Cache Exclusivity and Sharing: Theory and Optimization

CHENCHENG YE, Huazhong University of Science and Technology and the University of Rochester
CHEN DING, HAO LUO, JACOB BROCK, and DONG CHEN, University of Rochester
HAI JIN, Huazhong University of Science and Technology

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 a large private cache while still benefitting 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 article presents a new metric called the *victim footprint* (VFP). 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 work evaluates the VFP by using it to analyze cache sharing by parallel mixes of sequential programs, comparing the accuracy of the theory to hardware counter results, and measuring the benefit of exclusivity-aware analysis and optimization.

CCS Concepts: • **Computing methodologies** → **Modeling methodologies**; • **Hardware** → *Dynamic memory*;

Additional Key Words and Phrases: Cache analysis, cache sharing, exclusive cache

ACM Reference format:

Chencheng Ye, Chen Ding, Hao Luo, Jacob Brock, Dong Chen, and Hai Jin. 2017. Cache Exclusivity and Sharing: Theory and Optimization. *ACM Trans. Archit. Code Optim.* 14, 4, Article 34 (November 2017), 26 pages. <https://doi.org/10.1145/3134437>

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 [1, 14].

This work was done during C. Ye’s visit to the University of Rochester, funded in part by the Chinese Scholarship Council. He is a doctoral student from Huazhong University of Science and Technology. This is a new article, not an extension of a conference paper.

The work was supported by the National Science Foundation (contracts CCF-1717877, CCF-1629376, CNS-1319617), an IBM CAS Faculty Fellowship, the National High-Tech Research and Development Program of China (contract 2015AA015303), and the National Natural Science Foundation of China (contract 61433019). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding organizations.

Authors’ addresses: C. Ye and H. Jin, Service Computing Technology and System Lab, Cluster and Grid Computing Lab, Big Data Technology and System Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China; emails: yechencheng@gmail.com, hjin@hust.edu.cn; C. Ding, H. Luo, J. Brock, and D. Chen, Computer Science Department, University of Rochester, Rochester, New York, USA; emails: {cding, hluo, jbrock, dchen}@cs.rochester.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2017 ACM 1544-3566/2017/11-ART34 \$15.00

<https://doi.org/10.1145/3134437>

Exclusive cache differs from inclusive cache in two major aspects. First, there is no data duplication, which is a benefit because 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*.¹ The sharing of the victim cache is between the evicted data from multiple programs. We call the second effect *filtering*.

This article 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 the *victim footprint* (VFP) and shows how it can predict the performance of the program on a split LRU stack. With such prediction, it shows how to optimize the sharing of exclusive cache.

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

The new theory assumes a fully associative LRU cache. With this assumption, the composition and prediction techniques are entirely mathematical, allowing the presentation to be precise and complete. In fact, the analysis will be entirely shown in mathematical terms for single-level (Equation (13)) and multilevel (Equation (14)) cache 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 these parameters but using the same VFP. 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. A common strategy is direct measurement 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 portable, more general models have been developed. Early models were based on reuse distance [5, 47, 57], which was costly to measure. More recent models, particularly StatStack [16, 17] and higher-order theory of locality (HOTL) [60], use reuse time, which can be measured much faster. The new theory of VFP 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:

- *VFP formulation*, which defines split LRU (Section 3.1), VFP and victim cache fill time, and proves properties especially the uniqueness theorem (Section 3.2).
- *Composable analysis and portable prediction*, which uses VFP to model data sharing in AMD exclusive caches, Intel cache allocation technology (CAT), and IBM transient loads (Section 3.3).
- *Evaluation*, which tests the VFP on three models of AMD and Intel machines for performance prediction, optimization, and comparison with the previous theory and alternative heuristic-based solutions (Section 4).

VFP has at least four limitations. First, it models only the fully associative LRU cache, as mentioned earlier. Second, it models cache performance—that is, the number of accesses in main

¹ *Victim cache* was invented by Jouppi in 1990 as a small, fully associative cache [30]. The term is more recently used by AMD to refer to the L3 cache [1, 8, 14]. Similar uses were made by IBM [25], Intel [27], and academic researchers [36, 48]. In these specific cases, victim cache refers to large set-associative cache (256KB or larger).

Fig. 1. Illustration of the footprint and fill time functions defined in the HOTL theory (Equations (1) and (3)). The left line shows the cache size, and the right line shows the window length. The mappings are shown underneath (e.g., from x to c by footprint on the left and from c to x by fill time on the right).

memory, not running time. It is oblivious of the processor architecture or the effect of prefetching. Some memory accesses are more expensive than others. VFP predicts only the access count, not the access cost. Third, it uses postmortem analysis with the full execution trace of each program. In practice, it may be used online through sampling (shown by prior work discussed in Section 5). Finally, although the HOTL theory (Section 2) is recently extended to model data sharing [32], in this article we assume no data sharing between co-run programs.

2 BACKGROUND: HOTL THEORY

HOTL defines a set of characteristic functions of a program trace, and cache modeling entails operations on these functions [60]. Next, we review the functions and operations we will use in later sections.

The most important function is the footprint, which is the average working set. Following the notation of Denning [10], each time window in an execution trace is represented by (t, x) , where t is the end position and x is the window length. The number of distinct elements in the window is the *working-set size* (WSS) $\omega(t, x)$. For each x , the footprint $fp(x)$ is the average WSS of all windows of length x —that is, the total WSS divided by the number of length- x windows, as shown by the following equation:

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

where the parameter x is an integer $0 \leq x \leq n$ and n is the trace length.

From the footprint, the HOTL theory computes two functions of cache performance:

- *Miss ratio* $mr(c)$, which is the miss ratio of the fully associative LRU cache of all sizes $c \geq 0$, 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 c , computed by the inverse of the footprint.

Xiang et al. [60] called the second function volume fill time $vt(c)$. It is also known as the average eviction time (AET)—that is, the average time before a data block is evicted from cache after its last access [24].

Cache modeling becomes mathematical in HOTL. The miss ratio function is the derivative of the footprint, and the fill time is the inverse. Formally, we have

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

$$ft(c) = x \iff c = fp(x). \quad (3)$$

The two functions, fill time and footprint, are mappings between the dimensions of space and time. Figure 1 illustrates the two-way mappings by showing either dimension as a line. The first line is the spatial dimension measuring, in the unit of a cache block, the cache size and equivalently

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

the footprint, which is the data in cache. The second line is the temporal dimension measuring, in the unit of an access, the window length and equivalently the fill time. The mapping functions are shown beneath the lines: the fill time maps from space to time (e.g., c to x) and the footprint from time to space (e.g., x to c).

The HOTL theory states that the increase of the footprint is the miss ratio. From the point of the view of a single window, its WSS increases if and only if the next access is a miss. The footprint gives the average across all windows—so does its increase. 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 Equation (2). This is a key link between locality and cache performance—the one we will use to develop VFP.

The HOTL theory does not model exclusivity, which we consider next in the VFP theory.

3 VICTIM FOOTPRINT THEORY

This section first defines an abstraction of cache called the *split LRU stack* and then develops the theory of VFP.

3.1 Split LRU Stack

An LRU stack is the classic model of cache defined by Mattson et al. [35]. 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 either it is accessed again, when it is moved to H , or it is evicted. The L partition stores victims. It models the *victim cache*. As mentioned earlier, the victim cache was first invented by Jouppi in 1990 as a small, fully associative cache [30]. As an abstract model of the victim cache, the size of the lower partition can be arbitrarily large.

The shared victim cache is modeled by the sharing of the L partition. Figure 2(b) shows an example where an L partition serves the victims from two H partitions. On an AMD multicore processor, programs have private L1, L2 caches but share one L3 cache [1]. The private H partitions would include the L1s and L2s, and the L partition would be the shared L3.

Figure 3 simulates an LRU stack for a given access trace. Vertically, the stack positions are numbered top down, from the most recently accessed to least recently accessed.

Horizontally 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 Figure 3, are *eviction traces* or *victim traces*. Each row

Trace		a	b	c	d	d	c	b	a
	(1)	a	b	c	d	d	c	b	a
LRU	(2)		a	b	c	c	d	c	b
Stack	(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).

Fig. 4. Illustration of VFP and victim fill time. They are similar to the footprint and fill time in Figure 1, except the two lines are divided into three segments corresponding to the split LRU stack, and the victim functions map between the second segment of each line.

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 Figure 3.

The LRU stack by Mattson et al. [35] is an abstraction of cache in that it captures the common trait (i.e., the prioritizing of data) while parameterizing the implementation choice (i.e., the cache size). Similarly, the split LRU stack is an abstraction of exclusive cache. It captures the common trait, the split between two cache partitions, and two integer parameters $h, l \geq 0$. *All exclusive caches are instances of the split LRU stack.*

Next, we develop a new performance theory for the split LRU stack. It overcomes the limitation of HOTL. As shown in Figure 3, HOTL measures the locality of the original trace, not the victim traces, which are all different. The new theory will capture the locality of all victim traces.

3.2 Victim Footprint

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

3.2.1 VFP and Victim Fill Time. VFP is a function defined by using the footprint function $fp(x)$ and adding a second parameter h , the size of the H partition. Mathematically, the VFP is $vfp(h, x)$ for $h, x \geq 0$,

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

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

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

These two victim functions are illustrated in Figure 4. As the HOTL functions in Figure 1, the victim functions are mappings between the footprint (left line) and the window length (right line), but each line is now divided into three segments corresponding to the two parameters of the split LRU stack.

In the first graph in Figure 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 VFP $vfp(h, x_l)$.

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 functions—that is, $l = vfp(h, x_l)$ on the left and $x_l = vft(h, l)$ on the right.

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

The functions are defined for all $h, l \geq 0$ and therefore the (parameterized) split LRU stack. If we replace x_l with x and h with $f_p(x_h)$ in $vfp(h, x_l) = f_p(x_h + x_l) - f_p(x_h)$, we have Equation (4)—the definition of VFP.

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 counts not only the misses of the higher partition but all accesses (to the higher partition).

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

$$vmr(h, l) = vfp(h, x + 1) - vfp(h, x), \quad (6)$$

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

3.2.3 Correctness and Uniqueness. We formalize a requirement of the split LRU stack and then derive the VFP 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 therefore not shared), the split LRU stack should be identical to a single LRU stack of the size $h + l$. In terms of cache memory, when there are two cache levels, and they serve the same program, they should incur the same misses at the lower level as a single-level cache does.

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 miss ratio of a single cache with the combined size. Mathematically, the requirement means that

$$vmr(h, l) = mr(h + l) \quad (7)$$

for all $h, l \geq 0$. We call Equation (7) the *VCR equation*. For any specific cache, the VCR equation is a single equation, but for the split LRU stack, it is parameterized—the equality must hold for all nonnegative integers h, l .

We prove the *VFP theorem*, which states that the VFP is not only correct (i.e., it satisfies the VCR equation) but also unique (i.e., no other solution exists that satisfies the VCR equation).

THEOREM 3.1 (VFP THEOREM). *The VFP defined by Equation (4) is the only solution that satisfies the VCR stated in Equation (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 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 a single cache of size $h + l$. Hence, we have

$$vfp'(h, x + 1) - vfp'(h, x) = mr(h + l). \quad (8)$$

From Equation (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 Equation (4), we have $fp(x + t_h) = vfp(h, x) + h = l + h$, so we have

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

Combining Equations (8) and (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. $\square$

The proof of the VFP theorem is effectively the construction of $vfp(h, x)$ from $x = 0$, and the construction in effect derives Equation (4).

3.3 Composable Analysis of Cache Sharing

When sharing the cache, a set of co-run programs interact through shared cache. We first consider the sharing of victim cache on AMD processors and then two other types of sharing on Intel and IBM processors. In composable analysis, shared-cache miss ratio is *derived* without parallel testing.

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 L2s. Let the L2 size be c_2 and the L3 size be c_3 . Let the co-run group be $g = \{1, \dots, p\}$ and their VFPs be $vfp_i(h, x)$, $i = 1 \dots p$. In this article, we consider only independent programs that do not share data.

To explain, we start with a symmetric case, where a group G of p identical co-run programs are run together with uniform interleaving. The individual footprint is defined by the *solo logical clock*, which counts the accesses of a single program. We first normalize the time to the *co-run logical clock*, which counts all data accesses of the program group. A symmetric individual makes an access every p time ticks of the co-run clock.

Because of the time change, the co-run VFP of each program is the original VFP “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 the time change, the aggregate VFP, $vfp_G(c_2, x)$, is simply p times the individual VFP $vfp_{G_i}(c_2, x)$:

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

The victim cache (L3) miss ratio is the derivative of the group VFP:

$$vmr_G(c_2, c_3) = vfp_G(c_2, x + 1) - vfp_G(c_2, x), \quad (10)$$

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 has a distinct VFP $vfp_i(h, x)$. Second, in the parallel execution, each program has a distinct access rate ar_i , measured by 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 VFP of program i is “stretched” by $\frac{ar_i}{ar_G}$:

$$vfp_{G_i}(c_2, x) = vfp_i\left(c_2, \frac{x \cdot ar_i}{ar_G}\right). \quad (11)$$

The group VFP is their aggregate:

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

Finally, the group miss ratio is computed by Equation (10).

The access rate was used earlier in StatStack developed by Eklov et al. [16]. Brock et al. used the term *stretched footprint* [4] and called co-run logical clock *common logical time* [53]. A similar use was also given by Hu et al. [24] in computing the AET in shared cache.

Individual miss ratio. The miss ratio of individual programs in shared cache is the derivative of the individual VFP taken at the point where the group VFP is the cache size:

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

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

In shared cache, a program is affected by its peers. We can now show this interaction precisely in mathematical terms. Comparing the individual miss ratio computed in Equation (6) without sharing and in Equation (13) with sharing, we see that both equations differentiate, and both do so on the individual VFP. However, they differ in where to take the derivative. The private cache has one user, so differentiation happens when its VFP is equal to the cache size. The shared cache is communal, so differentiation happens when the *group* VFP is equal to the cache size.

Composition invariance. The group miss ratio can be “composed” in two ways: directly from the VFP of the group (Equation (10)) or indirectly as the sum of individual miss ratios (Equation (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 rearrangement of the terms in Equations (10), (12), and (13). It relies on the mathematical property of VFP operations: the differentiation is distributive over the addition.

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

3.3.2 Multilevel Sharing. AMD Bulldozer processors use cluster-based multithreading (CMT), where each core (or module) has two clusters. The new Zen processors use simultaneous multithreading (SMT) [7]. The two have the same effect of multilevel cache sharing. For example, a quad-core AMD processor can run eight programs in four pairs where each pair shares a separate L1 and L2, and all pairs share L3.

VFP can be used to model multilevel sharing as follows. The sharing of L1 is solved by HOTL (Section 2). In particular, we use the recent result by Brock et al. [4], who showed that HOTL implies the existence of a cache partition, called the *natural cache partition* (NCP), whose performance equals to that of cache sharing. The natural partition of a program is its effective occupancy in 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. [4], the effect of L1 sharing is equivalent to a partition of L1 into two parts of size $c_{1,a}, c_{1,b}$, where $c_{1,a} + c_{1,b} = c_1$ is 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 Equation (11), footprint composition in Equation (12), and miss ratio derivation in Equation (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

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

Finally, Equation (10) computes the co-run miss ratio from this group footprint.

In multilevel shared cache, program interaction at one level affects their interaction at the next level. We can now show this interactive interaction precisely in mathematical terms. Comparing the footprint aggregation computed in Equation (12) for single-level sharing and in Equation (14) for two-level sharing, we see that the first parameter differs in the VFP. In single-level sharing, the L1 cache is private, so the first parameter is the L1 size. In multilevel sharing, the L1 cache is shared, so the parameter is a fraction of the L1 size. The VFP models the split LRU stack. The first parameter is the upper-partition size, which can be any nonnegative integer. Hence, the multilevel analysis can model any effect of sharing in the previous level.

We model cache sharing in three or more 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 NCP of L1,L2. Then L3 sharing is modeled the same way as L2 has been in this section.

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

Finally, the VCR in Section 3.2.3 applies here. Before, it was two-level cache, not shared. Now, two programs a, b share both levels—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 NCP [4], we can prove that this shared cache VCR is satisfied by the VFP.

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 VFP) by taking the sequence of accesses for each cache set. A similar strategy exists in prior work to measure the reuse distance of accesses to each cache set [37, 41], which Sen and Wood [41] called the *per-set locality*. With per-set footprint, cache-way sharing is a special case of cache sharing. The split LRU stack defined in Section 3.1 can model two additional cache designs:

- **Intel's CAT:** Data of a program may be restricted (allocated) to use just a continuous group of cache ways [26]. When two CAT allocations partially overlap, there are three segments, as shown in Figure 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 [25]. The data of normal loads go to the MRU (most recently accessed) position, whereas the data of transient loads go to the LRU position, as shown in Figure 5(b). The

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

design can be viewed as a case of CAT, where normal loads are allocated all cache ways and transient loads only the last cache way.

VFP can be extended to model cache-way sharing. Here, we sketch the basic ideas:

- **Intel's CAT:** In Figure 5(a), the middle segment is shared between two programs. The interaction happens between the VFP of the first program and the footprint of the second program. The NCP is computed for the middle segment, and the effective occupancy for the second program determines the h parameter for its VFP for the third segment.
- **IBM's transient loads:** The last cache way in Figure 5(b) is a special case of the middle segment in Figure 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 Equation (2) is not always correct. The correct miss ratio is given by reuse distance. For instance, 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 that the footprint in all reuse windows has the same distribution as the footprint in all windows, for every window length [60].

The same reuse-window hypothesis is the condition of correctness for VFP 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 [60] and software caches including storage caches [15, 55], which has fixed data granularity, and Memcached [23], which does not. 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 [15] gave an example trace that has distinct phases, for which the average footprint gives the incorrect miss ratio on some cache sizes. A solution is to detect locality phases (e.g., using reuse distance [43]) 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. [39]

In practice, online footprint analysis and composition are necessary to analyze cache sharing. Since the VFP is mathematically derived from the footprint, they have identical measurement costs, which can be made very low by sampling (0.1 second overhead per program as reported in Wang et al. [53] and Xiang et al. [60]). The remaining question is whether the VFP can accurately model cache exclusivity. This section is the answer in theory. In the next section, we evaluate the accuracy of the VFP on real systems.

Table 1. Machines Used: One VFP Is Used to Predict Performance on Three Machine Models

Purpose	Profiling	2 Program Co-run	3/4 Program Co-run	Cache Allocation
Processor	Intel	AMD	AMD	Intel
	i7-4770	4226	FX-8120	E5-2630 V4
	Haswell	Opteron	Bulldozer	Broadwell-EP
Cache, L1 Data	32KB	16KB	16KB	32KB
L1 Instruction	32KB	64KB	64KB	32KB
L2	256KB	2MB	2MB	256KB
L3	8MB	8MB	8MB	25MB
OS	Fedora 17	Ubuntu15.04	Ubuntu14.04	Ubuntu 16.04
GCC	4.7.2	5.1.0	4.8.4	5.4.0

4 EVALUATION

The VFP theory models the performance of fully associative LRU cache. The cache on modern CPUs, however, is neither fully associative or exactly LRU. For practical relevance, it is vital to validate the VFP theory. This section presents a lengthy evaluation with five experiments: (1) accuracy of performance prediction for the (shared) exclusive cache, (2) comparisons between VFP and heuristics, (3) VFP-based optimal symbiotic scheduling for the exclusive cache, (4) main causes of prediction errors, and (5) the effect of non-LRU replacement policy.

4.1 Methodology

Table 1 shows the four machine models used in data collection and performance evaluation. An unusual aspect of the experimental design is that VFP is measured on an earlier model Intel machine (2013 first release) but used to predict performance on three other machines, two models of AMD (server and high-end desktop, both 2011) and a current model Intel machine (mid-2016 first release), with different processor architectures and cache hierarchies. The AMD caches are exclusive, whereas the Intel cache (on Broadwell-EP) is inclusive. The ability of cross-machine prediction comes from the generality of the VFP theory including the split LRU stack and composable analysis.

Implementation. We have implemented the composable analysis described in Section 3.3.1. VFP is derived from the footprint. To measure the footprint, we run a program with the dynamic binary rewriter Pin [31], which instruments every load and store instruction. As the program runs, we capture every load/store and use a hash table to identify first and last access times of each data block as well as all their interreference times. These times are stored in three histograms, which are then used to compute the footprint using the Xiang formula [59]. The method of measurement is identical to the one used previously and has similar costs—on average, 20 times slowdown [60]. Several techniques can measure the footprint in near real time using sampling [24, 53, 60]. We do not use sampling, as it complicates the experimental setup, and the results would have depended on sampling parameters.

Xiang et al. [60] showed that cache conflicts could be estimated using the Smith formula [22, 33, 44] but observed that the effect is negligible on modern large, highly associative CPU caches for the benchmarks we describe next. In all experiments, we use the prediction for fully associative cache and ignore set associativity in all prediction calculations.

Table 2. 28 SPEC CPU 2006 Benchmarks Grouped by Memory Intensity (L3 Miss Ratio)

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

Note: Ten of the programs (name in **bold**) are used in three- and four-program tests.

Actual machine versus simulation. All experiments in this article use actual machines. There is no simulation. However, simulation could strengthen the evaluation by allowing the analysis to dissect the effect of the model—for example, the cache geometry at each level, the effect of associativity, replacement policy, cache block size, and data prefetching (as we discuss in Section 4.5). A simulator could show how these cache parameters affect prediction accuracy.

Benchmarks. The evaluation uses SPEC CPU 2006 (version 1.1) benchmarks [46], 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*, as 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 distinguish phases. For programs with strong phase behavior, periodic sampling can predict the miss ratio more accurately [53].

We classify the benchmarks into three 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. [67], who defined cache intensity as L2 misses per million instructions.² Table 2 shows the three memory-intensity categories.

Exhaustive co-run testing. For completeness and no bias due to sampling, we test all two-program (duet) runs, the number of which is $\binom{28}{2} = 378$. For three- and four-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 alphabetical 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 numbers of tests $\binom{10}{3} = 120$ for triplets and $\binom{10}{4} = 210$ for quartets.

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

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 [29, 45, 60, 64]. It produces stable results and avoids

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

the problem of run-to-run performance variation. Such variability has been systematically studied by Sandberg et al. [39].

Metrics and measurement. The miss ratio is defined as the number of LLC misses divided by the number of memory accesses (unfiltered by L1/L2 caches). LLC misses and memory accesses are measured by the hardware counters as reported by the lightweight performance monitoring tool likwid [51]. The following events are counted. They include both on-demand accesses and prefetches. We do not disable the hardware prefetcher in experiments.

Event	Event IDMask	
DATA_CACHE_ACCESSES	0x040	N/A
UNC_L3_CACHE_MISS_ALL	0x4E1	0xF7

The miss ratio is computed as follows (see Drongowski and Boston Design Center [14] for a list of available hardware counters):

$$MissRatio = \frac{UNC_L3_CACHE_MISS_ALL}{DATA_CACHE_ACCESSES}.$$

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 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* memory-moderate, and *k* memory-light programs in Table 2. The lexicographical order of the category label is also the order of its intensity. Of duet-run categories for example, 0I0M2L is the least intense and 2I0M0L is the most intense. For lack of space, only duet and triplet tests are shown. Quartet results will be shown in the next section.

Figure 6 shows measured and predicted miss ratios and the absolute errors of prediction for duet groups (upper graph) and triplet groups (lower graph).⁴ The graphs divide 378 duet groups into six categories and 120 triplet groups into nine 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 category intensity.

In all categories except the most intense 3I0M0L, the average predicted miss ratio is lower than the actual, showing that VFP underpredicts more than it overpredicts. VFP underpredicts in 281 out of 378 duet groups and overpredicts in the remaining 97 groups. It underpredicts in 100 out of 120 triplet groups and overpredicts in the rest. The tendency does not depend on the miss ratio. The average miss ratio of the underpredicted 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 underpredict. A prefetcher estimates likely misses and may prefetch unnecessarily. A second reason is the effect of phases, which we 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.

Figure 7 shows the average relative error in all duet and triplet categories ordered (together in one graph) by intensity. The first six least intense categories (i.e., those with 0 intensive

³Per-program prediction can be tested when per-core counters are available, as it has been done in previous work [60].

⁴The results should be viewed within each graph, not compared between graphs, because two different processors were used for duet and triplet tests. However, they have the same cache sizes, as shown in Table 1.

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.

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

programs) have the highest relative errors, ranging between 44.7% and 61.5%. Their miss ratios are low, on average 0.03% and 0.55% for duet and triplet groups, respectively, so cache is unlikely the performance bottleneck, and these high relative errors are unlikely a problem in practice.

In the four 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).

4.3 Theory Versus Heuristics

As discussed at the start of the article, 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 versus 8MB L3.⁵

⁵L1,L2 are also exclusive, so the effective L2 size is 2MB + 64KB.

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.

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

- *VFP*, which is the analysis described in Section 3.3.1
- *HOTL*, which computes the miss ratio as described in Section 2 for a single cache of the combined size including LLC and all private caches (i.e., 12MB, 14MB, 16MB in two-, three-, and four-benchmark co-runs)
- *Even*, which assumes each co-run program uses an equal partition of the combined-size cache and then uses HOTL as described in Section 2 (Chen et al. [6] developed this heuristic for GPU caches.)
- *Proportional(MissRatio)* and *Proportional(MissRate)* 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.

Of the five techniques, VFP captures both effects of deduplication and victim cache sharing, and HOTL captures only the effect of deduplication. The remaining three methods capture both effects but are based on heuristics, which are simple assumptions on cache sharing. All methods require the footprint, so they have basically the same implementation cost.

Figure 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.

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 higher than VFP by 37%, 76%, and 85% for duet, triplet, and quartet tests. Hence, victim cache sharing differs from cache sharing, and the difference increases with the cache size, as shown by the growing gap between VFP and HOTL in Figure 8. The growth represents significant greater errors by HOTL as it models larger combined sizes in larger program groups (because of deduplication).

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 and has a similar implementation cost as VFP.

VFP is most accurate, and its advantage increases with the group size. This can be seen visually in Figure 8, where VFP is the curve that is highest and leftmost, 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 0.16%, 0.27%, and 0.32%, which are smaller.

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, which is 129% in triplet symbiosis and 240% in quartet symbiosis.

Proportional partitioning is incorrect for cache sharing. The two heuristics are worst performing, with the one based on the miss rate marginally better than the one based on the miss ratio. Beckmann and Sanchez [2] 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, 20, 44, 54] as discussed in Ding et al. [13]). 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 Snively and Tullsen [45]. We use the methodology developed by Wang et al. [53] to evaluate the quality of symbiosis. In this test, a symbiotic scheduler evenly divides a set of $2p$ programs into two 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) [18].

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

We compare the following four techniques:

- *OPT*, which tests the running time of all $\binom{2p}{p}$ group assignments and selects the best.
- *VFP*, which computes the miss ratio of all group assignments using VFP (Section 3.3.1) and chooses the best.
- *HOTL* by Wang et al. [53], which is similar to VFP but uses the HOTL theory (Section 2).
- *Distributed intensity (DI)* by Zhuravlev et al. [67], which sorts programs by their solo-run miss rates (misses per 1 million instructions) and assigns 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 [53].

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 six programs into two groups. The left graph of Figure 9 shows the

cumulative distribution (on the y -axis) of the 210 aggregate slowdowns (on the x -axis) obtained by each technique.

The median of the 210 aggregate slowdowns is 129.2% for VFP, shown by the vertical line in Figure 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 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 number of optimal symbiosis achieved out of 210 problems is 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, so it is worse than HOTL for 26% and DI for 14% of the tests.

Four-program co-runs. We also examine $p = 4$ and test all $\binom{10}{8} = 45$ problems for quartet symbiosis. Each problem has $\frac{\binom{8}{4}}{2} = 35$ ways to divide eight 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 Figure 9. HOTL, which was the second best in triplet symbiosis, becomes the worst in quartet symbiosis, because it fails to model the filtering effect. As shown by the results in Section 4.3, HOTL error increases substantially with the group size because of the larger combined cache size (due to deduplication). DI, in contrast, performs better for larger groups, showing that it is an effective scheduler for programs under high cache contention.

Private cache versus memory bandwidth versus shared cache. There is much greater resource contention when the group size increases from 3 to 4, evident by the range of slowdowns shown on the x -axis in Figure 9, which is 100% to 190% for triplets and 210% to 290% for quartets. The aggregate slowdowns nearly double when triplets become quartets. This suggests a saturation of memory bandwidth and controller resources, which Zhuravlev et al. [67] showed as having a greater effect than cache sharing. DI was developed as a heuristic solution (to balance the “intensity”). Instead of being qualitative, VFP and HOTL enable quantitative optimization by predicting the miss counts. The quartet results show that although the heuristic of DI is very effective under high contention, the quantitative theory of VFP still improves over the heuristic. But to do so, the theory must be exclusivity aware.

Heuristic versus theory. In the symbiosis for throughput, DI is effective and robust. However, DI does not predict the co-run miss ratio. In comparison, VFP predicts group (and individual) co-run miss ratios, as shown by the evaluation in Section 4.3, specifically Figure 8. The quantitative prediction is useful for providing a quality of service (QoS) guarantee beyond finding the best symbiosis. In addition, it is unclear how DI should be applied in multilevel sharing as described in Section 3.3.2, where the “intensity” of co-run programs in the LLC depends on how higher-level caches are shared. In comparison, the quantitative analysis by VFP predicts the sharing in higher-level caches.

The overhead of VFP prediction is small, $O(P * \log \log N)$ 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 and 0.22 seconds for all 120 triplet tests.

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.

Fig. 10. Across the test programs, the co-run prediction error by VFP correlates with the solo-run prediction error by HOTL but not with the co-run miss ratio.

Each program appears in 27 duet groups. Figure 10 shows the average VFP prediction error of the program in all its 27 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 WSS (Equation (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 [15] and empirical results by Wang et al. [53] on SPEC benchmarks, especially *mcf*. 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 WSS in every 1 million accesses in the period between 1 billionth and 2 billionth instructions. We measure WSS by the number of distinct data blocks accessed in each 1 million instruction interval. Figure 11 shows the WSS in four benchmarks. The four graphs are ordered by the per-program (in)accuracy as in Figure 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. For programs with regular phase variation, both VFP and HOTL theories make accurate predictions.

The hardware prefetcher is a possible source of errors. As mentioned before, we count both on-demand misses and prefetches. The hardware prefetcher may increase the miss count due to mis-prefetch (i.e., fetching data blocks not used by a program). However, there is no practical way to count mis-prefetches, so we cannot quantify the effect on the prediction accuracy. Furthermore, the prefetching effect may change between solo- and co-run tests and affect the accuracy differently.

4.6 Performance Prediction for Intel CAT

We evaluate the prediction on an Intel machine with an E5-2630V4 (codenamed “Broadwell-EP”) processor and 64GB memory. The machine differs from AMD in three aspects. First, it has

Fig. 11. Regular versus irregular phase variation in four programs shown by the WSS in the period between 1 billionth and 2 billionth accesses. The regularity of phase variation affects the VFP prediction accuracy shown in Figure 10.

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-EP processor.

very different cache sizes, shown in Table 1. In addition, Intel caches implement different cache replacement policies. The third and most recent difference is the Intel CAT [21], discussed earlier in Section 3.3.3.

We use CAT to run a program with 20 cache sizes, from 1.25MB to 25MB at 1.25MB increments. We use HOTL (Section 2) to predict these miss ratios. For validation, we use events OFFCORE_RESPONSE_1:ANY_REQUEST:L3_MISS, MEM_UOPS_RETIRED:ALL_LOADS, and MEM_UOPS_RETIRED:ALL_STORES. The first two count the number of accesses of loads and stores, respectively, and the last one counts the number of L3 misses. All of them count both on-demand accesses and prefetches.

Figure 12 shows the measured and predicted miss ratios for 560 tests,—that is, 28 SPEC CPU 2006 benchmark programs each using 20 cache sizes. The logarithmic scale is used since these miss ratios span many orders of magnitude.

The average prediction error is 0.52% in (absolute) miss ratio. The highest happens in *gcc* with a 1.25MB cache. The measured miss ratio is 5.46%, whereas the prediction is 0.77%. For a single program across all 20 cache sizes, the highest average error is 2.20% in *libquantum*, where the measured miss ratios range from 4.2% to 5.4% and the prediction ranges from 7.39% to 7.40%. The next highest average error is 2% in *milc*. All other programs have an average error less than 2.0%.

Modern Intel processors manage their caches using an adaptive replacement policy similar to re-reference interval prediction (RRIP) [28] and dynamic insertion policy (DIP) [38]. Both RRIP and DIP can detect the effect of streaming accesses and manage the cache to store none or only a part of the streaming data.

Due to the exclusivity, AMD caches (2MB L2 and 8MB L3) are equivalent to a 10MB LLC on Intel. We compare the AMD miss ratio to the Intel CAT miss ratio at 10MB. There are several differences that suggest RRIP/DIP-like benefits. For *libquantum*, HOTL predicts 7.4%, whereas the actual is 6.8% on AMD and between 4.2% and 5.4% on Intel. HOTL also overpredicts *GemsFDTD* 4.5% for 5.1%, *leslie3d* 3.3% for 2.8%, and *bzip2* 0.03% for 0.01% on Intel but not on AMD (4.8%, 3.7%, and 0.035%, respectively), likely because they have streaming accesses, and caching is improved by an adaptive insertion policy.

There are a total of seven programs where AMD and Intel differ by more than 0.5% in absolute miss ratio. Among them, the Intel miss ratio is lower in four and higher in three. In addition to the three just mentioned, the fourth program is *bwaves*, whose miss ratios are 3.832%, 3.00%, and 2.50% for Intel, AMD, and HOTL, respectively. AMD performs better on the other three programs. The miss ratio triples are (5.4%, 7.2%, 5.3%) for *milc*, (7.3%, 8.0%, 8.4%) for *mcfl*, and (0.75%, 1.3%, 0.2%) for *astar*.

On both machines, the largest prediction error by HOTL happens in *gcc* because of the irregular phase behavior, as shown in Section 4.5. For *cactusADM*, HOTL predicts relatively well on the first five cache sizes, 0.6% to 0.8% predicted versus 0.6% to 1.4% actual. However, at 7.5MB or higher, it predicts around 10^{-5} miss ratio, whereas 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.

As established in Section 4.5, VFP is accurate if HOTL is. This section shows the accuracy of HOTL for Intel caches and hence provides evidence that if there were Intel caches that were exclusive, VFP would still be an accurate model of exclusivity, as it is for AMD caches.

4.7 Review of Results

This section has presented five experiments. The first evaluated VFP in modeling the shared exclusive cache. Twenty-eight sequential runs were used to measure VFP and predict the miss ratio of 708 parallel runs, including 378 duet, 120 triplet, and 210 quartet groups. The prediction was validated by testing these groups on two types of AMD processors. In all test categories, the absolute error in miss ratio was 0.5%, except for one category, which was 0.6%. The absolute error did not increase with the miss ratio, so the relative error actually decreased. In the most memory-intensive category, the relative error was within 5%, corresponding to an absolute error of 0.2%.

The second experiment compared VFP to HOTL and three heuristics. It tested four 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 > two other heuristics. The theory of exclusivity performed the best. The average prediction error of HOTL (which modeled deduplication but not filtering) was near twice that of VFP in triplet and quartet tests. The best heuristic outperformed HOTL, but two other heuristics performed poorly.

The third experiment was exclusivity-aware program symbiosis, for a total of 255 tests including 210 triplet symbiosis and 45 quartet symbiosis. It compared VFP, HOTL, the heuristic solution DI, and optimal symbiosis, which required exhaustive testing of 2,100 triplet runs and 1,575 quartet runs. HOTL improved over DI at low contention and DI over HOTL at high contention. VFP improved over both, closing near 80% of the gap between the prior work and the optimal. Furthermore, VFP predicted individual miss ratios (and hence can ensure QoS), but DI cannot. Whether it is symbiosis for throughput or QoS, exclusivity modeling is necessary.

The fourth experiment showed that the error of VFP largely came from HOTL. It showed high correlation between VFP prediction errors in 708 co-run tests and HOTL prediction errors in 28 solo-run tests.

The last experiment was a first study of HOTL on Intel CAT and tested 20 cache sizes each for 28 programs. There are two main sources of errors in HOTL: irregular phase behavior and non-LRU cache replacement policy. However, among the 28 programs tested, these limitations affected just three or four programs each, and most of these errors were marginal. Furthermore, if HOTL becomes more accurate, VFP will become more accurate as well. The HOTL problems are orthogonal to the VFP theory, which targets cache exclusivity.

5 RELATED WORK

In this review, we focus on analytical models that 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 WSS and the number of page faults [10, 11, 12]. The recent theory by Xiang et al. [60] defines a new type of WSS called the *footprint* and shows the conversion between footprint, miss ratio curve, and reuse distance [60], as reviewed in Section 2. In both theories, the mathematical properties—monotonicity and concavity—are important for cache performance modeling. Neither theory considers cache exclusivity.

Thiébaud and Stone [50] coined the term *cache footprint*. Falsafi and Wood [19] redefined the *footprint* as the set of unique data blocks a program accesses. Both studies computed the footprint for a single window length, the length of the scheduling quantum [50].

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. [47] for choosing the best scheduling quantum, Chandra et al. [5] and Eklove et al. [16] and Eklove and Hagersten [17] for multicore cache sharing, and Shen et al. [42] for fast reuse distance analysis. The equations in previous work were not completely constrained, so the solution was not unique and depended on modeling assumptions. Xiang et al. [59] defined the footprint that is precise, deterministic, and measurable in linear time.

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

Early composable models used reuse distance and footprint and had only one way to compute the group miss ratio [5, 47, 58, 59]. Recent models are composition invariant, first from HOTL [4, 53, 60] and recently based on the the AET [24]. The effect of cache sharing is computed differently. In AET, sharing means that all co-run programs have the same AET. In VFP, as in HOTL, the sharing means that the group footprint is equal to the cache size. VFP is the first model of victim cache to have this property (see Section 3.3.1).

Online locality measurement. Initial theories were complex and too expensive to use in real-time solutions. Xie and Loh [61] noted that the model by Chandra et al. [5] (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. [67], as it was not “computationally fast enough to be used in the robust scheduling algorithm.”

Reuse distance is accurate and not affected by phase behavior. It has been used to model cache sharing for independent [40] and multithread programs [40, 56]. Reuse distance cannot be used in composable analysis, as shown by counterexamples in the past [13, 58]. Footprint and VFP enable composable analysis. However, their accuracy is affected by phase behavior, as discussed in Section 4.5.

For virtual memory management, reuse distance can be measured in real time in software [62, 66]. On CPU cache, Xiang et al. [60] showed a model that converts the footprint into reuse distance, so it no longer needs to measure the reuse distance. It takes 0.1 second per program to measure footprint with adaptive bursty sampling [53]. The efficiency has been further improved in SHARDS [52] and AET [24]. SHARDS measures reuse distance directly, based on reuse distance sampling [40, 65]. AET used random sampling pioneered by StatStack and found it to provide highest stability and accuracy [24]. AET enables composable analysis similar to HOTL and VFP. It is highly efficient, especially if its sampling is supported in hardware as it has been done by RapidMRC and StatStack [17, 49]. A recent CounterStack algorithm reduces the space complexity of reuse distance measurement asymptotically [55]. Efficient analysis of reuse distance can be used by VFP by first converting it to footprint using the HOTL theory [60]. With VFP, all of 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.

Shared-cache program symbiosis. The term *symbiotic scheduling* was coined by Snaveley and Tullsen [45]. It is also called *contention-aware scheduling* [67] or *thread co-scheduling* [63]. For shared cache, symbiosis is important in improving the total throughput or ensuring QoS. DI, although a simple heuristic, is highly effective [67]. Its performance in practice has been independently verified for inclusive caches ([53]) 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. Paragon was based on machine learning and had similar limitations [9]. Bubble-up predicted the co-run performance (not just the miss ratio), but its measurement was intentionally machine dependent (and probe dependent) [34]. PORPLE was developed for GPUs for data “symbiosis” (rather than task) [6] and assumed even partitioning of shared cache. Section 4.3 shows that the PORPLE heuristic is highly accurate for exclusive CPU caches. Still, VFP provides a greater precision at the same cost.

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

6 SUMMARY

The theory of VFP includes formalism and techniques to model the performance of exclusive cache hierarchies. The formalism includes the split LRU stack, VFP, victim fill time, and the properties, especially the uniqueness theorem. The techniques include composable analysis and prediction for any program groups sharing any types of AMD exclusive caches, Intel CAT, and IBM transient loads.

In evaluation, VFP is measured in a single set of 28 sequential executions and is used to predict the miss ratio in 4,383 parallel executions on two 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 for the most memory intensive and when used for shared-cache symbiosis removes near 80% of the gap between the best prior solution and the optimal. In conclusion, the VFP theory is practical and effective, and exclusivity modeling is necessary for its accuracy and benefit.

ACKNOWLEDGMENTS

The authors would like to express sincere thanks to Alex Benishek, Zhizhou Zhang, Linlin Chen and the anonymous referees of TACO for their careful corrections and suggestions.

REFERENCES

- [1] AMD. 2014. Software Optimization Guide for AMD Family 15h Processors. Retrieved September 26, 2017, from https://support.amd.com/TechDocs/47414_15h_sw_opt_guide.pdf.
- [2] Nathan Beckmann and Daniel Sanchez. 2016. Modeling cache performance beyond LRU. In *Proceedings of the 2016 IEEE International Symposium on High-Performance Computer Architecture (HPCA'16)*. IEEE, Los Alamitos, CA, 225–236.
- [3] Erik Berg, Håkan Zeffer, and Erik Hagersten. 2006. A statistical multiprocessor cache model. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*. 89–99.
- [4] Jacob Brock, Chencheng Ye, Chen Ding, Yechen Li, Xiaolin Wang, and Yingwei Luo. 2015. Optimal cache partition-sharing. In *Proceedings of the International Conference on Parallel Processing*.
- [5] Dhruva Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. 2005. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA'05)*. 340–351.
- [6] Guoyang Chen, Bo Wu, Dong Li, and Xipeng Shen. 2014. PORPLE: An extensible optimizer for portable data placement on GPU. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*.
- [7] Mike Clark. 2016. A new x86 core architecture for the next generation of computing. In *Proceedings of the 2016 IEEE Hot Chips 28 Symposium (HCS'16)*. IEEE, Los Alamitos, CA, 1–19.
- [8] Pat Conway and Bill Hughes. 2007. The AMD Opteron Northbridge architecture. *IEEE Micro* 27, 2, 10–21.
- [9] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 77–88.
- [10] Peter J. Denning. 1968. The working set model for program behaviour. *Communications of the ACM* 11, 5, 323–333.
- [11] Peter J. Denning and Stuart C. Schwartz. 1972. Properties of the working set model. *Communications of the ACM* 15, 3, 191–198.
- [12] Peter J. Denning and Donald R. Slutz. 1978. Generalized working sets for segment reference strings. *Communications of the ACM* 21, 9, 750–759.
- [13] Chen Ding, Xiaoya Xiang, Bin Bao, Hao Luo, Ying-Wei Luo, and Xiao-Lin Wang. 2014. Performance metrics and models for shared cache. *Journal of Computer Science and Technology* 29, 4, 692–712. DOI: <http://dx.doi.org/10.1007/s11390-014-1460-7>
- [14] Paul J. Drongowski and Boston Design Center. 2008. *Basic Performance Measurements for AMD Athlon 64, AMD Opteron and AMD Phenom Processors*. AMD Whitepaper 25. AMD.
- [15] Zachary Drudi. 2014. *A Streaming Algorithms Approach to Approximating Hit Rate Curves*. Masters Thesis. University of British Columbia.
- [16] David Eklov, David Black-Schaffer, and Erik Hagersten. 2011. Fast modeling of shared caches in multicore systems. In *Proceedings of the International Conference on High-Performance Embedded Architectures and Compilers*. 147–157.
- [17] David Eklov and Erik Hagersten. 2010. StatStack: Efficient modeling of LRU caches. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*. 55–65.
- [18] Stijn Eyerman and Lieven Eeckhout. 2010. Probabilistic job symbiosis modeling for SMT processor scheduling. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 91–102. DOI: <http://dx.doi.org/10.1145/1736020.1736033>
- [19] Babak Falsafi and David A. Wood. 1997. Modeling cost/performance of a parallel computer simulator. *ACM Transactions on Modeling and Computer Simulation* 7, 1, 104–130.

- [20] A. Fedorova, M. Seltzer, and M. D. Smith. 2007. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*.
- [21] Andrew Herdrich, Edwin Verplanke, Priya Autee, Ramesh Illikkal, Chris Gianos, Ronak Singhal, and Ravi Iyer. 2016. Cache QoS: From concept to reality in the Intel® Xeon® processor E5-2600 v3 product family. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA'16)*. 657–668. DOI: <http://dx.doi.org/10.1109/HPCA.2016.7446102>
- [22] Mark D. Hill and Alan Jay Smith. 1989. Evaluating associativity in CPU caches. *IEEE Transactions on Computers* 38, 12, 1612–1630.
- [23] Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo, Chen Ding, Song Jiang, and Zhenlin Wang. 2015. LAMA: Optimized locality-aware memory allocation for key-value cache. In *Proceedings of the USENIX Annual Technical Conference*.
- [24] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Chen Ding, and Zhenlin Wang. 2016. Kinetic modeling of data eviction in cache. In *Proceedings of the USENIX Annual Technical Conference*. 351–364. <https://www.usenix.org/conference/atc16/technical-sessions/presentation/luo>.
- [25] IBM. 2015. *POWER8 Processor User's Manual for the Single-Chip Module*. IBM.
- [26] Intel. 2015. *Improving Real-Time Performance by Utilizing Cache Allocation Technology*. Intel.
- [27] Intel Corporation. 2017. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Number 248966-037. Intel.
- [28] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely Jr., and Joel Emer. 2010. High performance cache replacement using re-reference interval prediction (RRIP). *ACM SIGARCH Computer Architecture News* 38, 60–71.
- [29] Yunlian Jiang, Kai Tian, and Xipeng Shen. 2010. Combining locality analysis with online proactive job co-scheduling in chip multiprocessors. In *Proceedings of the International Conference on High-Performance Embedded Architectures and Compilers*. 201–215.
- [30] Norman P. Jouppi. 1990. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*. IEEE, Los Alamitos, CA, 364–373.
- [31] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 190–200.
- [32] Hao Luo, Pengcheng Li, and Chen Ding. 2017. Thread data sharing in cache: Theory and measurement. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 103–115. <http://dl.acm.org/citation.cfm?id=3018759>.
- [33] G. Marin and J. Mellor-Crummey. 2004. Cross architecture performance predictions for scientific applications using parameterized models. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*. 2–13.
- [34] Jason Mars, Lingjia Tang, Kevin Skadron, Mary Lou Soffa, and Robert Hundt. 2012. Increasing utilization in modern warehouse-scale computers using bubble-up. *IEEE Micro* 32, 3, 88–99.
- [35] R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger. 1970. Evaluation techniques for storage hierarchies. *IBM System Journal* 9, 2, 78–117.
- [36] Javier Merino, Valentín Puente, Pablo Prieto, and José Ángel Gregorio. 2008. SP-NUCA: A cost effective dynamic non-uniform cache architecture. *ACM SIGARCH Computer Architecture News* 36, 2, 64–71.
- [37] Cedric Nugteren, Gert-Jan van den Braak, Henk Corporaal, and Henri E. Bal. 2014. A detailed GPU cache model based on reuse distance theory. In *Proceedings of the International Symposium on High-Performance Computer Architecture*.
- [38] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel S. Emer. 2007. Adaptive insertion policies for high performance caching. In *Proceedings of the International Symposium on Computer Architecture*. 381–391.
- [39] Andreas Sandberg, Andreas Sembrant, Erik Hagersten, and David Black-Schaffer. 2013. Modeling performance variation due to cache sharing. In *Proceedings of the International Symposium on High-Performance Computer Architecture*. 155–166. DOI: <http://dx.doi.org/10.1109/HPCA.2013.6522315>
- [40] Derek L. Schuff, Milind Kulkarni, and Vijay S. Pai. 2010. Accelerating multicore reuse distance analysis with sampling and parallelization. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*. 53–64.
- [41] Rathijit Sen and David A. Wood. 2013. Reuse-based online models for caches. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*. 279–292.
- [42] Xipeng Shen, Jonathan Shaw, Brian Meeker, and Chen Ding. 2007. Locality approximation using time. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 55–61.

- [43] Xipeng Shen, Yutao Zhong, and Chen Ding. 2004. Locality phase prediction. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 165–176.
- [44] A. J. Smith. 1976. On the effectiveness of set associative page mapping and its applications in main memory management. In *Proceedings of the International Conference on Software Engineering*.
- [45] Allan Snaveley and Dean M. Tullsen. 2000. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 234–244.
- [46] SPEC-CPU. SPEC CPU Benchmarks. (2017). Retrieved September 26, 2017, from <http://www.spec.org/benchmarks.html#cpu>.
- [47] G. Edward Suh, Srinivas Devadas, and Larry Rudolph. 2001. Analytical cache models with applications to cache partitioning. In *Proceedings of the International Conference on Supercomputing*. 1–12.
- [48] David Tam, Reza Azimi, Livio Soares, and Michael Stumm. 2007. Managing shared L2 caches on multicore systems in software. In *Proceedings of the Workshop on the Interaction Between Operating Systems and Computer Architecture*. 26–33.
- [49] David K. Tam, Reza Azimi, Livio Soares, and Michael Stumm. 2009. RapidMRC: Approximating L2 miss rate curves on commodity systems for online optimizations. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 121–132.
- [50] Dominique Thiébaud and Harold S. Stone. 1987. Footprints in the cache. *ACM Transactions on Computer Systems* 5, 4, 305–329.
- [51] Jan Treibig, Georg Hager, and Gerhard Wellein. 2010. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops (ICPPW'10)*. IEEE, Los Alamitos, CA, 207–216.
- [52] Carl A. Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. 2015. Efficient MRC construction with SHARDS. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. 95–110.
- [53] X. Wang, Y. Li, Y. Luo, X. Hu, J. Brock, C. Ding, and Z. Wang. 2015. Optimal program symbiosis in shared cache. In *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid*.
- [54] Richard West, Puneet Zaroo, Carl A. Waldspurger, and Xiao Zhang. 2010. Online cache modeling for commodity multicore processors. *Operating Systems Review* 44, 4, 19–29.
- [55] Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas J. A. Harvey, Andrew Warfield, and Coho Data. 2014. Characterizing storage workloads with counter stacks. In *Proceedings of the Symposium on Operating Systems Design and Implementation*. 335–349.
- [56] Meng-Ju Wu and Donald Yeung. 2013. Efficient reuse distance analysis of multicore scaling for loop-based parallel programs. *ACM Transactions on Computer Systems* 31, 1, 1. DOI: <http://dx.doi.org/10.1145/2427631.2427632>
- [57] Meng-Ju Wu, Minshu Zhao, and Donald Yeung. 2013. Studying multicore processor scaling via reuse distance analysis. In *Proceedings of the International Symposium on Computer Architecture*. 499–510.
- [58] Xiaoya Xiang, Bin Bao, Tongxin Bai, Chen Ding, and Trishul M. Chilimbi. 2011. All-window profiling and composable models of cache sharing. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 91–102.
- [59] Xiaoya Xiang, Bin Bao, Chen Ding, and Yaoqing Gao. 2011. Linear-time modeling of program working set in shared cache. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*. 350–360.
- [60] Xiaoya Xiang, Chen Ding, Hao Luo, and Bin Bao. 2013. HOTL: A higher order theory of locality. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 343–356.
- [61] Yuejian Xie and Gabriel H. Loh. 2008. Dynamic classification of program memory behaviors in CMPs. In *Proceedings of the CMP-MSI Workshop*.
- [62] Ting Yang, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. 2006. CRAMM: Virtual memory support for garbage-collected applications. In *Proceedings of the Symposium on Operating Systems Design and Implementation*. 103–116.
- [63] Eddy Z. Zhang, Yunlian Jiang, and Xipeng Shen. 2012. The significance of CMP cache sharing on contemporary multithreaded applications. *IEEE Transactions on Parallel and Distributed Systems* 23, 2, 367–374.
- [64] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. 2009. Towards practical page coloring-based multicore cache management. In *Proceedings of the EuroSys Conference*. 89–102.
- [65] Yutao Zhong and Wentao Chang. 2008. Sampling-based program locality approximation. In *Proceedings of the International Symposium on Memory Management*. 91–100. DOI: <http://dx.doi.org/10.1145/1375634.1375648>

- [66] Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. 2004. Dynamic tracking of page miss ratio curve for memory management. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 177–188.
- [67] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. 2010. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 129–142.

Received June 2017; revised July 2017; accepted August 2017