal to exclusivity modeling.

5 RELATED WORK

We focus on analytical techniques which predict performance for all cache sizes and shared cache optimization that uses analytical models.

Analytical Models of Cache Sharing. The working-set theory by Denning was the first to convert between the working-set size and the number of page faults [8–10]. The recent theory by Xiang et al. defines a new type of working-set size called *footprint* and shows the conversion between footprint, miss ratio curve and reuse distance [53], as reviewed in Section 2. In both theories, the mathematical properties, monotonicity and concavity, are important for the metrics conversion. Neither theory has considered cache exclusivity.

Thiebaut and Stone [43] coined the term *cache footprint*. Falsafi and Wood [17] redefined the *footprint* as the set of unique data blocks a program accesses. Both studies computed what is essentially the single-length window footprint, where the length is the CPU scheduling quantum [43].

On multicore processors, programs interact in shared cache at all times. The problem of all-window footprint was computationally difficult, because the number of windows is quadratic to the length of the trace. Approximate solutions were pioneered by Suh et al. [41] for choosing the best scheduling quantum, Chandra et al. [5] and Eklove and Hagersten [14, 15] for multicore cache sharing, and Shen et al. for fast reuse distance analysis [36]. The equations in previous work were not completely constrained, so the solution was not unique and depended on modeling assumptions. Xiang et al. defined the footprint which is precise and deterministic and gave a linear-time algorithm to measure it [52].

While previous theories, e.g. HOTL, can model deduplication, victim footprint is the first theory to fully model cache exclusivity. The original footprint is now a special case of victim footprint, and the composition of victim footprint gives the general solution for any cache hierarchy, inclusive or exclusive (see Section 3.3.2). The analysis is composition invariant (see Section 3.3.1). VFP the first model of victim cache to have this property.

Online Locality Measurement. Initial theories were complex and too expensive to use in real-time solutions. Xie and Loh [54] noted that the model by Chandra et al. (mentioned earlier) “is fairly involved; the large number of complex statistical computations would be very difficult to directly implement in hardware.” This model was not used in the comparison study of Zhuravlev et al. [59], because it was not “computationally fast enough to be used in the robust scheduling algorithm.”

Reuse distance is accurate and not affected by phase behavior discussed in Section 4.5. It has been used to model cache sharing for independent [35] and multi-thread programs [35, 49].

For virtual memory management, reuse distance can be measured in real time in software [55, 58]. On CPU cache, Xiang et al. showed a model that converts the footprint into reuse distance, so it no longer needs to measure the reuse distance [53]. It takes 0.1 second per program to measure footprint with adaptive bursty sampling [46]. The efficiency has been further improved in SHARDS [45] and AET [22]. SHARDS measures reuse distance directly, based on reuse distance sampling [35, 57]. Sampling can also be supported in hardware as in RapidMRC and StatStack [15, 42]. AET analysis used random sampling pioneered by StatStack and found it to provide highest stability and accuracy. A recent CounterStack algorithm asymptotically reduced the space complexity of reuse distance measurement [48]. Efficient analysis of reuse distance can be used by victim footprint by first converting it to footprint using the HOTL theory [53]. With victim footprint, all these efficient techniques can potentially be used in online modeling of exclusivity, as well as other cache designs mentioned in Section 3.3.3, with little additional cost.

Contention-aware Scheduling. Contention-aware scheduling may also be called program symbiosis and job co-location. For shared cache, it can improve the total throughput or ensure quality of service (QoS). DI is a simple and effective heuristic [59]. Its performance in practice has been independently verified for inclusive caches ([46]) and exclusive caches (Section 4.4). However, it does not predict the co-run miss ratio and as a result cannot be used for predictive optimization or miss-ratio-based QoS. In addition, its measurement is machine dependent. A related system, Paragon, is based on machine learning and has similar limitations [7]. Another method, Bubble-up, predicts the co-run performance (not just miss ratio) [30]. Still, its measurement is machine dependent (and probe dependent). A fourth technique, PORPLE, is a tool developed for GPUs (for data “symbiosis” rather than task) [6]. It is predictive, its measurement is machine independent, but it assumes even partition of shared cache. Section 4.3 has adapted this model to predict co-run miss ratio and found its assumption fairly effective in practice.

These techniques do not consider the effect of exclusive cache hierarchy. Victim footprint is the first model that is exclusivity aware, and as a result, it is more accurate in both prediction and optimization than the techniques we have tested.

6 SUMMARY

This paper has presented the theory and techniques to model the performance of exclusive cache hierarchies. The formalism includes the split LRU stack, victim footprint and victim cache fill time, and their properties especially the uniqueness theorem. The techniques are composable analysis and portable prediction for any program groups sharing any types of AMD exclusive caches, Intel cache allocation technology (CAT), and IBM transient loads.

In evaluation, the victim footprint is measured in a single set of 28 sequential executions and used to predict the miss ratio in 4383 parallel executions on 2 models of AMD machines and 588 sequential executions on an Intel machine. The results show that VFP outperforms the previous theory HOTL and all heuristics solutions that were tested. For the AMD exclusive cache hierarchy, the VFP prediction has on average less than 0.2% absolute errors and 5% relative errors when predicting high miss ratios and when used for shared-cache program symbiosis, removes near 80% of the gap between the prior work and the optimal. Hence, we conclude that the new theory is practical on real systems, and exclusivity modeling is necessary for its accuracy and benefit.

REFERENCES

[1] AMD. Software optimization guide for amd family 15h processors, January 2014.

[2] N. Beckmann and D. Sanchez. Modeling cache performance beyond lru. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 225–236. IEEE, 2016.

[3] E. Berg, H. Zeffer, and E. Hagersten. A statistical multiprocessor cache model. In *Proceedings of ISPASS*, pages 89–99, 2006.

[4] J. Brock, C. Ye, C. Ding, Y. Li, X. Wang, and Y. Luo. Optimal cache partition-sharing. In *Proceedings of ICPP*, 2015.

[5] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of HPCA*, pages 340–351, 2005.

[6] G. Chen, B. Wu, D. Li, and X. Shen. Porple: An extensible optimizer for portable data placement on GPU. In *Proceedings of MICRO*, 2014.

[7] C. Delimitrou and C. Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Proceedings of ASPLOS*, pages 77–88, 2013.

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

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

[10] P. J. Denning and D. R. Slutz. Generalized working sets for segment reference strings. *Communications of the ACM*, 21(9):750–759, 1978.

[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] P. J. Drongowski and B. D. Center. Basic performance measurements for amd athlon64, amd opteron and amd phenom processors. 2008.

[13] Z. Drudi. A streaming algorithms approach to approximating hit rate curves. Technical Report MS Thesis, The University Of British Columbia, 2014.

[14] D. Eklov, D. Black-Schaffer, and E. Hagersten. Fast modeling of shared caches in multicore systems. In *Proceedings of HiPEAC*, pages 147–157, 2011. *Best paper*.

[15] D. Eklov and E. Hagersten. StatStack: Efficient modeling of LRU caches. In *Proceedings of ISPASS*, pages 55–65, 2010.

[16] S. Eyerman and L. Eeckhout. Probabilistic job symbiosis modeling for SMT processor scheduling. In *Proceedings of ASPLOS*, pages 91–102, 2010.

[17] B. Falsafi and D. A. Wood. Modeling cost/performance of a parallel computer simulator. *ACM Transactions on Modeling and Computer Simulation*, 7(1):104–130, 1997.

[18] A. Fedorova, M. Seltzer, and M. D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proceedings of PACT*, 2007.

[19] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer. Cache qos: From concept to reality in the intel xeon processor e5-2600 v3 product family. In *HPCA*, pages 657–668, March 2016.

[20] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12):1612–1630, 1989.

[21] 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.

[22] X. Hu, X. Wang, L. Zhou, Y. Luo, C. Ding, and Z. Wang. Kinetic modeling of data eviction in cache. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016.

[23] IBM. Power8 processor user's manual for the single-chip module, January 2015.

[24] Intel. *Improving Real-Time Performance by Utilizing Cache Allocation Technology*. Intel, April 2015.

[25] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer. High performance cache replacement using re-reference interval prediction (rrip). In *ACM SIGARCH Computer Architecture News*, volume 38, pages 60–71. ACM, 2010.

[26] Y. Jiang, K. Tian, and X. Shen. Combining locality analysis with online proactive job co-scheduling in chip multiprocessors. In *Proceedings of HiPEAC*, pages 201–215, 2010.

[27] C.-K. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of PLDI*, pages 190–200, 2005.

[28] H. Luo, P. Li, and C. Ding. Thread data sharing in cache: Theory and measurement. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, pages 103–115, New York, NY, USA, 2017. ACM.

[29] G. Marin and J. Mellor-Crummey. Cross architecture performance predictions for scientific applications using parameterized models. In *Proceedings of SIGMETRICS*, pages 2–13, 2004.

[30] 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.

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

[32] C. MIKE. A new x86 core architecture for the next generation of computing. In *Hot Chips*, 2016.

[33] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. S. Emer. Adaptive insertion policies for high performance caching. In *Proceedings of ISCA*, pages 381–391, 2007.

[34] 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.

[35] 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.

[36] X. Shen, J. Shaw, B. Meeker, and C. Ding. Locality approximation using time. In *Proceedings of POPL*, pages 55–61, 2007.

[37] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *Proceedings of ASPLOS*, pages 165–176, 2004.

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

[39] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Proceedings of ASPLOS*, pages 234–244, 2000.

[40] Spec cpu benchmarks. http://www.spec.org/benchmarks.html#cpu.

[41] G. E. Suh, S. Devadas, and L. Rudolph. Analytical cache models with applications to cache partitioning. In *Proceedings of ICS*, pages 1–12, 2001.

[42] 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 ASPLOS*, pages 121–132, 2009.

[43] D. Thiébaut and H. S. Stone. Footprints in the cache. *ACM Transactions on Computer Systems*, 5(4):305–329, 1987.

[44] J. Treibig, G. Hager, and G. Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, pages 207–216. IEEE, 2010.

[45] C. A. Waldspurger, N. Park, A. Garthwaite, and I. Ahmad. Efficient mrc construction with shards. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 95–110, 2015.

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

[47] R. West, P. Zaroo, C. A. Waldspurger, and X. Zhang. Online cache modeling for commodity multicore processors. *Operating Systems Review*, 44(4):19–29, 2010.

[48] 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.

[49] 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.

[50] M.-J. Wu, M. Zhao, and D. Yeung. Studying multicore processor scaling via reuse distance analysis. In *Proceedings of ISCA*, pages 499–510, 2013.

[51] X. Xiang, B. Bao, T. Bai, C. Ding, and T. M. Chilimbi. All-window profiling and composable models of cache sharing. In *Proceedings of PPoPP*, pages 91–102, 2011.

[52] 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.

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

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

[55] T. Yang, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. CRAMM: Virtual memory support for garbage-collected applications. In *Proceedings of OSDI*, pages 103–116, 2006.

[56] 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.

[57] Y. Zhong and W. Chang. Sampling-based program locality approximation. In *Proceedings of ISMM*, pages 91–100, 2008.

[58] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. In *Proceedings of ASPLOS*, pages 177–188, 2004.

[59] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of ASPLOS*, pages 129–142, 2010.