Appears in *Proc. of the 18th Int'l Conf. on Parallel Architectures and Compilation Techniques*. Raleigh, NC. Sept. 2009.

Using Aggressor Thread Information to Improve Shared Cache Management for CMPs

Wanli Liu and Donald Yeung
Department of Electrical and Computer Engineering
Institute for Advanced Computer Studies
University of Maryland at College Park
{*wanli,yeung*}*@eng.umd.edu*

***Abstract*—Shared cache allocation policies play an important role in determining CMP performance. The simplest policy, LRU, allocates cache implicitly as a consequence of its replacement decisions. But under high cache interference, LRU performs poorly because some memory-intensive threads, or *aggressor threads*, allocate cache that could be more gainfully used by other (less memory-intensive) threads. Techniques like cache partitioning can address this problem by performing explicit allocation to prevent aggressor threads from taking over the cache.**

Whether implicit or explicit, the key factor controlling cache allocation is *victim thread selection*. The choice of victim thread relative to the cache-missing thread determines each cache miss's impact on cache allocation: if the two are the same, allocation doesn't change, but if the two are different, then one cache block shifts from the victim thread to the cache-missing thread. In this paper, we study an omniscient policy, called ORACLE-VT, that uses off-line information to always select the best victim thread, and hence, maintain the best per-thread cache allocation at all times. We analyze ORACLE-VT, and find it victimizes aggressor threads about 80% of the time. To see if we can approximate ORACLE-VT, we develop AGGRESSOR-VT, a policy that probabilistically victimizes aggressor threads with strong bias. Our results show AGGRESSOR-VT comes close to ORACLE-VT's miss rate, achieving three-quarters of its gain over LRU and roughly half of its gain over an ideal cache partitioning technique.

To make AGGRESSOR-VT feasible for real systems, we develop a sampling algorithm that "learns" the identity of aggressor threads via runtime performance feedback. We also modify AGGRESSOR-VT to permit adjusting the probability for victimizing aggressor threads, and use our sampling algorithm to learn the per-thread victimization probabilities that optimize system performance (*e.g.*, weighted IPC). We call this policy AGGRESSORpr-VT. Our results show AGGRESSORpr-VT outperforms LRU, UCP [1], and an ideal cache way partitioning technique by 4.86%, 3.15%, and 1.09%, respectively.

***Keywords*-shared cache management; cache partitioning; memory interleaving; aggressor thread**

This research was supported in part by NSF CAREER Award #CCR-0093110, and in part by the Defense Advanced Research Projects Agency (DARPA) through the Department of the Interior National Business Center under grant #NBCH104009. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsement, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

I. Introduction

CMPs allow threads to share portions of the on-chip memory hierarchy, especially the lowest level of on-chip cache. Effectively allocating shared cache resources to threads is crucial for achieving high performance.

The simplest policy is to allocate cache implicitly as a consequence of the cache's default replacement policy–*e.g.*, LRU. This approach works well as long as per-thread working sets can co-exist symbiotically in the cache [2], [3]. Under *low cache interference*, the LRU policy allows individual threads to freely use the aggregate cache capacity in a profitable fashion. In contrast, when per-thread working sets conflict in the cache, LRU degrades performance because it allows certain memory-intensive threads to allocate cache that could be more gainfully used by other (less memory-intensive) threads. In this paper, we call such threads "aggressor threads." Under *high cache interference*, explicit cache allocation is needed to keep aggressor threads from detrimentally starving other threads of cache resources.

Prior research has investigated explicit cache allocation techniques–most notably, *cache partitioning* [2], [4], [5], [1], [6], [7], [8], [9], [10]. Cache partitioning explicitly allocates a portion of the shared cache to individual threads, typically in increments of cache ways. The per-thread partitions isolate interfering working sets, thus guaranteeing resources to threads that would otherwise be pushed out of the cache. Researchers have also proposed techniques that adapt to different levels of interference by switching between partition-like and LRU-like allocation [2], [11].

Whether implicit or explicit, the key factor controlling cache allocation is *victim thread selection*. On a cache miss, the cache must select a victim thread from which a cache block will be replaced. Then, from the selected thread's pool of resident cache blocks, a replacement can be made. The choice of victim thread relative to the cache-missing thread determines the cache miss's incremental impact on cache allocation: if the two threads are the same, cache allocation doesn't change, but if the two threads differ, then one cache block shifts from the victim thread to the cache-missing thread. Over time, the sequence of selected victim threads determines how much cache each thread receives.

In this paper, we study an omniscient cache allocation policy, called ORACLE-VT. ORACLE-VT uses an on-line policy (LRU) to identify replacement candidates local to each thread. However, ORACLE-VT uses off-line information to select victim threads by identifying the per-thread local LRU block that is referenced furthest in the future. Since this block is the most profitable block to replace, the block's owner is the ideal victim thread choice. Because ORACLE-VT selects victim threads perfectly, it maintains the best per-thread cache allocation at all times (*i.e.*, after every cache miss). Across a suite of 216 2-thread workloads and 13 4-thread workloads, we find ORACLE-VT achieves a 3% miss rate improvement compared to an ideal cache way partitioning technique.

We analyze ORACLE-VT's victim thread decisions, and find that under high cache interference, it usually victimizes aggressor threads. In particular, ORACLE-VT selects an aggressor thread for victimization roughly 80% of the time. To see if we can approximate ORACLE-VT's omniscient decisions without using future reuse distance information, we tried a victim thread selection policy that heavily biases victimization against aggressor threads. This policy, called AGGRESSOR-VT, victimizes an aggressor thread 100% of the time if it owns the most LRU cache block in the set, and 99% of the time if it does not. The other 1% of the time, AGGRESSOR-VT victimizes the "non-aggressor" owner of the set's most LRU block. The non-aggressor is also victimized by default if no block belonging to an aggressor thread exists in the cache set. Using trace-driven simulation, we show AGGRESSOR-VT comes within 1.7% and 1.0% of ORACLE-VT's miss rate for 2- and 4-thread workloads, respectively, and achieves roughly half of ORACLE-VT's benefit over ideal cache way partitioning.

To make AGGRESSOR-VT feasible for real systems, we must identify aggressor threads on-line. We develop a sampling algorithm that "learns" which threads are aggressors via runtime performance feedback. Another issue is ORACLE-VT, and hence AGGRESSOR-VT, optimizes the global cache miss rate, which may not always translate into improved system performance and/or fairness [5]. To address this problem, we modify AGGRESSOR-VT to provide flexibility in steering resources to different threads. Rather than always victimize aggressor threads, we include an intermediate probability, 50%, for selecting an aggressor vs non-aggressor, and permit separate threads to use different probabilities. Then, we use the same sampling algorithm to learn the per-thread victimization probabilities that optimize system performance (*e.g.*, weighted IPC). We call this policy AGGRESSOR*pr*-VT. Our results show AGGRESSOR*pr*-VT outperforms LRU, UCP [1], and ideal cache way partitioning (in terms of weighted IPC) by 4.86%, 3.15%, and 1.09%, respectively.

The remainder of this paper is organized as follows. Section II discusses the role of memory reference interleaving in determining cache interference, and how it impacts the efficacy of different allocation policies. Section III presents our ORACLE-VT study and its approximation using AGGRESSOR-VT. Next, Section IV presents AGGRESSOR*pr*-VT, and evaluates its performance. Finally, Sections V and VI discuss prior work and conclusions.

II. CACHE INTERFERENCE

Cache interference arises in CMPs with shared caches because threads bring portions of their working sets into the shared cache simultaneously. A critical factor that determines the severity of this interference is the *granularity of memory reference interleaving*. Section II-A discusses how interleaving granularity gives rise to different degrees of cache interference, and how they are best addressed using different cache allocation techniques. Then, Section II-B studies interleaving granularity quantitatively.

A. Interleaving Granularity

Figure 1 illustrates interference in a shared cache due to memory reference interleaving. In Figure 1, program 1 references memory locations A-C and then reuses them, while program 2 does the same with locations X-Z. Assume a fully associative cache with a capacity of 4 and an LRU replacement policy. For the reuse references to hit in the cache, each program must receive at least 3 cache blocks (*i.e.*, the intra-thread stack distance is 3). Suppose the programs run simultaneously, and their memory references interleave in a *fine-grain manner*, as shown in Figure 1. Then, the cache capacity is divided amongst the two programs as a consequence of the LRU replacements. (The numbers in Figure 1 report the per-program cache allocation after each memory reference). Due to the memory interleaving, each reuse reference's stack distance increases to at least 5, so the LRU policy is unable to provide sufficient resources to each program at the time of reuse. Instead, programs replace each other's blocks (the asterisks in Figure 1 indicate references causing inter-thread replacements), and all reuse references miss in the cache.

As Figure 1 shows, fine-grain reference interleaving results in high cache interference, and requires explicit cache allocation to prevent inter-thread replacements from degrading intra-thread locality. For example, cache partitioning can be used to guarantee cache resources to programs. In Figure 1, cache partitioning provides one program with a partition of 3 cache blocks, and the other with a partition of 1 cache block. This guarantees one program enough resources to exploit its reuse despite the fine-grain interleaving. Although the other reuse references will still miss in the cache, performance improves overall due to the additional cache hits. In this case, explicit cache allocation is necessary and improves performance because the programs exhibit actual cache interference.

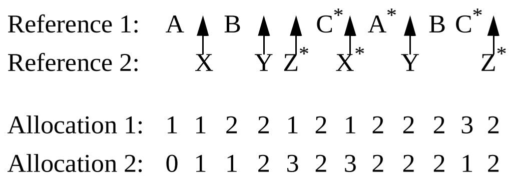
Reference 1: A B C* A* B C*
Reference 2: X Y Z* X* Y Z*
Allocation 1: 1 1 2 2 1 2 1 2 2 2 3 2
Allocation 2: 0 1 1 2 3 2 3 2 2 2 1 2

Figure 1. Fine-grain interleaving. Numbers below the reference trace indicate per-thread cache allocation; asterisks indicate references causing inter-thread replacements.

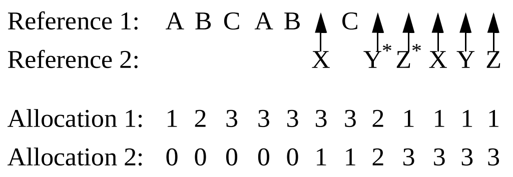
Reference 1: A B C A B C
Reference 2: X Y* Z* X Y Z
Allocation 1: 1 2 3 3 3 3 3 2 1 1 1 1
Allocation 2: 0 0 0 0 0 1 1 2 3 3 3 3

Figure 2. Coarse-grain interleaving. Format is identical to Figure 1.

Interestingly, explicit allocation via cache partitioning can actually degrade performance in the absence of fine-grain interleaving. Consider the same two programs again, but now assume the memory references interleave in a *coarse-grain manner*, as shown in Figure 2. This time, each program's inherent locality is only slightly impacted by the simultaneous execution–the reuse references' stack distances increase to at most 4. Consequently, all reuse references hit in the cache because the requisite cache capacity can be gainfully time-multiplexed between the two programs, as indicated in Figure 2 by the cache allocation counts and the lower frequency of inter-thread replacements. However, if we naively apply the 3-vs-1 partitioning suggested for fine-grain interleaving, one of the programs would be forced into the smaller partition of 1 cache block, and its reuse references would miss in the cache. These cache misses directly degrade performance since the other program receives no added benefit from its larger partition. In this case, explicit cache allocation, and in particular cache partitioning, is unnecessary–the cache interference it tries to mitigate doesn't occur.

This example illustrates the best cache allocation policy depends on how programs' memory references interleave at runtime. The finer-grained the interleaving, the more cache interference occurs and the more intra-thread locality is disrupted, increasing the importance of explicit cache allocation. The coarser-grained the interleaving, the less cache interference occurs and the more intra-thread locality remains intact, increasing the importance of flexible cache sharing provided by LRU.

B. Interleaving Measurements

Having discussed qualitatively the relationship between memory reference interleaving and cache interference, we now conduct studies that quantify the granularity of memory reference interleaving in actual multiprogrammed workloads. To do this, we observe the post-L1 *per-thread memory reference runlength–i.e.* the number of consecutive memory references a thread performs before another thread performs an interleaving reference–to indicate the interleaving granularity at the shared L2 cache of a multicore processor. Since references destined to distinct cache sets cannot interfere in the cache, we perform this analysis on a per cache set basis, and then average the observed per-set runlengths across the entire cache.

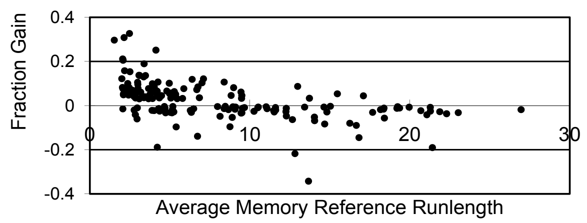

Figure 3. Partitioning's gain (as a fraction) over LRU versus average per-thread memory reference runlength.

Our study considers 216 2-thread workloads and 13 4-thread workloads consisting of SPEC CPU2000 benchmarks. We ran all multiprogrammed workloads on a cycle-accurate simulator of a multicore processor. For each workload, we acquired memory reference traces at the shared L2 cache (assuming an LRU policy) to enable our interleaving analysis. We also measured the workload's weighted IPC (WIPC) [12] when using LRU or cache partitioning to determine the technique under which the workload performs best. (See Section III for more details on our methodology).

Figure 3 presents the runlength and performance results. In Figure 3, each datapoint represents a single multiprogrammed workload. The datapoint's X-axis value is the workload's observed average per-thread memory reference runlength–*i.e.*, workloads appear from finest to coarsest interleaving granularity from left to right. The datapoint's Y-axis value is the workload's performance gain under cache partitioning compared to LRU–*i.e.*, workloads above the X-axis prefer cache partitioning while workloads below the X-axis prefer LRU. We will refer to these as *partitioning workloads* and *LRU workloads*, respectively.

Figure 3 shows a strong correlation between runlength and the preference for partitioning or LRU. At long runlengths (12 and beyond), LRU workloads dominate; at medium runlengths (from 5 to 11), there exists a mixture of partitioning and LRU workloads; and at short runlengths (below 5), partitioning workloads dominate. As explained in Section II-A, interleaving granularity determines cache interference, and in turn, the efficacy of different allocation policies. Our results confirm this insight.

But Figure 3 only tells part of the story. Because it plots the average runlength workload-wide, it does not show the *variation in runlength* that occurs within each workload. To provide more insight, we analyze our workloads' L2

	No Int	Coarse-Grain Int	Fine-Grain Int
LRU (2-thread)	56.2%	22.2%	21.6%
Partitioning (2-thread)	17.3%	10.7%	72.0%
Partitioning (4-thread)	3.9%	5.6%	90.5%

Table I
PERCENT MEMORY REFERENCES PERFORMED IN SETS WITH NO INTERLEAVING, COARSE-GRAIN INTERLEAVING, AND FINE-GRAIN INTERLEAVING FOR 2-/4-THREAD LRU/PARTITIONING WORKLOADS.

accesses, and identify memory reference runs of varying granularity across different sets in the cache. To perform this analysis, we first divide each workload's execution into fixed time intervals, called *epochs*, each lasting 1 million cycles. Then, we examine the memory references performed to each set within each epoch. If 100% of the memory references in a particular "set-epoch" is performed by a single thread, we say the references experience no interleaving. In the remaining set-epochs where interleaving occurs, we measure the per-thread runlength. For runlengths greater than 8, we say the associated memory references experience coarse-grain interleaving; otherwise, we say the memory references experience fine-grain interleaving. We choose 8 as a granularity threshold because it is in the middle of the "medium runlength" category, and roughly separates the LRU workloads from the partitioning workloads in Figure 3.

Table I reports the percentage of L2 memory references that occur in cache sets with no interleaving, coarse-grain interleaving, and fine-grain interleaving. The results are broken down for the LRU and partitioning workloads, and in the partitioning case, for the 2- and 4-thread workloads. (12 out of 13 4-thread workloads are partitioning workloads, so we excluded the LRU 4-thread case). In Table I, we see the 2-thread LRU workloads incur most of their references in sets with no interleaving or coarse-grain interleaving–56.2% and 22.2%, respectively. Furthermore, partitioning workloads incur most of their references in sets with fine-grain interleaving–72.0% and 90.5% for the 2- and 4-thread cases, respectively. In other words, the dominant interleaving granularity in each workload determines whether partitioning or LRU is preferred workload-wide, a result consistent with Figure 3. More interestingly, regardless of a workload's policy preference, there exists a non-trivial number of memory references participating in the other interleaving granularity categories. LRU workloads incur 21.6% of their references in sets with fine-grain interleaving, while partitioning workloads incur 17.3% and 3.9% of their references in sets with no interleaving and 22.2% and 5.6% in sets with coarse-grain interleaving for the 2- and 4-thread workloads, respectively. This result demonstrates interleaving granularity not only varies across different workloads, but it also varies across *cache sets within a single workload*.

III. IDEAL THREAD VICTIMIZATION

As discussed in Section II-A, under high cache interference, explicit cache allocation is needed to guarantee resources to threads that would otherwise be pushed out of the cache. An important question then is, what is the best policy for allocating cache to threads? To address this question, it is helpful to study victim thread selection. On each cache miss, the cache must select a victim thread from which to replace a cache block. The choice of victim thread determines the cache miss's impact on cache allocation: if the cache-missing thread itself is victimized, then cache allocation doesn't change; however, if a thread different from the cache-missing thread is victimized, then one cache block shifts from the victim thread to the cache-missing thread. Over time, the sequence of selected victim threads determines how much cache each thread receives.

To gain insight on how to effectively allocate cache to threads, we study an omniscient victim thread selection policy, called ORACLE-VT. ORACLE-VT uses LRU to replace cache blocks within a thread. However, to select victim threads, ORACLE-VT considers all per-thread LRU blocks, and uses off-line information to identify the cache block used furthest in the future. The thread owning this cache block is the victim thread. ORACLE-VT's thread victimization is inspired by Belady's MIN replacement algorithm [13]. Whereas Belady's algorithm uses future reuse information to select victim cache blocks within a thread, ORACLE-VT uses the same information to select victim threads within a workload.

Notice ORACLE-VT makes perfect victim thread decisions only; per-thread working sets are still managed (imperfectly) via LRU. Hence, ORACLE-VT allows us to study the performance impact of the cache allocation policy in isolation of other cache management effects. Moreover, and perhaps more important, by observing the omniscient decisions ORACLE-VT makes, we can acquire valuable insights for improving existing cache allocation techniques.

A. Experimental Methodology

Our study uses both event- and trace-driven simulation to compare ORACLE-VT against an ideal partitioning technique and LRU. This section discusses our methodology.

1) Simulator and Workloads: We use M5 [14], a cycle-accurate event-driven simulator, to quantify performance of cache partitioning and LRU, and for acquiring traces. In both cases, we configured M5 to model either a dual-core or quad-core system. Our cores are single-threaded 4-way out-of-order issue processors with an 128-entry RUU and a 64-entry LSQ. Each core also employs its own hybrid gshare/bimodal branch predictor. The on-chip memory hierarchy consists of private split L1 caches, each 32-Kbyte in size and 2-way set associative; the L1 caches are connected to a shared L2 cache that is 1-Mbyte (2-Mbytes) in size and 8-way (16-way) set associative for the dual-core (quad-core) system. When

Processor Parameters	
Bandwidth	4-Fetch, 4-Issue, 4-Commit
Queue size	32-IFQ, 80-Int IQ, 80-FP IQ, 256-LSQ
Rename reg / ROB	256-Int, 256-FP / 128 entry
Functional unit	6-Int Add, 3-Int Mul/Div, 4-Mem Port 3-FP Add, 3-FP Mul/Div
Memory Parameters	
IL1	32KB, 64B block, 2 way, 2 cycles
DL1	32KB, 64B block, 2 way, 2 cycles
UL2-2core	1MB, 64B block, 8 way, 10 cycles
UL2-4core	2MB, 64B block, 16 way, 10 cycles
Memory	200 cycles (6 cycle bw)
Branch Predictor	
Predictor	Hybrid 8192-entry gshare / 2048-entry Bimod / 8192-entry meta table
BTB/RAS	2048 4-way / 64

Table II
SIMULATOR PARAMETERS.

App	Type	Skip	App	Type	Skip
applu	FP	187.3B	mgrid	FP	135.2B
bzip2	Int	67.9B	swim	FP	20.2B
equake	FP	26.3B	wupwise	FP	272.1B
fma3d	FP	40.0B	eon	Int	7.8B
gap	Int	8.3B	perlbmk	Int	35.2B
lucas	FP	2.5B	crafty	Int	177.3B
mesa	FP	49.1B	ammp	FP	4.8B
apsi	FP	279.2B	sixtrack	FP	299.1B
art	FP	14B	twolf	Int	30.8B
facerec	FP	111.8B	vpr	Int	60.0B
galgel	FP	14B	gzip	Int	4.2B
gcc	Int	2.1B	vortex	Int	2.5B
mcf	Int	14.8B	parser	Int	66.3B

Table III
SPEC CPU2000 BENCHMARKS USED TO DRIVE OUR STUDY (B = BILLION).

acquiring traces, the L2 is managed using LRU. The latency to the L1s, L2, and main memory is 2, 10, and 200 cycles, respectively. Table II lists the detailed simulator parameters.

To drive our simulations, we used multiprogrammed workloads created from the complete set of 26 SPEC CPU2000 benchmarks shown in Table III. Many of our results are demonstrated on 2-program workloads: we formed all possible pairs of SPEC benchmarks–in total, 325 workloads. To verify our insights on larger systems, we also created 13 4-program workloads, which are listed in Table IV.[1] All of our benchmarks use the pre-compiled Alpha binaries (with highest level of compiler optimization) provided with the SimpleScalar tools [15],[2] and run using the SPEC reference input set.

We fast forward the benchmarks in each workload to

[1]In creating the 4-thread workloads, we ensured that each benchmark appears in 2 workloads.

[2]The binaries we use are available at http://www.simplescalar.com/benchmarks.html.

ammp-applu-gcc-wupwise	apsi-gcc-ammp-swim
apsi-bzip2-swim-vpr	eon-sixtrack-facerec-mgrid
art-vortex-facerec-fma3d	applu-fma3d-galgel-equake
crafty-parser-mgrid-twolf	bzip2-art-lucas-crafty
equake-galgel-mcf-sixtrack	gap-mesa-gzip-lucas
perlbmk-twolf-vortex-wupwise	gzip-mesa-parser-gap
eon-mcf-perlbmk-vpr	

Table IV
4-PROGRAM WORKLOADS USED IN THE EVALUATION.

their representative simulation regions; the amount of fast forwarding is reported in the columns labeled “Skip” of Table III. These were determined by SimPoint [16], and are posted on the SimPoint website.[3] After fast forwarding, detailed simulation is turned on. For performance measurements, we simulate in detailed mode for 500M cycles which yields over 1 and 2 billion instructions for the 2- and 4-program workloads, respectively. When acquiring traces, we simulate for a smaller window, 100M cycles, due to the large number of workloads we study and the significant disk storage their traces consume.

2) Cache Allocation Techniques: We investigate using ORACLE-VT, cache partitioning, and LRU to manage the shared L2 cache of our multicore processor. ORACLE-VT is studied using trace-driven simulation only. After acquiring traces as described in Section III-A, we replay them on a cache simulator that models ORACLE-VT. During trace-driven simulation, we make the entire trace available to the cache model so that it can determine future reuse information at all times, as needed by ORACLE-VT.

For partitioning, we consider an ideal cache way partitioning technique, which we call iPART. We study iPART using both event-driven and trace-driven simulation. iPART performs partitioning dynamically, as shown in Figure 4. CMP execution is divided into fixed time intervals, called *epochs*. We use an epoch size of 1 million cycles, which is comparable to other studies of dynamic partitioners. At the beginning of each epoch, a partitioning of the cache is selected across threads. iPART performs way partitioning [1], [10], so each thread is allocated some number of cache ways. During execution in the epoch, the cache replacement logic enforces the way partitioning. This repeats for every epoch until the workload is completed.

iPART is an ideal technique–it omnisciently selects the best partitioning every epoch. To do this, we checkpoint the simulator state at the beginning of each epoch (*i.e.*, the entire CMP state for event-driven simulations, or the L2 cache state for trace-driven simulations). From the checkpoint, we simulate every possible partitioning of the cache for 1 epoch. Before each exhaustive trial, the simulator rolls back to the checkpoint so that every trial begins from the

[3]Simulation regions we use are published at http://www-cse.ucsd.edu/ calder/simpoint/multiple-standard-simpoints.htm.

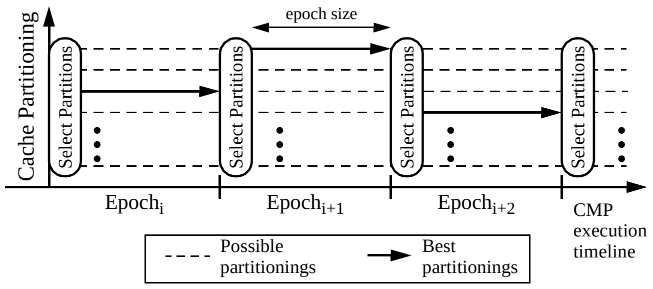

Figure 4. Epoch-based ideal cache way partitioning.

same architectural point. After trying all partitionings, the best one is identified, and the simulator advances to the next epoch using this partitioning. The best partitioning for event-driven simulations is the one that achieves the highest WIPC, while for trace-driven simulations, it's the one that achieves the highest hit rate. At the end of the simulation, the cumulative statistics (either WIPC or cache hits) across all the best partitionings is the performance achieved by the workload, while the overhead for all the exhaustive trials is ignored. iPART represents an upper bound on cache partitioning performance.

Due to exhaustive search, iPART simulation is expensive, especially for large numbers of threads. In fact, it is intractable for event-driven simulation of 4-thread workloads. In Section IV, we will measure cache partitioning's WIPC for 4-thread workloads using a different approach.

B. ORACLE-VT Insights

We begin by comparing iPART and LRU's WIPC achieved in our performance simulations to identify workloads that require explicit and implicit cache allocation. (These same results were used in Figure 3). Figure 5 shows the analysis for our 2-thread workloads. Out of the 325 2-thread workloads, 109 do not show any appreciable performance difference ($\leq$ 1%) between iPART and LRU, as reported by the "PART = LRU" bar. In these workloads, the allocation policy is irrelevant, usually because the programs' working sets fit in cache. Of the remaining 216 "policy-sensitive" workloads, 154 perform best under iPART while 62 perform best under LRU, as indicated by the second and third bars in Figure 5. For 4-thread workloads, we find all are policy-sensitive, with 12 performing best under partitioning, and only one performing best under LRU. (Essentially, all of them are partitioning workloads).

We also compare iPART and LRU in terms of miss rate. Figure 6 reports the two policies' L2 cache miss rates achieved in our trace-driven simulations, normalized to LRU. The results are broken down for the 2- and 4-thread workloads, and among 2-thread workloads, for the partitioning and LRU workloads identified in Figure 5. As expected, Figure 6 shows iPART achieves a better miss rate than LRU for the partitioning workloads (4.0% and

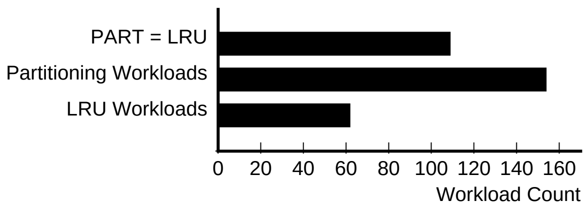

Figure 5. LRU and partitioning workload breakdown for 2-thread workloads.

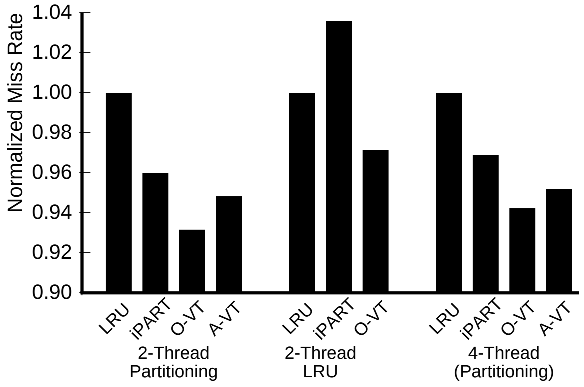

Figure 6. LRU, iPART, ORACLE-VT (O-VT), and AGGRESSOR-VT (A-VT) miss rate normalized to LRU.

3.1% better for the 2- and 4-thread workloads, respectively), while LRU achieves a better miss rate than iPART for the LRU workloads (3.6% better for the 2-thread workloads). As discussed in Section II-A, partitioning workloads exhibit fine-grain interleaving and high cache interference, requiring explicit cache allocation to perform well. In contrast, LRU workloads exhibit coarse-grain interleaving and low cache interference, requiring implicit cache allocation to permit flexible cache sharing.

Having identified the partitioning and LRU workloads, we now study ORACLE-VT. In Figure 6, the bars labeled "O-VT" report the cache miss rates achieved by ORACLE-VT. Figure 6 shows ORACLE-VT is superior to both iPART and LRU. In particular, ORACLE-VT's miss rate is 3.0% and 2.8% better than iPART in the 2-thread partitioning and 4-thread workloads, respectively. These results show that for workloads requiring explicit allocation, ORACLE-VT does a better job allocating cache to threads than partitioning. In fact, ORACLE-VT's advantage over iPART is almost as large as iPART's advantage over LRU, which is very significant given that iPART is itself an ideal technique. Not only is ORACLE-VT superior for partitioning workloads, it also holds an advantage for LRU workloads. Figure 6 shows ORACLE-VT's miss rate is 2.9% better than LRU in the 2-thread LRU workloads.

While Figure 6 shows there exists significant room to improve cache allocation, it's unclear how to do this in

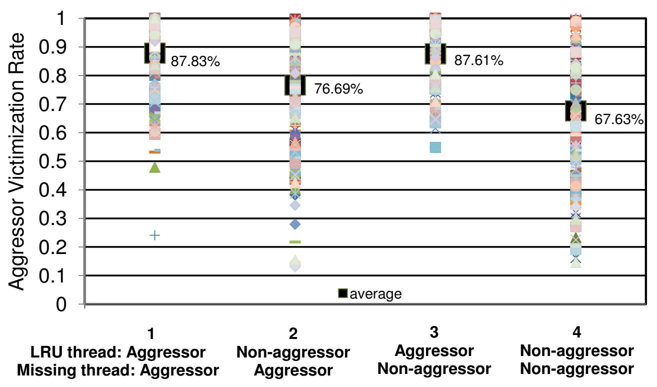

Figure 7. Aggressor thread victimization rates for the partitioning workloads when an aggressor thread (groups 1 and 3) or non-aggressor thread (groups 2 and 4) owns the LRU block.

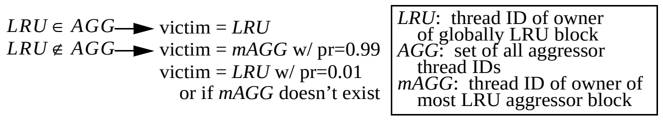
LRU ∈ AGG → victim = LRU
LRU ∉ AGG → victim = mAGG w/ pr=0.99
victim = LRU w/ pr=0.01
or if mAGG doesn't exist
LRU: thread ID of owner of globally LRU block
AGG: set of all aggressor thread IDs
mAGG: thread ID of owner of most LRU aggressor block

Figure 8. AGGRESSOR-VT policy.

practice since ORACLE-VT uses off-line information to select victim threads which is unavailable in real systems. Although ORACLE-VT is unimplementable, we analyzed its omniscient decisions for any recognizable patterns that can be practically exploited. In particular, we focus on its decisions under high cache interference since by far the majority of our workloads are partitioning workloads. From our analysis, we observe the following key insight: *ORACLE-VT often victimizes aggressor threads, especially when they own the most LRU cache block in the set.* (Henceforth, we will refer to this block as the "globally LRU block").

Figure 7 illustrates this insight by studying ORACLE-VT's aggressor thread victimization rate for the partitioning workloads. To acquire the data for Figure 7, we first identified aggressor threads in our trace-driven simulations by comparing each workload's per-thread cache allocation under LRU and ORACLE-VT. Threads that allocate more cache under LRU than ORACLE-VT are by definition aggressor threads (*i.e.*, given the opportunity, they acquire more cache than the optimal allocation provides). Then, we examined the ORACLE-VT traces again, and counted the cache misses that victimize aggressor threads. We break down the aggressor thread victimization rate into 4 groups. Groups 1 and 3 report the rate when the aggressor thread owns the globally LRU block, while groups 2 and 4 report the rate when a non-aggressor thread owns the globally LRU block. We also differentiate whether the cache-missing thread itself is an aggressor thread (groups 1 and 2) or a non-aggressor thread (groups 3 and 4). Each group plots the corresponding rate values for all 2-thread partitioning and 4-thread workloads (small symbols). A single darkened square (large symbol) indicates the average rate within each group.

As Figure 7 shows, ORACLE-VT victimizes an aggressor thread with very strong bias when the aggressor thread owns the globally LRU block: groups 1 and 3 show an 87.7% victimization rate on average, with practically no workloads exhibiting less than a 60% rate. This makes sense. Because aggressor threads are memory-intensive, they tend to have longer memory reference runs compared to non-aggressor threads. Consequently, intra-thread locality degradation due to reference interleaving (see Section II-A) is not as severe for aggressor threads. So, when an aggressor thread's cache block is globally LRU, this often reliably indicates intra-thread locality–*i.e.*, that the block will not be referenced again in the near future. Hence, the cache block should be evicted.

Figure 7 also shows ORACLE-VT victimizes an aggressor thread most of the time when a non-aggressor thread owns the globally LRU block: groups 2 and 4 show a 72.2% victimization rate on average. Non-aggressor threads tend to have shorter memory reference runs, so intra-thread locality degradation from interleaving references is more severe. Even if a non-aggressor thread's cache block is globally LRU, it may not be a good indicator of intra-thread locality–*i.e.*, the block may be referenced again in the near future. ORACLE-VT shows it's a good idea to still victimize an aggressor thread most of the time.

C. *Approximating ORACLE-VT*

To validate our insights, we develop a cache allocation policy that uses the aggressor thread information from Section III-B (rather than future reuse distance information) to select victim threads. This policy, called AGGRESSOR-VT, appears in Figure 8. On a cache miss, we check the globally LRU block. If it belongs to an aggressor thread, we always victimize that thread, and evict its LRU block. However, if it belongs to a non-aggressor thread, we find the aggressor thread that owns the most LRU block, and victimize it *probabilistically* with probability $pr = 0.99$. The other 1% of the time, we victimize the non-aggressor thread that owns the globally LRU block (otherwise, blocks belonging to non-aggressor threads may never leave the cache). If an aggressor thread's block does not exist in the cache set, we also victimize the non-aggressor thread that owns the globally LRU block.

In Figure 6, the bars labeled "A-VT" report the miss rates achieved by AGGRESSOR-VT in our trace-driven simulations for the partitioning workloads. (We exclude the 2-thread LRU workloads since they do not exhibit the cache interference that AGGRESSOR-VT was designed to mitigate). As Figure 6 shows, AGGRESSOR-VT approaches ORACLE-VT, coming within 1.7% and 1.0% of ORACLE-VT's miss rate for the 2-thread partitioning and 4-thread workloads, respectively. Moreover, AGGRESSOR-

VT achieves more than three-quarters of ORACLE-VT's benefit over LRU, and roughly half of ORACLE-VT's benefit over iPART. This shows aggressor thread information can be used to approximate ORACLE-VT's omniscient decisions, in essence serving as a proxy for future reuse distance information.

Although AGGRESSOR-VT faithfully approximates ORACLE-VT's decisions, why does it perform well from the standpoint of cache sharing patterns? Recall Section II-B. As Table I shows, the granularity of interleaving varies considerably across cache sets. *AGGRESSOR-VT permits a different allocation boundary per cache set that is tailored to the interference experienced in that set.* In cache sets with no interleaving, all cache blocks usually belong to the same thread. With only one possible victim thread, AGGRESSOR-VT reverts to LRU, allowing the thread to utilize the entire cache set and exploit as much intra-thread locality as possible. In cache sets with fine-grain interleaving, typically an aggressor thread incurs many more cache misses than non-aggressor threads, attempting to push them out of the set. AGGRESSOR-VT imposes a strong bias against the aggressor thread's blocks, evicting them with high frequency to keep the aggressor in check. These cache blocks often exhibit poor locality and are good eviction candidates, as Figure 7 shows us. Notice, AGGRESSOR-VT controls cache allocation without imposing a fixed allocation boundary. This can benefit sets with coarse-grain interleaving since threads can freely time-multiplex the available cache blocks. In particular, non-aggressor threads can easily allocate blocks from aggressor threads in such sets to exploit intra-thread locality.

In contrast, cache partitioning controls cache allocation using a fixed partition boundary across all sets. While this prevents aggressor threads from over-allocating in fine-grain interleaving sets, it sacrifices opportunities for flexible sharing in no interleaving and coarse-grain interleaving sets. Figure 6 shows AGGRESSOR-VT's flexibility provides benefits even over ideal partitioning. Lastly, AGGRESSOR-VT outperforms LRU since LRU doesn't provide any explicit allocation control, which is harmful in fine- and coarse-grain interleaving sets.

IV. Tuning Victimization On-Line

While Section III presents insights for effective cache allocation, it does not propose an implementable solution. This section builds upon AGGRESSOR-VT to develop a policy that can be used in real systems. Specifically, we address two important issues. First, we identify aggressor threads without relying on ORACLE-VT. We propose a sampling approach: take turns assuming each thread is an aggressor or non-aggressor, and select the setting with the highest observed performance. Essentially, "learn" the identity of the aggressor threads via performance feedback. Second, we

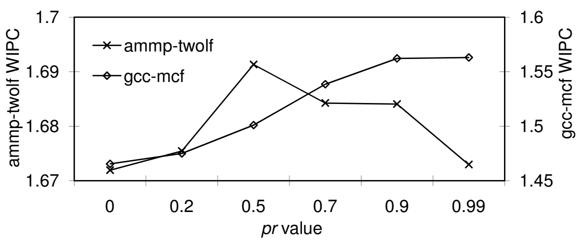

Figure 9. WIPC as a function of pr for the ammp-twolf and gcc-mcf workloads.

ensure AGGRESSOR-VT's cache miss rate gains in Section III translate into actual performance gains. To do so, we make AGGRESSOR-VT's cache allocation mechanism more flexible, and use the same sampling approach for identifying aggressor threads to tune the allocation for performance. In the following, we first discuss the second issue, and then go back to address the first issue.

A. Varying pr

AGGRESSOR-VT optimizes the global miss rate (*i.e.*, for all threads collectively) because it doesn't care which thread it victimizes so long as the thread exhibits the largest future reuse distance. While improving global miss rate generally improves performance, it doesn't always. Even if it does, it often doesn't improve fairness [5]. This is due to the complex way in which IPC can depend on miss rate within a thread, as well as the high variance in miss rates that can occur across threads. To be useful, a policy must be flexible, at times permitting cache allocations that sacrifice global miss rate to enable improvements in performance and/or fairness.

Quite the opposite, the AGGRESSOR-VT policy in Figure 8 is rigid: it employs a single allocation policy that almost always victimizes aggressor threads. We propose making AGGRESSOR-VT more flexible by tuning how strongly it is biased against aggressor threads. A natural place to do this is in the probability for deviating from the global LRU choice–*i.e.*, the "pr" value in Figure 8. Rather than fix $pr = 0.99$, we allow it to vary. By making pr smaller, we victimize aggressor threads less frequently, allowing them to allocate more cache. Conversely, by making pr larger, we victimize aggressor threads more frequently, guaranteeing more cache to the non-aggressors.

Besides varying the bias against aggressor threads, another dimension along which AGGRESSOR-VT can be made more flexible is to allow a *different* bias or pr value per thread. This permits tuning the cache allocation of aggressor threads or non-aggressor threads relative to each other, an important feature as thread count increases (*e.g.*, 4-thread workloads). We refer to the policy that permits flexible per-thread pr tuning as AGGRESSORpr-VT.

We implemented AGGRESSORpr-VT in our M5 simulator from Section III-A, permitting measurements on the policy's performance (see Section IV-D for details). Figure 9 reports the WIPC achieved by this policy as pr is varied from 0 to 0.99 for ammp-twolf and gcc-mcf, two example partitioning workloads. For simplicity, in each simulation, we use a single pr value across the entire workload. As Figure 9 shows, for the gcc-mcf workload, $pr = 0.99$ achieves the best WIPC. But for the ammp-twolf workload, $pr = 0.99$ does not achieve the best WIPC; instead, WIPC improves by reducing pr up to some point, after which performance degrades (*i.e.*, the curve is concave). Interestingly, when pr is set to 0, AGGRESSORpr-VT reverts to the basic LRU policy. This case performs poorly for both examples because they are partitioning workloads. From our experience, the two examples in Figure 9 are representative of many partitioning workloads: they either achieve their best performance around $pr = 0.99$ or $pr = 0.5$.

B. On-Line Sampling

To drive AGGRESSORpr-VT at runtime, we use on-line sampling. We propose identifying aggressor threads by sampling the weighted IPC when different threads are assumed to be aggressors, and selecting the setting with the highest observed WIPC. (In essence, we sample different permutations of thread IDs for the *AGG* set in Figure 8). We also propose using on-line sampling to learn the best pr values. In particular, we associate a separate pr value with each thread, as discussed in Section IV-A. When a thread cache misses, we use its pr value to select a victim thread. Along with different aggressor thread assignments, we sample the WIPC of different per-thread pr values as well, and select the best performing ones. Lastly, as mentioned in Section IV-A, AGGRESSORpr-VT reverts to LRU when $pr = 0$ for all threads. As long as we sample this setting, we can also learn whether a workload performs best under LRU, and thus should not employ any bias against aggressor threads.[4]

Unfortunately, on-line sampling incurs runtime overhead. When sampling poor configurations, the system can suffer reduced performance. To mitigate this problem, we exploit the lesson from Figure 9 that most partitioning workloads perform best around a small number of pr values. Specifically, we only try $pr = 0.5$ and 0.99 for each thread. In addition, we determine the parameters for each thread (aggressor/non-aggressor and pr) separately to avoid sampling the cross-product of all parameters. From our experience, this approach works well.

Figure 10 shows the pseudocode for our sampling technique. The algorithm proceeds in epochs. As shown by the "main" function in Figure 10, execution alternates between two operation modes. Normally, epochs are executed using

[4]Because WIPC [12] treats all threads equally, our sampling technique does not consider different thread priorities. However, it is possible to weight each thread's contribution to overall WIPC in proportion to its priority.

```
/* T = number of threads */
/* wipc$_i$ = multithread_IPC$_i$ / singlethread_IPC$_i$ */
/* WIPC = $\sum$ wipc$_i$ */
/* do_epoch(pr$_1$, ..., pr$_T$): execute 1 epoch using pr$_i$'s */
/* NewPhase(): true if wipc$_i$ order changes, else false */
main() {
  while (1) {
    Sample();
    do {
      do_epoch(pr$_1$, pr$_2$, ..., pr$_T$);
    } while (!NewPhase());
  }
}

Sample() {
  WIPC$_{LRU}$ = do_epoch(0, ..., 0);
  for (i = 0; i < T; i ++) {
    WIPC$_{i+}$ = do_epoch(0, ... +0.5 ..., 0);
    if (WIPC$_{i+}$ > WIPC$_{LRU}$) {
      pr$_i$ = +0.5;
      WIPC$_{i++}$ = do_epoch(0, ... +0.99 ..., 0);
      if (WIPC$_{i++}$ > WIPC$_{i+}$) pr$_i$ = +0.99;
      continue;
    }
    WIPC$_{i-}$ = do_epoch(0, ... -0.5 ..., 0);
    if (WIPC$_{i-}$ > WIPC$_{LRU}$) {
      pr$_i$ = -0.5;
      WIPC$_{i--}$ = do_epoch(0, ... -0.99 ..., 0);
      if (WIPC$_{i--}$ > WIPC$_{i-}$) pr$_i$ = -0.99;
    }
  }
}
```

Figure 10. Sampling algorithm for identifying aggressor threads and selecting pr values for AGGRESSORpr-VT.

the current pr values, pr_i for thread i (the "do_epoch" function). When a workload phase change occurs, the algorithm transitions to a sampling mode to determine new pr_i values (the "Sample" function). To detect phase changes, the algorithm monitors the weighted IPC of each thread, $wipc_i$, and assumes a phase change anytime the relative magnitude of the $wipc_i$'s change (the "NewPhase" function). By alternating between execution and sampling modes, the algorithm continuously adapts the pr values.

In the sampling mode, the algorithm first executes 1 epoch using LRU ($pr_i = 0, \forall i$). Then, the algorithm considers each thread in the workload one at a time to determine its pr_i value (either 0.5 or 0.99), first assuming the thread is an aggressor and then assuming it's a non-aggressor. In Figure 10, aggressor/non-aggressor status is designated by a "+" or "-" symbol, respectively, in front of the thread's pr_i value. After sampling, the thread's pr_i is set to the one yielding the highest WIPC, and the algorithm moves onto the next thread. In total, the algorithm performs at most $4 \times T + 1$ samples per phase change, where T is the number of threads. Each sample lasts 1 epoch.

C. Hardware Cost

AGGRESSORpr-VT requires supporting probabilistic victimization and the sampling algorithm described in Sec-

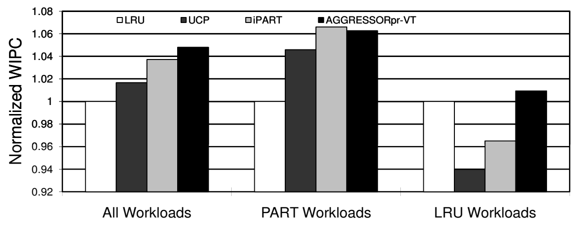

Figure 11. Normalized WIPC for LRU, UCP, iPART, and AGGRESSOR*pr*-VT for 2-thread workloads.

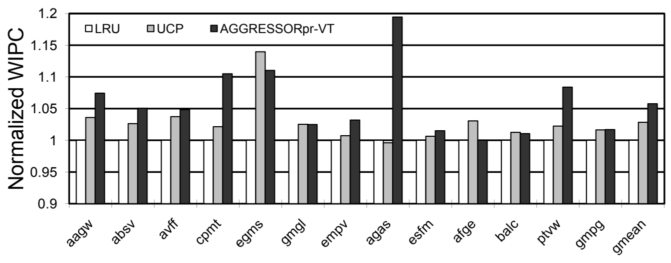

Figure 12. Normalized WIPC for LRU, UCP, and AGGRESSOR*pr*-VT for 4-thread workloads. Labels contain the first letter from each workload's four benchmarks, as listed in Table IV.

tions IV-A and IV-B. Probabilistic victimization requires maintaining two LRU lists per cache set: a global LRU list and an LRU list just for aggressor threads in case we need to probabilistically victimize the most-LRU cache block belonging to an aggressor thread. This hardware is comparable to what's needed for partitioning. (For 2 threads, partitioning would also need 2 LRU lists, one for each thread. For 4 threads, partitioning would need 4 LRU lists, whereas we would still only need 2.) In addition, probabilistic victimization also requires a random number generator, and a comparator to determine if the generated random number meets a pr threshold. The randomization logic is only accessed on cache misses, so it is off the hit path. We must also maintain a pr value per thread, and a single list of aggressor thread IDs to distinguish cache-missing threads as being either aggressors or non-aggressors.

The sampling algorithm can be implemented in a software handler, invoked once each epoch. We estimate the software handler would incur on average a few hundred cycles to perform the calculations in Figure 10 associated with a single epoch, which is insignificant compared to our epoch size (1M cycles). Hence, in the evaluation to follow, we do not model this overhead (though, of course, we model the performance degradation of sampling poor configurations which can be very significant).

D. AGGRESSOR*pr*-VT Evaluation

1) Methodology: We now evaluate AGGRESSOR*pr*-VT's performance. We use the same M5 simulator from Section III-A1 modified to implement the AGGRESSOR*pr*-VT policy and sampling algorithm. To drive our simulations, we use the same 2- and 4-thread workloads from Section III-A1. For all the performance experiments, we use our larger 500M-cycle simulation windows except the first few epochs are not timed to permit the sampling algorithm to determine the initial aggressor threads and pr values (although re-sampling at phase changes within the simulation window are timed).

As in Section III-B, we compare AGGRESSOR*pr*-VT against LRU and the iPART policy from Section III-A2. Since iPART is too expensive to simulate for 4-thread workloads, we implemented in our M5 simulator utility-based cache partitioning (UCP) [1], a recent partitioning technique. Like most partitioning techniques, UCP selects partitionings from stack distance profiles (SDPs) acquired on-line using special profiling hardware, called utility monitors (UMON). Two UMON profilers have been proposed [1]: UMON-global profiles SDPs exactly but incurs a very high hardware cost, while UMON-dss uses sampling to reduce the hardware cost but with some loss in profiling accuracy. Our M5 UCP implementation employs UMON-global. At the beginning of each epoch, UCP analyzes the SDPs profiled for each thread from the previous epoch, and computes the best partitioning. Our M5 UCP implementation analyzes SDPs for all possible partitionings at every epoch (we assume this analysis is performed at zero runtime cost). Although very aggressive, we consider our UCP implementation representative of state-of-the-art cache partitioners.

2) Performance Results: Figures 11 and 12 evaluate the performance of the AGGRESSOR*pr*-VT policy. In particular, Figure 11 compares the WIPC achieved by AGGRESSOR*pr*-VT against LRU, UCP, and iPART for our 2-thread workloads. The first group of bars (labeled "All Workloads") report the evaluation across all 216 (policy-sensitive) workloads. Then, the results are broken down for the 154 partitioning workloads (labeled "PART Workloads") and the 62 LRU workloads (labeled "LRU Workloads"). All bars are normalized against the WIPC achieved by LRU.

As the "PART Workloads" bars in Figure 11 show, AGGRESSOR*pr*-VT outperforms LRU by 6.3% for the partitioning workloads. This is not surprising: these workloads require explicit allocation to mitigate cache interference which LRU does not provide but AGGRESSOR*pr*-VT does. However, Figure 11 also shows AGGRESSOR*pr*-VT is slightly worse than iPART (by 0.4%) for the partitioning workloads. Although AGGRESSOR-VT is noticeably better than iPART in terms of miss rate (see Figure 6), this benefit is outweighed by the on-line sampling overhead incurred by AGGRESS*pr*-VT. Because iPART always finds the best partitioning in every epoch with zero overhead, it can handle workloads exhibiting phased behavior with unrealistic efficiency. When compared to an on-line partitioning technique, UCP, Figure 11 shows AGGRESSOR*pr*-VT holds a 1.6% gain. This comparison more accurately reflects the per-set flexibility benefits of AGGRESSORpr-VT discussed in Section III-C.

As the "LRU Workloads" bars in Figure 11 show, AGGRESSORpr-VT outperforms UCP and iPART by 7.3% and 4.0%, respectively, for the LRU workloads. Again, this is not surprising: these workloads require flexible sharing to permit time-multiplexing of cache capacity, which partitioning does not allow but AGGRESSORpr-VT does. Interestingly, Figure 11 also shows AGGRESSORpr-VT is slightly better than LRU (by 0.9%) for the LRU workloads. As discussed in Section II-B, even LRU workloads exhibit fine-grain interleaving in some cache sets. Occasionally, AGGRESSORpr-VT employs a non-zero pr value for certain threads to enforce explicit allocation in these sets (without sacrificing flexible sharing in sets with no interleaving or coarse-grain interleaving). This allows AGGRESSORpr-VT to achieve a slight performance boost over LRU. When combining results for partitioning and LRU workloads in the "All Workloads" bars, we see overall, AGGRESSORpr-VT outperforms LRU, UCP, and iPART by 4.86%, 3.15%, and 1.09%, respectively.

Due to the large number of 2-thread workloads, Figure 11 only presents summary results. Looking at individual workloads, we find AGGRESSORpr-VT can provide a much larger gain in many cases. For example, when considering 18 2-thread workloads for which AGGRESSORpr-VT provides its largest gains, AGGRESSORpr-VT outperforms UCP by 17.5% on average, and by as much as 28%.

Figure 12 evaluates AGGRESSORpr-VT on the 4-thread workloads. These results are presented in a format similar to Figure 11, except we omit iPART (which is infeasible for 4 threads), we present the data per workload, and the group of bars labeled "gmean" reports the average across all the workloads. In Figure 12, we see AGGRESSORpr-VT outperforms LRU in 12 workloads, and matches it in 1. Overall, AGGRESSORpr-VT achieves a 5.77% gain over LRU. This makes sense since all 4-thread workloads, except for one, are partitioning workloads. Figure 12 also shows AGGRESSORpr-VT outperforms UCP in 9 workloads while UCP outperforms AGGRESSORpr-VT in 4 workloads. Overall, AGGRESSORpr-VT achieves a 2.84% gain over UCP. Similar to the comparison in Figure 11, this advantage reflects AGGRESSORpr-VT's per-set flexibility benefit over cache partitioning.

V. Related Work

One advantage of AGGRESSORpr-VT is its ability to adapt to varying levels of interference across different cache sets. We are not the first to exploit per-set adaptation. SQVR [11] employs partitioning, but reverts to an LRU-like policy for cache sets that exhibit low interference. Also, cache-partitioning aware replacement [17] selects a per-set partition boundary by profiling the utility of giving each thread 1 more cache block in each set. Both techniques employ partitioning as their basic allocation control mechanism, and rely on per-set monitoring hardware (saturating counters [11] or shadow tags [17]) to modify the default partitioning decision per set. Instead of starting from partitioning, we propose a new allocation control mechanism, probabilistic victimization, that can adapt to per-set interference variation using a single parameter (pr) for each thread. Hence, we do not need per-set profiling hardware–we just profile the best pr value per thread. We are also the first to study ORACLE-VT, and to quantify the upper bound achievable by per-set allocation boundaries.

Rather than adapt policies to per-set interference variation, Adaptive Set Pinning (ASP) [18] re-directs references destined to high-interference sets into per-processor caches. Because ASP eliminates interference (rather than just managing it), it is not bounded by ORACLE-VT, and can potentially outperform AGGRESSORpr-VT. However, ASP requires additional per-processor caches to alleviate interference in the main cache. And like SQVR and cache-partitioning aware replacement, ASP also requires per-set profiling hardware to identify the problematic sets.

Cooperative cache partitioning (CCP) [2] is a hybrid technique that switches between a partitioning-like and LRU-like policy workload wide. But as our analysis shows, cache interference varies at the cache set level. By adapting at the workload level, CCP misses opportunities for optimization across cache sets.

Lastly, a large body of research has studied cache partitioning [4], [1], [6], [5], [9], [7], [8], [10]. AGGRESSORpr-VT is related to all of this prior research in that it provides another form of cache allocation control. To our knowledge, we are the first to propose a probabilistic victimization mechanism, and to demonstrate its benefits.

In addition to AGGRESSORpr-VT, our work also contributes insight on why cache interference varies. While previous research has pointed out that cache interference impacts the efficacy of partitioning and LRU [2], [3], it explains the cache interference variation in terms of locality. For example, Moreto et al [3] measure per-thread locality, and then use the locality measurements to predict cache interference. Our work identifies granularity of memory reference interleaving–in addition to per-thread locality–as a root cause of cache interference.

VI. Conclusion

This paper studies an ideal cache allocation technique, called ORACLE-VT, that selects victim threads using off-line information. ORACLE-VT always selects the best thread to victimize, so it maintains the best per-thread cache allocation at all times. We analyze ORACLE-VT, and find it victimizes aggressor threads about 80% of the time. To see if we can approximate ORACLE-VT, we develop AGGRESSOR-VT, a simple policy that probabilistically victimizes aggressor threads with strong bias. Our results show AGGRESSOR-VT can come close to ORACLE-VT's miss rate, achieving three-quarters of its advantage over LRU, and roughly half of its advantage over iPART. To make AGGRESSOR-VT feasible for real systems, we develop an on-line sampling technique to learn the identity of aggressor threads at runtime. We also modify AGGRESSOR-VT to include an intermediate probability for victimizing aggressor threads, and to permit separate threads to use different probabilities. The per-thread victimization probabilities that optimize WIPC are learned on-line using the sampling technique for identifying aggressor threads. This technique, which we call AGGRESSOR*pr*-VT, outperforms LRU, UCP [1], and iPART by 4.86%, 3.15%, and 1.09%, respectively.

Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful comments, and Aamer Jaleel, Bruce Jacob, Meng-Ju Wu, and Rajeev Barua for insightful discussion.

References

[1] M. K. Qureshi and Y. N. Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," in *Proceedings of the International Symposium on Microarchitecture*, Los Alamitos, CA, 2006, pp. 423–432.

[2] J. Chang and G. S. Sohi, "Cooperative Cache Partitioning for Chip Multiprocessors," in *Proceedings of the International Conference on Supercomputing*, Seattle, WA, June 2007, pp. 242–252.

[3] M. Moreto, F. J. Cazorla, A. Ramirez, and M. Valero, "Explaining Dynamic Cache Partitioning Speed Ups," *IEEE Computer Architecture Letters*, vol. 6, no. 1, pp. 1–4, 2007.

[4] L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni, "Communist, Utilitarian, and Capitalist Cache Policies on CMPs: Caches as a Shared Resource," in *Proceedings of the International Symposium on Parallel Architectures and Compilation Techniques*, Seattle, WA, 2006, pp. 13–22.

[5] S. Kim, D. Chandra, and Y. Solihin, "Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Washington, DC, 2004, pp. 111–122.

[6] H. S. Stone, J. Turek, and J. L. Wolf, "Optimal Partitioning of Cache Memory," *IEEE Transactions on Computers*, vol. 41, no. 9, pp. 1054–1068, 1992.

[7] E. Suh, S. Devadas, and L. Rudolph, "Analytical Cache Models with Applications to Cache Partitioning," in *Proceedings of the International Conference on Supercomputing*, Sorrento, Italy, 2001, pp. 1–12.

[8] E. Suh, S. Devadas, and L. Rudolph, "A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning," in *Proceedings of the International Symposium on High Performance Computer Architecture*, Washington, DC, 2002, p. 117.

[9] E. Suh, L. Rudolph, and S. Devadas, "Dynamic Cache Partitioning for Simultaneous Multithreading Systems," in *Proceedings of the IASTED International Conference on Parallel and Distributed Computing Systems*, 2001, pp. 116–127.

[10] E. Suh, L. Rudolph, and S. Devadas, "Dynamic Partitioning of Shared Cache Memory," *The Journal of Supercomputing*, vol. 28, no. 1, pp. 7–26, 2004.

[11] N. Rafique, W.-T. Lim, and M. Thottethodi, "Architectural Support for Operating System-Driven CMP Cache Management," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Seattle, WA, 2006, pp. 2–12.

[12] A. Snavely, D. M. Tullsen, and G. Voelker, "Symbiotic Jobscheduling with Priorities for a Simultaneous Multithreading Processor," in *Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, Marina Del Rey, CA, 2002, pp. 66–76.

[13] L. A. Belady, "A Study of Replacement Algorithms for a Virtual-Storage Computer," *IBM Systems Journal*, vol. 5, no. 2, pp. 78–101, 1966.

[14] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, "The M5 Simulator: Modeling Networked Systems," *IEEE Micro*, vol. 26, no. 4, pp. 52–60, 2006.

[15] D. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0," University of Wisconsin-Madison, CS TR 1342, 1997.

[16] T. Sherwood, E. Perelman, and B. Calder, "Basic block distribution analysis to find periodic behavior and simulation points in applications," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Barcelona, Spain, 2001, pp. 3–14.

[17] H. Dybdahl, P. Stenstrom, and L. Natvig, "A Cache-Partitioning Aware Replacement Policy for Chip Multiprocessors," in *Proceedings of the Conference on High Performance Computing*, Bangalore, India, 2006, pp. 22–34.

[18] S. Srikantaiah, M. Kandemir, and M. J. Irwin, "Adaptive Set Pinning: Managing Shared Caches in Chip Multiprocessors," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, Seattle, WA, 2008, pp. 135–144.