Write Locality and Optimization for Persistent Memory

Dong Chen

Chencheng Ye[§]

Chen Ding

Department of Computer Science, University of Rochester
500 Joseph C. Wilson Blvd
Rochester, New York 14627
{dchen39, cye, cding}@cs.rochester.edu

ABSTRACT

Persistent memory is a disruptive technology that drastically reduces memory cost and static power but introduces the problems of slow writes and limited write endurance. An effective solution is caching. However, existing cache has been designed for fast reads. It does not minimize the number of writebacks from cache to memory.

In this paper, we propose a metric to quantify the write locality and a theory to analyze and optimize write locality. It includes a linear-time algorithm to predict the write-back frequency for all cache sizes. In shared cache, it predicts the number of writebacks for co-run programs based on sole-run profiling. The paper evaluates the accuracy of the prediction against cache simulation. It then uses the theory to optimize write locality in a set of co-run programs in shared cache by cache partitioning. The theory predicts that such write-locality optimization can reduce the number of writebacks by 12% to 35%, compared to uncontrolled cache sharing.

Keywords

Write locality; Cache-sharing optimization

1. INTRODUCTION

Compared to existing DRAM, non-volatile memory (NVM) such as Phase change memory (PCM) and memristor usually have lower energy consumption, larger capacity, larger read/write latency and limited endurance as shown in Table 1.

Future design is likely a hybrid to combine the strength of both types of memory. Two hybrid designs are proposed in the previous research as Figure 1 shows. The first is vertical, shown in Figure 1(a), where DRAM serves as the cache for NVM. CPU read from or write to DRAM but not NVM directly. Moniuddin proposed this organization and intro-

[§]Chencheng Ye is a student visiting University of Rochester, he comes from Huazhong University of Science and Technology, funded by Chinese Scholarship Council.

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.

MEMSYS 2016 October 3–6, 2016, Washington, DC, USA

© 2016 ACM. ISBN 978-1-4503-4305-3.

DOI: <http://dx.doi.org/10.1145/2989081.2989119>

Table 1: DRAM vs NVM

	Read (ns)	Write (ns)	Endurance (cycle)
DRAM	10 – 50	10 – 50	10^{17}
Memristor	< 10	20 – 30	10^{12}
PCM	20 – 70	50 – 500	$10^7 - 10^8$
STT-RAM	10 – 30	13 – 95	10^{15}
Flash	2500	200000	$10^3 - 10^6$

duced a write queue between DRAM and NVM to hide the write latency of NVM [13]. In organization (b), DRAM and NVM are on the same level. CPU can access both DRAM and NVM directly. Zhang proposed the side by side organization in 3D die-stacking memory. Pages are allocated in either DRAM and NVM initially. Sensors are used to monitor the temperature of PRAM regions as an indication of write density [27]. While both organizations achieve promising results, Su et al. showed that none of them alone can achieve the best performance and lowest energy consumption for a range of high performance computing (HPC) applications. They proposed a hybrid memory controller to switch between the two designs[18].

Figure 1: Hybrid memory design

Regardless of the approaches, a common problem is the use of caching to reduce the number writes to NVM. In DRAM only systems, different cache coherence policies such as write-back, write-through and write-around are designed to reduce the data access latency and I/O contention for different scenarios. With the emerging NVM technologies, writeback reduction should also be considered in order to address the problem of limited endurance of NVM and ameliorate the high latency and power consumption of NVM writes.

Locality theory is useful in studying memory systems because it is precise and abstract, independent of cache implementation. The independence is needed for portable performance, and in the case of hybrid memory, also needed because the material, design and key performance parameters of persistent memory are highly varied and rapidly evolving. It is possible that multiple types of persistent memories may be used to run a single program. As a result, we need a machine-independent theory.

The genesis of the locality problem is the asymmetry in latency and bandwidth between nearby and farther away memory. Persistent memory gives rise to another asymmetry — the performance and cost between memory reads and writes. Traditional locality deals almost entirely with read accesses, and locality optimization read misses. The cost of writebacks was unimportant. On persistent memory, however, writes are special. Unlike reads, write backs are asynchronous, happening at the background long after a cache block is last accessed. The existing theory of locality is inadequate, and we need a new theory of write locality.

In this paper, we focus on leveraging cache to reduce the number of writebacks. First, we propose a novel metric to compute the writeback ratio of a program in linear time. Then we show how to optimally partition the cache for co-run programs to minimize the number of writebacks. Using SPEC benchmarks, we evaluate the effect of optimal partitioning with equal partitioned cache and with cache sharing. We also characterize the cache behavior of these programs when they are in co-run groups.

2. WRITE LOCALITY

2.1 Formal Notions of Locality

Locality has many meanings. Before defining write locality, we introduce two basic requirements such a definition should meet. First, it should be a characteristic of a program. Like parallelism, locality should be machine independent. This independence is important because it enables locality optimization that is not tied to a specific machine. In practice, the distinction separates machine-independent and machine-dependent improvements. Second, the locality should be quantitative in that it is uniquely defined for a given execution. Only when it is quantified can we compare and optimize locality.

Machine independence, however, is relative. The ultimate goal of locality optimization is to maximize cache utilization. The real meaning of machine independence is general (rather than limited) applicability: from the locality, we should be able to derive cache performance of all machines.

There are multiple definitions of locality that are quantitative and machine independent: reuse distance [5], working-set size [4], and footprint [23]. They can be used to derive the miss ratio for all-size caches. Knowing all-size cache

performance is important in shared cache, where the cache occupancy of a program can be any size between 0 and the size of the cache.

Locality is a composite effect — the locality of one access depends on other accesses. Data writes add two new problems to the mix. First, a write back in cache is asynchronous and may happen long after the cache block is last written. Second, the effect of a write is cached, and the “dirty” state extends beyond the ensuing reads. As a result, a write back may follow a read access.

For write locality, we want a metric to capture the essential character of a program enough to derive the number of writebacks for all-size caches. We make the following assumptions about the cache system:

- Fully associative LRU cache of all (integer) sizes
- The cache is write back, not write through. It uses write allocate at a write miss.

We first introduce the footprint locality, which we will use to define write locality.

2.2 Background: Higher Order Theory of Locality

A higher order theory of locality (HOTL) was formulated by Xiang et al. [23]. The theory defines the following metrics to quantify locality, which we will use in later sections.

Window $W(size)$: The continuous memory accesses with length $size$ in memory access trace. For example, $W[0, 2)$ is the memory access sequence “a b” in the trace shown in Table 2.

Footprint $fp(w)$: The averaged working set size (WSS) for window w where WSS for window w is the number of distinct memory address accessed and n is the length of the trace.

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

Reuse Time $rt(i, j)$: The difference between the reference time of two memory accesses i and j . For example, the reuse time between the first two neighboring “a” is 3 which is $4 - 1$.

$$rt(i, j) = j - i \quad (2)$$

Fill Time $ft(c)$: The number of memory references needed to fill up the cache with size c . As footprint is the average number of distinct memory accesses in window with length w , that is to say the number of distinct memory accesses which equals to $fp(w)$ requires w memory accesses on average. So we have the relation that fill time is the inverse function of footprint.

$$ft(c) = fp^{-1}(c) \quad (3)$$

Miss Ratio $mr(c)$: A memory access is a miss if its reuse time is greater than the fill time $ft(c)$. The miss ratio is the portion of memory accesses whose reuse time is greater than $ft(c)$.

2.3 Write Caching

Memory accesses have different addresses and two accessing types: read or write. A memory write may or may not lead to a writeback. If a data block resides in cache for two consecutive writes, then a write back is saved because of cache reuse. We call this a *write reuse*. The number of writebacks is the number of writes minus the number of write reuses. A write reuse requires the cache residence between two consecutive writes, which depends on four factors. The first is the distance between the writes. The second is the reads of the data block between these two writes. The third is the access to other data blocks between the two writes. The last is the size of the cache. To model all these factors, we introduce the concept of a *write interval*.

A **Write Interval** is the time window between two consecutive writes to some address. There is no other intervening writes to that address in the interval, although there can be reads to that address.

An example is shown in Table 2. The example trace has 10 accesses with 7 writes to 3 different addresses. There are 4 write intervals.

Table 2: Decompose accesses into write intervals

reference time	1	2	3	4	5	6	7	8	9	10
address	a	b	c	a	b	c	c	c	a	b
write/read	w	r	w	r	w	r	w	w	w	w
write interval a	w	-	-	r	-	-	-	-	w	
write interval b					w	-	-	-	-	w
write interval c_0			w	-	-	r	w			
write interval c_1							w	w		

Each write interval contains r read accesses $r \geq 0$ to the same cache block, which divide the interval into $r + 1$ sub-intervals. Each sub-interval is measured by a reuse distance, which is the number of distinct data blocks accessed. Among all $r + 1$ reuse distances, we choose the *largest* value as the *write reuse distance*.

Reuse time is faster to measure than reuse distance. To leverage the efficiency of reuse time, we take each write interval and divide it into sub-intervals as before. Instead of reuse distance, we use the length of each sub-interval, i.e. the reuse time. The write reuse time (*WRT*) is the largest of all reuse times in a write interval. Let $wrt(i)$ be the histogram of all write-back reuse times and $P(wrt > x)$ be its cumulative probability function.

For the example shown in Table 2, Table 3 shows the footprint and fill time in the upper half of the table. In the lower half, it shows the write locality. For each write interval, it shows the write reuse time. From the write reuse time, it determines whether it leads to a writeback by comparing the write reuse time with the fill time $ft(c = 2)$.

2.4 Counting Writebacks in All Cache Sizes

We formulate writeback count (WBC) as a metric computed from Equation 4. The function $wbc(c)$ is the total number of writebacks for cache size c . WI is the set of all write intervals in the trace. $I(x)$ is the indicator function which outputs 1 if x is greater than 0; otherwise, it outputs 0.

Table 3: Maximum reuse time and writebacks

Win / Cache Size	1	2	3	4
Footprint	1	1.78	2.5	2.71
Fill Time	1	2.31	∞	∞
Write Interval	a	b	c_0	c_1
Write reuse time	5	5	3	1
Writeback ($c = 2$)	Yes	Yes	Yes	No

$$wbc(c) = \sum_{i \in WI} I(i's\ wrt > ft(c)) \quad (4)$$

With Equation 4, we have to record all the WRTs for each write interval. But different write intervals may have the same WRT which will lead to redundant comparison between WRT and fill time. The histogram wrt to get the count for each unique WRT and sum the count if the WRT is greater than fill time. The equation is shown in Equation 5 which is equivalent to Equation 4.

$$wbc(c) = \sum_{t=ft(c)}^n wrt(t) \quad (5)$$

Algorithm 1 is a linear time $O(n)$ write locality analysis based on Equation 4. It has two procedures: an online algorithm to calculate the WRT histogram, and the other calculate writeback count from the WRT histogram and the fill time computed according to Equation 5.

Procedure *OnlineWRTHisto* is called for each memory access with the address *addr*, access type *type* (which is READ or WRITE) and reference time *time* (which is the total number of accesses so far) as the inputs. Three hash tables are used: *wrt* is the histogram for the write reuse time for write intervals, *prev* is used to record the reference time of the latest access for each distinct memory address and *cmrt* is used to record the maximum reuse time in a write interval. If an access is the first write to *addr*, the procedure initializes *cmrt*[*addr*] to 0. If an access is a read in the write interval of *addr*, the procedure compares the past maximal reuse time *cmrt*[*addr*] with the current reuse time *time* - *prev*[*addr*] and updates *cmrt*[*addr*]. If an access is a write but not the first write to *addr*, the algorithm adds the maximal reuse time to *wrt* and reset *cmrt* to 0. Once we have the *wrt*, Procedure *WritebackAllWindows* iterates over all cache sizes to compute the number of writebacks. We assume that the number of cache sizes is constant, e.g. three cache levels. The overall time complexity of the algorithm is $O(n)$, where n is the number of memory accesses.

2.5 Write Ratio and Write Back Ratio

Let n be the number of accesses in an execution and n_w the number of writes. The write ratio is

$$wr = \frac{n_w}{n} \quad (6)$$

The write back ratio for cache size c is

$$wbr(c) = \frac{wbc(c)}{n_w} \quad (7)$$

We choose to define the write-back ratio over all memory writes rather than all accesses for two reasons. First, the

Algorithm 1 $O(n)$ writeback prediction algorithm

```
1: wrt : Hash table for write reuse time histogram
2: procedure ONLINEWRTHISTO(addr, type, time)
3:   prev : Hash table for previous access time
4:   cmrt : Hash table for current maximum reuse time
5:   if addr is not in prev.keys() then
6:     cmrt[addr] = 0
7:   end if
8:   if type is 'WRITE' then
9:     if addr is in cmrt.keys() then
10:      if cmrt[addr] > time - prev[addr] then
11:        wrt[cmrt[addr]] + = 1
12:      else
13:        wrt[time - prev[addr]] + = 1
14:      end if
15:    end if
16:    cmrt[addr] = 0
17:  else
18:    if addr is in cmrt then
19:      if cmrt[addr] < time - prev[addr] then
20:        cmrt[addr] = time - prev[addr]
21:      end if
22:    end if
23:  end if
24: end procedure
25:
26: procedure WRITEBACKALLWINDOW(fp)
27:   for each cacheSize to be predicted do
28:     filltime = w where fp[w] == cacheSize
29:     wbc[cacheSize] = 0
30:     for each reuse time i > filltime do
31:       wbc[cacheSize] + = wrt[i]
32:     end for
33:   end for
34: end procedure
```

write-back ratio ranges between 0 and 1, with the worse case equal to 1 (when cache size is 0). The magnitude of the ratio shows the effect of write caching, i.e. buffering writes before a writeback. Second, writebacks are a problem for endurance. For endurance, we are not concerned with memory reads. Writebacks differ from cache misses. Misses are problems of performance. They are on the critical path of the execution. Writebacks are asynchronous and buffered and do not affect execution speed directly.

The write ratio shares two properties with the well known miss ratio, which is the ratio of all cache misses to all accesses. First, the miss ratio also ranges between 0 and 1, with the worse case equal to 1 (when the case size is 0). Second, a lower ratio means better locality in the cache. If we were to define the write-back ratio based on all accesses, the maximal ratio would be *wr* the write ratio, and a lower write-back ratio would have not been necessarily the result of write locality. With the current definition, a lower write ratio means better write locality.

2.6 Write Locality in Shared Cache

Write locality as defined in the previous sections is composable for independent programs that do not share data. According to Equation 5, writeback is calculated from WRT histogram and footprint, which are both composable.

For the miss ratio, Wang [21] proposed a composition

method by normalizing the footprints for solo runs through access rates *r* by Equation 8. As window size for footprint represents the number of accesses (reference time range), so for co-run programs the accesses in a window are contributed by all the programs and the portion contributed is depended on the access rate. The higher the access rate, the more accesses it contributes. For *p* co-run programs and a window with size *w*, each program *i* will contribute $w \times \frac{r_i}{\sum_{j=1..p} r_j}$ accesses. So the joint footprint for program group *g* is the sum of all the solo footprint of the decomposed windows.

$$fp_g(w) = \sum_{i=1..p} fp_{solo,i}(w \times \frac{r_i}{\sum_{j=1..p} r_j}) \quad (8)$$

WRT histogram is counting the maximum reuse time for write intervals. Compared to the time measured in solo runs, the time range of a write interval as well as WRT will be lengthened by the accesses from the co-run peers. That is to say, co-run WRT *t* for *p* co-run programs will shrink to $t \times \frac{r_i}{\sum_{j=1..p} r_j}$ when the program *i* runs solo. We propose a probability function *P*(*t*) to calculate the probability that WRT is greater than *t*. As when maximum reuse time is greater than the fill time of the cache size, there will be a write back. *P*(*t*) is equivalent to *wbr*(*fp*(*t*)) as Equation 9 shows. The co-run probability of WRT is greater than joint fill time *ft_g*(*c*) can be formulated by Equation 10 which gives the write back ratio for cache size *c*.

$$P(t) = \frac{\sum_{x=t}^{\infty} wrt(x)}{n_w} = \frac{\sum_{x=t}^{\infty} wrt(x)}{n_w} = wbr(fp(t)) \quad (9)$$

$$wbr(c) = P_{co}(ft_g(c)) = \sum_{i=1}^p P_{solo}(ft_g(c) \times \frac{r_i}{\sum_{j=1..p} r_j}) \quad (10)$$

So we can use a two-element tuple of maximum reuse time histogram *wrt* and footprint *fp* to represent writeback as $\langle wrt, fp \rangle$. For *p* programs, we have *p* tuples for solo writebacks $\langle wrt_0, fp_0 \rangle, \langle wrt_1, fp_1 \rangle, \dots, \langle wrt_{p-1}, fp_{p-1} \rangle$. The co-run write back ratio can be calculated by Equation 9 and 10.

3. WRITE LOCALITY OPTIMIZATION

In a modern CPU architecture, the Last Level Cache (LLC) is usually shared. Uncontrolled sharing may lead to poor performance or unfair sharing. Intel cache allocation technology (CAT) provides hardware support to partition the cache [7]. In this section, we assume that the cache can be partitioned at the granularity of a cache block and gives a method to maximize write locality through cache partitioning.

In this section, we show first that cache sharing is equal in performance to a natural cache partition, then a method to partition the cache for maximal write locality, and finally a method to balance write locality and read locality.

3.1 Natural Cache Partition

Consider a group *g* of co-run programs with IDs range from 1 to *p* sharing the cache, $g = \{1, \dots, p\}$. In actual use, the cache occupancy of a program varies from time to time and is usually not fixed. Brock et al. showed that in theory, natural cache partition (NCP) computes a constant,

effective cache occupancy o_i for each program $i = 1 \dots p[1]$. They proved two properties. First, a program's miss ratio in the shared cache is the same as its miss ratio in a private cache of size o_i , its effective cache occupancy in the shared cache. Second, the cache occupancies, $(o_1, o_2, \dots, o_p)$ form a partition of the shared cache. They called it the *natural cache partition*.

NCP is a mathematical result. The shared cache “behaves” the same way as a partitioned cache with the natural partition. The performance of any shared cache is then equivalent to the performance of naturally partitioned cache. The problem of finding the best partition-sharing is thus reduced to finding the best partitioning.

In the method in Section 2.6, the footprint is composed in the same way for write locality as it is for read locality. The natural cache partition derived by Brock et al. has the same number of writebacks, as it has the same number of cache misses, as the shared cache. Therefore, the problem of finding the best partition-sharing for write locality is also reduced to finding the best partitioning.

3.2 Optimal Partitioning for Write Locality

A locality model captures the essential character of a program enough to derive the miss ratio. It is useful to optimize cache sharing in two ways. The first is program symbiosis, that is, how to divide a set of programs into peer groups that minimize the miss ratio in shared cache. The second is cache partitioning, that is, how to divide the cache among a set of programs to minimize the total miss ratio. Previous work solved these problem individually and for the miss ratio [21, 1]. In this section, we solve the problem of joint optimization for write locality and present an integrated method that chooses the best program group that minimizes the total writebacks under cache partitioning. No other program grouping or other cache partition can further improve the write locality.

Figure 2 shows that the number of solution candidates is $\binom{n}{m} \times \binom{p-1}{m-1}$. First we need to choose m programs out of n . There are $\binom{n}{m}$ choices. Then for the m chosen programs, we partition the cache into m partitions, which can be seen as putting $m - 1$ separators between p cache ways. This will guarantee that each program will have at least one way and the sum of the ways assigned to all m programs equal to p , so it is a cache partition. There are $\binom{p-1}{m-1}$ partition choices. To find the optimal solution, we use dynamic programming, which runs in polynomial time.

Equation 11 shows that G_m is the set of m programs which are chosen from n programs, and c_i is the size of cache assigned to program i .

$$\begin{aligned} & \text{minimize} && \sum_{i=0}^{i < m} wbc_i(c_i) \\ & c_0, c_1, \dots, c_{n-1}, i \in G_m \\ & \text{subject to:} && c = \sum_{i=0}^{i < n} c_i \end{aligned} \quad (11)$$

As Equation 12 shows, the optimal partition for m -program co-run group can be reduced to that of $(m - 1)$ -program co-run group by checking all the possible partitions for $(m - 1)$ -program group and one new addition j . By divide and conquer, the m -program co-run problem can be reduced to single program solo-run base case.

Figure 2: Solution space for choosing m out of n programs to co-run with p -way cache

$$opt(m, n, c) = \min_{i=1 \dots p-m+1} (opt(m-1, n, c \times \frac{p-i}{p}) + wbc_j(c \times \frac{i}{p})) \quad (12)$$

In the evaluation, we consider only optimal cache partitioning for a given set of m programs. This is a special case of the joint optimization and will be solved by the method just presented, by setting $n = m$.

3.3 Joint Locality Optimization

Locality now has two aspects: the access locality, which is the locality of memory accesses that determines the miss ratio; and the write locality, which is the locality of memory writes and determines the writeback ratio. Two questions arise concerning the relation between the two. The first is whether they actually differ, that is, whether the maximizing access locality also maximizes write locality. If they differ, the second question is how to optimize the two locality types together. We address both problems with baseline optimization.

We draw an analogy with the prior work on balancing performance and fairness in shared cache. Kim et al. gave a set of fairness conditions such as equal slowdown and relative miss ratio increase [9]. The conditions are precise but do not allow optimization. Using the concepts of game theory, Zahedi and Lee defined the conditions called sharing incentive, envy freeness and Pareto efficiency [26]. Formulating the sharing incentive as a baseline constraint, Brock et al. defined *baseline optimization* [1] and later added *elasticity* [24].

We optimize joint locality by adapting the elastic baseline optimization by Ye et al. [24] First, we select a baseline, for example the miss ratio of equally partitioned cache. Then we choose an elasticity threshold, e.g. 5%. Then we find the cache partition with the lowest writebacks, under the constraint that the miss ratio does not increase by more than 5% from the baseline. In addition to the fair cache, the baseline can also be the miss ratio of the shared cache. The threshold may be 0%, that is, no elasticity, which means that it can remove writebacks only if doing so does not add

cache misses.

The optimization algorithm in Section 3.2 can be adapted by adding the baseline constraint as a filter to exclude candidate partitions that exceed the baseline. The modification is shown in Equation 13, where $c_{g,j}$ is the occupancy for program j of co-run group g running on shared cache with size c .

$$\begin{aligned} \text{opt}(m, n, c) = & \min_{i,j} \left(\text{opt}(m-1, n, c \times \frac{p-i}{p}) + \text{wbc}_j(c \times \frac{i}{p}) \right) \\ \text{subject to: } & 1 \leq i \leq p - m + 1 \\ & \text{mr}_j(c \times \frac{i}{p}) < \text{MRIC} \times \text{mr}_j(c_{g,j}) \end{aligned} \quad (13)$$

Baseline optimization directly solves the second problem, which is how to optimize read and write locality together. Indirectly, its results will answer the first question, whether read and write locality differs. If a higher elasticity on the miss ratio leads to a reduction in write ratio, then this means that we can increase write locality at the expense of decreasing access locality, and hence the two locality types are different.

4. EVALUATION

Methodology.

We first evaluate the accuracy of write locality in predicting writeback ratio in solo-run tests. Then we test write-locality optimization and joint optimization and compare then with shared cache and fair cache. Finally, we analyze the effect of optimization on individual programs.

We have implemented the online algorithm described in Algorithm 1 with the Pin tool. We use SPEC CPU2006 benchmarks. The same input, *reference*, is used in both profiling and testing. We do not use *dealIII* and *specrand.f*. *dealIII* could not be successfully compiled¹. *specrand.f* is the floating-point version of *specrand.i* (an integer random generator), their operations are exactly the same, so it is better not to duplicate. The remaining 29 programs are all used.

When measuring the write locality, the most time consuming steps are obtaining *wrt* histogram and footprint. For footprint, sampling has been used, and the average overhead for these programs was 0.1 second per program. This low overhead is due to adaptive bursty sampling [22]. For *wrt*, we use full trace profiling result to avoid sampling errors. The overhead is high, and for a single program, the current implementation takes 4–6 days. However, the profiling can be done in parallel, and the cost can be reduced by large factors by sampling.

4.1 Write Locality of SPEC 2006 Programs

With this metric, we calculated the all window write back ratio curves for each of the programs in SPEC with thousands of cache sizes between 0 to 32M². Figure 3 shows the write back ratio of each of the 29 programs in SPEC as a curve.

¹we are using SPEC CPU2006 1.1

²The number of cache sizes depends on the reuse-time histogram used in footprint analysis. The histogram is the same as in Xiang et al., which gives the miss ratio of 3073 cache sizes between 16KB and 64KB [23].

Figure 3: Write back ratio curve

We roughly classified them into 3 categories: *cache-size insensitive*, marked by red, are the programs whose write back ratio changes very little with the cache size; *cache-size oversensitive*, marked by green, are the programs whose write back ratio quickly drops to 0 at small cache sizes; *cache-size sensitive*, marked by blue, are the programs whose write back ratio drops to 0 slowly or in stages. These programs are:

- **cache-size oversensitive:** *perlbench*, *specrand*, *povray*, *libquantum*, *calculix*, *Xalan*, *hammer*, *tonto*, *games*, *soplex*
- **cache-size sensitive:** *bzip2*, *cactusADM*, *gobmk*, *gcc*, *milc*, *leslie3d*, *h264ref*, *omnetpp*, *gromacs*, *astar*
- **cache-size insensitive:** *sjeng*, *bwaves*, *GemsFDTD*, *lbm*, *sphinx*, *namd*, *mcf*, *wrf*, *zeusmp*

The classification depends on the range of cache sizes we choose. For example, if we consider the cache sizes between 0 and 8M, some cache-size sensitive programs will become cache-size insensitive and some programs in cache-size oversensitive will change to cache-size sensitive.

The classification is useful in understanding the effect of write locality optimization. When it maximizes the total writebacks, it can reduce the cache occupancy for cache insensitive programs and give more space to their co-run peers, especially those that are cache-size oversensitive. For the cache-size sensitive programs, the optimization will try to choose just the cache size where the ratio drops the most.

4.2 Accuracy

We compare prediction with simulation. In this study, we do not consider the effect of associativity, so we simulate fully associative LRU cache. Instead of all sizes, we simulate and compare the accuracy for 4 cache sizes: 64KB, 256KB, 1MB, and 8MB.

The average absolute errors for write back ratio are 1.15%, 0.88%, 0.63% and 0.27% respectively as shown in Table 4. The write back ratio ranges from 0 to 100% (Section 2.5). Programs *h264ref*, *gobmk*, *lbm*, *leslie3d*, *cactusADM* have the highest accuracy, and the average relative error (RE) is

around 3.45% in all four cache sizes. Programs *specrand*, *libquantum*, *astar*, *gamess* are least precise, and the average RE is around 60%. Most of the imprecise programs are cache-size oversensitive programs. With a small cache size, the write back ratio will be 0. For other, cache-size sensitive and insensitive programs (S & I), the relatively large errors can be tolerated. The table shows the absolute and relative errors for S & I programs. RE drops from as high as 48% to as low as 25%.

Table 4: Prediction errors

Cache Size	64KB	256KB	1MB	8MB
Absolute Error (AE)	1.15%	0.88%	0.63%	0.27%
AE (S & I)	0.69%	0.66%	0.54%	0.22%
Relative Error (RE)	43.91%	44.93%	37.26%	47.6%
RE (S & I)	25.2%	28.7%	26%	28.1%

4.3 Minimizing Co-run Writebacks

We consider *all* co-run groups of size 2, 3, and 4. Out of 29 programs, the number of such groups is 406, 3,654, and 23,751 respectively. The shared cache has a total size of 8MB and is partitioned at the granularity of 256KB blocks.

We evaluate the following types of cache sharing methods:

- *Shared Cache*: Cache is shared without any partitioning.
- *Fair Cache*: Cache is equally partitioned for co-run programs.
- *Optimal Partition-Sharing (OPS)*: Cache is optimally partitioned according to the algorithm in Section 3.2.
- *Elastic Baseline (EB) Optimization*: Cache is optimally partitioned according to the algorithm in Section 3.2.

In the following, OPS is treated as a special case of EB optimization. It is EB with infinite elasticity.

4.3.1 EB Optimized vs. Shared and Fair Cache

We first compare EB optimized cache and shared cache. Table 5 shows the average writeback ratio reduction by the optimization for the three co-run tests for different elasticity thresholds for 0 to ∞ . The table has two sections to show all the thresholds. The top section shows thresholds in linear increments from 0% to 100%, and the lower section shows thresholds in exponential increments from 1% to 100x and then ∞ .

The last row in the lower section shows the effect of OPS (∞ threshold). OPS provides the greatest possible reduction, which is 12.22% writebacks on average for 2-program co-run groups, 27.49% writebacks for 3-program co-run groups and 35.57% writebacks for 4-program co-run groups. As the size of co-run group grows, so is the number of possible cache partitionings and hence is reduction in writeback ratio.

OPS does not bound the increase of the miss ratio when optimizing the write locality. The effect of such a bound is shown by other, finite elasticity thresholds. The upper section shows the effect of increasing elasticity in linear scale. The reduction hardly changes from 20% to 100%. The difference is less than 0.1%, often less than 0.01%. This shows

Table 5: Average writeback reduction (%) by elastic baseline optimization over shared cache

elasticity	0%	20%	40%	60%	80%	100%
2-prog.	4.09	9.35	9.59	9.79	9.92	9.95
3-prog.	23.97	26.26	26.43	26.53	26.58	26.61
4-prog.	33.93	35.08	35.15	35.19	35.22	35.24
elasticity	1%	10%	100%	10x	100x	∞
2-prog.	6.42	8.93	9.95	10.45	10.8	12.22
3-prog.	25.63	26.31	26.61	26.98	27.21	27.49
4-prog.	34.49	35.01	35.24	35.42	35.45	35.57

that a linear increase of elasticity does not increase the ability of baseline optimization. The lower section shows the effect of increasing elasticity exponentially. The baseline optimization is more effective at (exponentially) greater elasticity. 0% is the strictest baseline. When the optimization does not increase the miss ratio compared to shared cache, it still reduces the writeback ratio by 4%, 24% and 34%.

For 4-program groups, infinite elasticity barely improves the optimization compared to no elasticity. The writebacks are reduced by 34.49% when allowing no elasticity. The reduction is 1.08% more with infinite elasticity. We can say that in this case, there is no real difference between access locality and write locality. Optimizing one means optimizing both.

Table 6 compares EB optimized cache with fair cache. Since linear increase of elasticity has little impact, the table shows increasing elasticity in an exponential scale. The writeback reduction is higher with fair cache than with shared cache. At infinite elasticity, OPS reduces the writebacks by 25%, 34% and 46% compared to fair cache, higher than the reductions, 12%, 27% and 36% for shared cache. One may calculate the difference between shared cache and fair cache in write locality from these numbers.

Table 6: Average writeback reduction (%) by elastic baseline optimization over fair cache

elasticity	0%	1%	10%	100%	10x	100x	∞
2-prog.	14.72	15.78	18.19	21.08	21.53	21.71	24.96
3-prog.	31.20	31.79	32.41	32.87	33.10	33.23	34.31
4-prog.	44.57	44.86	45.19	45.56	45.59	45.59	45.77

As mentioned earlier, the number of co-run groups is 406, 3654 and 23751 for 2-, 3- and 4-program co-runs. So far in the tables, we have evaluated the average reduction of all the groups. The reduction differs from group to group, which we examine next.

The writeback reduction ratios of all 406, 3654 and 23751 groups are plotted for 2, 3 and 4 co-run groups in Figure 4. In all plots, the co-run groups are sorted by the writeback reduction in increasing order. The x -axis is the group ids and the y -axis is the writeback reduction ratio. Different lines shows the reductions of different elasticity.

The line plots of Figure 4 confirm the observation and analysis earlier. The baseline optimization is more effective

Figure 4: Writeback reduction by optimization over shared cache for all co-run groups, sorted by increasing reduction ratios

at exponentially greater elasticity (large gaps between lines in (d,e)) but not at linearly increasing elasticity (clustered lines in plots (a,b)). The improvement is significant for 2-program co-runs but largely diminishes for 4-program co-runs, shown by the clustered lines in (c,f).

Figure 4 shows that the writeback reduction is over 80% for many groups and in some groups even approach 100%. We have analyzed these groups and found that they are more likely to contain at least one cache sensitive program. The program has limited cache resources when co-running on shared cache, but EB optimization allocates sufficient cache for it to reach the sharp drop in the writeback ratio curve. Furthermore, the number of groups with large reductions, e.g. 80%, grows rapidly when more programs co-run together. For 2 program co-run with OPS, about 17% of the groups have a reduction ratio more than 20%. When the number of co-run programs is 3 and 4, the percentage of the groups which have above 20% ratio increases to 40% and 60%.

4.4 Optimization Effect on Individual Programs

While the optimization reduces the total writebacks for a co-run group, not every program in the group may benefit from the optimization. It is possible that as a result of optimization, some programs will be caged in a very small number of cache blocks in order to minimize total writebacks of the group. This seems unfair but elastic baseline optimization can let us limit the degradation of individual performance as it has to bound the miss ratio increase.

For each program in all its co-run groups, we analyzed the average writeback reduction and cache occupancy increase relative to fair cache. The writeback reduction $owbr$ and cache occupancy increment ocr for each program i are cal-

culated by Equation 14. G_i is the set of program groups which contains the program (i).

$owbr$ does not only indicate that whether on average the program will be benefit (decrease the writebacks) or sacrificed (increase the writebacks) when co-running on OPS cache. As the base is set to the overall writebacks for groups, we can also tell how much this program will contribute to the group reduction. Same applies to ocr . We can tell the cache occupancy for programs when the base cache management policy is fair cache.

$$owbr^i = \text{avg}_{G \in G_i} \left(\frac{wbc_{opt}^i - wbc_{base}^i}{\sum_{j \in G} wbc_{base}^j} \right)$$

$$ocr^i = \text{avg}_{G \in G_i} \left(\frac{c_{opt}^i - c_{base}^i}{\text{total cache size}} \right) \quad (14)$$

Figure 5 shows the overall write back reduction and cache occupancy increase for each program in 2, 3, 4-program co-run groups with different elasticity. The bars shows the write back reduction, and the lines shows the cache occupancy increase. As the Figure 3 shows, writeback will not decrease when the cache size increases. For each program, when the writebacks are reduced (bars blow 0), the cache occupancy is expected to increase (lines above 0). Most of the programs match with this pattern, but for *povray* and *astar*, both the writebacks and cache occupancy decreases. This is because both $owbr$ and ocr are average results. For program groups which contains *povray* (or *astar*), *povray* (or *astar*) appears to be either decrease writebacks with higher ratio and increase the cache occupancy with lower ratio or increase the writebacks with lower ratio and decrease the cache occupancy with higher ratio in most cases. So on av-

erage, both the overall writebacks are reduced and the cache occupancy is decreased.

With EB optimization, *leslie3d* is reduced by almost 80% of the writebacks. *lbm*, *bzip2*, *gcc*, *omnetpp*, *gromacs* and *astar* are reduced almost 20% of the writebacks. 13 of the rest programs increase the writebacks, but the increase is small. For the cache occupancy, the programs with a higher writeback reduction are more likely to gain more cache space under optimization. *perlbench*, *specrand*, *libquantum* on average are giving around 40% of the total cache size to the other co-run peers. On the other hand, the programs with a high writeback reduction are more likely to gain around 30% of the cache from co-run peers.

5. RELATED WORK

Writeback modeling: Kant proposed a simple Markovian model to estimate invalidation traffic and writeback traffic for symmetric multiprocessor system [8]. The solution is statistical and may not be accurate. Our solution is computational and computes the locality based on actual access traces.

Writeback reduction: Current writeback reduction techniques are mainly focused on the hardware designs. Shi et al. introduced a smart victim cache to reduce the writebacks to flash memory [16]. The smart victim cache only stores the evicted dirty cache lines and increases the probability to combine with a later write. Zhou et al. used a combination of techniques to reduce and distribute writebacks: redundant writes removal, byte shifting and swapping [28]. For every NVM row write, they issued a preceding read to the same row and used an addition XOR logic to get the modified portion of the row. Only the modified bits were written to the row instead of the entire row.

cache partition: Cache partitioning may be done by hardware or software. For hardware partitioning, way partitioning [20, 14, 3] and block partitioning [19] are some of the approaches. Intel also added support to some of their products with cache allocation technology to support way partitioning. For software partition, Page coloring (e.g. [25]) is often employed. The mapping between physical address space to cache address space are used to decide the color bits. Traditional cache partitioning techniques are mainly focused on performance. Brock et al. investigated optimal cache partition-sharing [1]. In this paper, we focus on reducing the cache writebacks.

Set-Associative Cache: A design in CPU cache is limited associativity. Smith gave a probabilistic formula that estimates cache conflicts based on the reuse distance [17]. The Smith formula becomes standard in later uses [6, 10, 12]. Victim footprint can use the Smith formula to model set-associative cache, because victim footprint can be used to compute the reuse distance in the same way that footprint can using the HOTL theory [23]. Modern processor design is effective in uniformly distributing data to cache sets, for example, hash-based caches on Intel’s SandyBridge architecture. Under uniform mapping, the probabilistic formula of Smith is accurate [17].

The key idea in the Smith formula is to take a reuse distance and model the distribution of the data into cache sets. The model assumes that the data maps to all cache sets uniformly. This limitation is recently removed. Sen and Wood [15] and Nugteren et al. [11] gave an accurate solution, which measures the reuse distance of the access stream

in each cache set as the *per-set locality* [15]. The precise solution supports all associativity levels but requires that the number of sets in the cache be fixed.

The write locality in this paper assumes full associativity and ignores the effect of set-associativity. On CPU caches, conflict misses are not significant because of the high associativity. Cantin and Hill showed that 8-way or higher associativity cache has effectively the same performance as fully associative cache [2]. When footprint was used to model CPU cache in two recent studies, the authors did not find associativity a significant factor in performance [23, 22].

6. SUMMARY

In this paper, we have presented formal definitions of write locality including write reuse distance and write reuse time. We have presented techniques to predict the writeback frequency for a program for all cache sizes. For co-run shared cache, we give a compositional analysis that computes the effect of cache sharing without co-run testing. By testing on SPEC CPU2006 benchmarks, we show that the new model is accurate. The absolute error is about 1% and the relative error is around 20% for the program with a high writeback ratio. We have analyzed 29 programs in SPEC CPU2006 and classified them into three categories.

We have developed a new algorithm that jointly optimizes program symbiosis and cache partitioning in shared cache and adapted the new algorithm to optimize both access locality and write locality. In evaluation, we have tested all 2-, 3- and 4-program co-run groups. Several observations are made from the results: (1) When more programs co-run together, we can reduce more writebacks in shared cache by optimization; (2) When more program co-run together, the optimization is more likely to minimize the number of writebacks without increasing the miss ratios of the group; and (3) access locality differs from write locality, but the difference quickly diminishes when the size of the co-run group increases to four. In addition to average group writeback reduction, we also evaluated the effect of optimization on individual programs.

We expect that the new model and algorithms are also effective to guide page allocation / placement for hybrid memory system at the level of the DRAM cache. The new model can also be used to estimate the write locality for individual data structures and enable program-level optimization of write locality.

7. ACKNOWLEDGMENTS

The authors would like to thank Hao Luo, Pengcheng Li, Jacob Brock and Rahman Lavaee for their comments which have helped this work. The on-going research is supported by the National Science Foundation (Contract No. CCF-1629376, CNS-1319617), IBM CAS Faculty Fellowship, the National Science Foundation of China (Contract No. 61328201) and a grant from Huawei. 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.

8. REFERENCES

- [1] J. Brock, C. Ye, C. Ding, Y. Li, X. Wang, and Y. Luo. Optimal cache partition-sharing. In *Parallel*

(a) 2-program co-run

(b) 3-program co-run

(c) 4-program co-run

Figure 5: Writeback reduction (negative means reduction) and cache occupancy increase (positive means increase) for EB optimized cache vs. fair cache

- Processing (ICPP), 2015 44th International Conference on*, pages 749–758. IEEE, 2015.
- [2] J. F. Cantin and M. D. Hill. Cache performance for SPEC CPU2000 benchmarks. <http://www.cs.wisc.edu/multifacet/misc/spec2000cache-data>.
- [3] D. Chiou, P. Jain, L. Rudolph, and S. Devadas. Application-specific memory management for embedded systems using software-controlled caches. In *Proceedings of the 37th Annual Design Automation Conference*, pages 416–419. ACM, 2000.
- [4] P. J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, 1968.
- [5] C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. In *ACM SIGPLAN Notices*, volume 38, pages 245–257. ACM, 2003.
- [6] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*,

38(12):1612–1630, 1989.

- [7] Intel. Cache monitoring technology and cache allocation technology.
- [8] K. Kant. Estimation of invalidation and writeback rates in multiple processor systems. *Submitted for publication*, 2000.
- [9] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of PACT*, pages 111–122, 2004.
- [10] G. Marin and J. Mellor-Crummey. Scalable cross-architecture predictions of memory hierarchy response for scientific applications. In *Proceedings of the Symposium of the Las Alamos Computer Science Institute*, Sante Fe, New Mexico, 2005.
- [11] C. Nugteren, G. van den Braak, H. Corporaal, and H. E. Bal. A detailed GPU cache model based on reuse distance theory. In *Proceedings of HPCA*, 2014.
- [12] A. Qasem and K. Kennedy. A cache-conscious profitability model for empirical tuning of loop fusion. In *Proceedings of the LCPC Workshop*, pages 106–120, 2005.
- [13] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. *ACM SIGARCH Computer Architecture News*, 37(3):24–33, 2009.
- [14] P. Ranganathan, S. Adve, and N. P. Jouppi. *Reconfigurable caches and their application to media processing*, volume 28. ACM, 2000.
- [15] R. Sen and D. A. Wood. Reuse-based online models for caches. In *Proceedings of SIGMETRICS*, pages 279–292, 2013.
- [16] L. Shi, C. J. Xue, J. Hu, W.-C. Tseng, X. Zhou, and E. H.-M. Sha. Write activity reduction on flash main memory via smart victim cache. In *Proceedings of the 20th symposium on Great lakes symposium on VLSI*, pages 91–94. ACM, 2010.
- [17] A. J. Smith. On the effectiveness of set associative page mapping and its applications in main memory management. In *Proceedings of ICSE*, 1976.
- [18] C. Su, D. Roberts, E. A. León, K. W. Cameron, B. R. de Supinski, G. H. Loh, and D. S. Nikolopoulos. Hpmc: An energy-aware management system of multi-level memory architectures. In *Proceedings of the 2015 International Symposium on Memory Systems*, pages 167–178. ACM, 2015.
- [19] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic cache partitioning for simultaneous multithreading systems. In *Proceeding of the Iasted international conference on parallel and distributed computing and systems*, pages 116–127, 2001.
- [20] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 28(1):7–26, 2004.
- [21] X. Wang, Y. Li, Y. Luo, X. Hu, J. Brock, C. Ding, and Z. Wang. Optimal footprint symbiosis in shared cache. In *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, pages 412–422. IEEE, 2015.
- [22] Wang et al. Optimal program symbiosis in shared cache. In *Proceedings of CCGrid*, June 2015.
- [23] X. Xiang, C. Ding, H. Luo, and B. Bao. HOTL: a higher order theory of locality. In *Proceedings of ASPLOS*, pages 343–356, 2013.
- [24] C. Ye, J. Brock, C. Ding, and H. Jin. Recu: Rochester elastic cache utility. *Proceedings of NPC*, 2015.
- [25] Y. Ye, R. West, Z. Cheng, and Y. Li. Coloris: a dynamic cache partitioning system using page coloring. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 381–392. ACM, 2014.
- [26] S. M. Zahedi and B. C. Lee. REF: resource elasticity fairness with sharing incentives for multiprocessors. In *Proceedings of ASPLOS*, pages 145–160, 2014.
- [27] W. Zhang and T. Li. Exploring phase change memory and 3d die-stacking for power/thermal friendly, fast and durable memory architectures. In *Parallel Architectures and Compilation Techniques, 2009. PACT’09. 18th International Conference on*, pages 101–112. IEEE, 2009.
- [28] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. In *ACM SIGARCH computer architecture news*, volume 37, pages 14–23. ACM, 2009.