University of Maryland Technical Report UMIACS-TR-2008-13

Probabilistic Replacement: Enabling Flexible Use of Shared Caches for CMPs

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

Abstract

CMPs allow threads to share portions of the on-chip cache. Critical to successful sharing are the policies for allocating the shared cache to threads. Researchers have observed the best allocation policy often depends on the degree of cache interference. Typically, workloads with high cache interference require explicit working set isolation via cache partitioning, while workloads with low cache interference perform well without explicit allocation–i.e., using LRU.

While cache interference impacts cache allocation policies, relatively little is known about its root causes. This paper investigates how different sharing patterns in multiprogrammed workloads give rise to varying degrees of cache interference. We show cache interference is tied to the granularity of interleaving amongst inter-thread memory references: fine-grain interleaving yields high cache interference while coarse-grain interleaving yields low cache interference. Furthermore, interleaving granularity varies due to differences in mapping and timing of per-thread references in the cache. For example, coarse-grain interleaving occurs anytime per-thread references map to disjoint cache sets, or are performed during distinct time intervals. We quantify such spatial *and* temporal isolation *of per-thread memory references, and correlate its degree to LRU and partitioning performance.*

This paper also proposes probabilistic replacement *(PR), a new cache allocation policy motivated by our reference interleaving insights. PR controls the rate at which inter-thread replacements transfer cache resources between threads, and hence, the per-thread cache allocation boundary. Because PR operates on*

inter-thread replacements, it adapts to cache interference. When interleaving is coarse-grained (low interference), inter-thread replacements are rare, so PR reverts to LRU. As interleaving becomes more fine-grained (higher interference), inter-thread replacements increase, and PR in turn creates more resistance to impede cache allocation. Our results show PR outperforms LRU, UCP [1], and an ideal cache partitioning technique by 4.86%, 3.15%, and 1.09%, respectively.

1 Introduction

Chip multiprocessors (CMPs) allow simultaneous threads to share on-chip hardware structures. For example, it is common in today's CMPs to share portions of the on-chip memory hierarchy, especially the lowest level of on-chip cache. Critical to the success of sharing are the policies for allocating the shared cache to threads. The simplest policy is to allocate cache on demand as a consequence of the cache's default replacement policy–*e.g.*, LRU. Under LRU, cache allocation is implicit since the system cannot specify how much cache individual threads receive. Alternatively, researchers have also investigated *cache partitioning* [2, 3, 4, 1, 5, 6, 7, 8, 9] which explicitly allocates cache to threads by enforcing a hard partition boundary, thus forcing spatial isolation between threads' working sets. A common approach is to partition across separate cache ways, though cache-set partitioning is also possible.

Previous research has shown neither cache partitioning nor LRU is universally beneficial [2, 10]. This is because the two policies are tailored towards different levels of cache interference which varies from workload to workload. Under high cache interference, partitioning is best because it isolates the interfering working sets, guaranteeing cache resources to those threads that would otherwise be pushed out. On the other hand, under low cache interference, working sets can coexist symbiotically in the cache without partitioning. In this case, LRU is best because it allows threads to share the cache in a flexible manner.

While cache interference is a familiar concept, exactly *how* it arises and *why* it varies has not been fully explored. Given the important role it plays in cache allocation, a deeper understanding of cache interference's root causes can potentially help architects develop better policies. This paper conducts an in-depth study of the memory reference patterns in multiprogrammed workloads, and provides new insights into the nature of cache interference. We then develop a new cache allocation policy that uses these insights to more effectively allocate cache to threads compared to previous techniques.

We show cache interference is intimately tied to the *granularity of interleaving amongst inter-thread memory references*. If fine-grain interleaving occurs, then intra-thread locality can degrade significantly, yielding high cache interference. But if coarse-grain interleaving occurs, then threads can retain much of their inherent locality despite cache sharing, yielding low cache interference. Furthemore, the granularity of reference interleaving varies due to differences in the mapping and timing of per-thread memory references. Threads whose references map to *disjoint cache sets* or are performed during *distinct time intervals* exhibit coarse-grain interleaving; otherwise, fine-grain interleaving occurs.

To demonstrate these insights, we compare cache partitioning against LRU across an exhaustive set of 325 2-program workloads as well as a smaller set of 13 4-program workloads consisting of SPEC CPU2000 benchmarks. Among workloads where the allocation policy makes a difference, we find partitioning outperforms LRU 71% of the time while LRU outperforms partitioning 29% of the time, confirming that neither cache partitioning nor LRU is universally beneficial. Our results also show there exists a strong correlation between reference interleaving granularity and the amount of cache interference each workload experiences (and hence, whether it prefers partitioning or LRU). For example, in the 2-program workloads, when the combined average memory reference runlength for threads is less than (greater than) 7.4, the workload performs best under partitioning (LRU) 89% of the time. Finally, while interleaving granularity varies across workloads, our results also indicate such variation occurs considerably across cache sets from the *same workload* as well. In other words, the preference for partitioning or LRU is a per-cache set phenomenon; whether a workload prefers partitioning or LRU overall depends on which type of cache set dominates.

In addition to studying cache interference, we also propose *probabilistic replacement* (PR), a new cache allocation policy. Our approach integrates an explicit cache allocation mechanism into the basic LRU policy. Under LRU, the amount of cache each thread receives is determined by the rate at which threads replace each other's cache blocks (the inter-thread replacement rate). Rather than always replace the LRU block, on certain cache misses, PR replaces a block belonging to a different thread *probabilistically*. By tuning the probability for deviating from the LRU choice, the inter-thread replacement rate can be modulated, thus controlling cache allocation. In essence, while cache partitioning "sets up a wall" to contain cache allocation,

PR "puts up resistance" to impede the rate of cache allocation.

Because PR is a rate-based allocation mechanism, it naturally adapts to different granularities of memory reference interleaving, and hence, to different degrees of cache interference. When interleaving is coarse-grained, inter-thread replacements are rare, so PR becomes LRU-like. As interleaving becomes more fine-grained, inter-thread replacements increase–*i.e.*, greater cache interference–and in turn, PR is invoked more frequently. Consequently, the resistance PR puts up to impede cache allocation is in proportion to the degree of cache interference. This allows PR to adapt to varying cache interference levels, both across workloads as well as across cache sets within the same workload. Our results show PR outperforms LRU, a state-of-the-art cache partitioning technique, and an ideal cache partitioning technique by 4.86%, 3.15%, and 1.09%, respectively.

The remainder of this paper is organized as follows. Section 2 discusses the basic issues behind memory reference interleaving and its impact on cache allocation, and then motivates PR. Then, Section 3 studies memory reference interleaving in detail. Next, Section 4 presents the full PR policy, and evaluates its performance. Finally, Sections 5 and 6 discuss related work and conclusions, respectively.

2 Memory Interleaving's Impact on Cache Allocation

When programs share a cache, performance can degrade due to interference between per-program working sets. Without explicit management, the memory-intensive programs will likely receive the most cache, which often results in poor performance. The current solution to this problem is cache partitioning [2, 3, 4, 1, 5, 6, 7, 8, 9]. Cache partitioning enforces a hard partition boundary between threads in the cache, thus spatially isolating their working sets. The most popular approach is *way partitioning* [1, 9] which gives each thread a fixed allocation of cache blocks in every cache set–*i.e.*, allocation occurs by ways of the cache. The per-set allocations are enforced via the cache replacement logic which replaces a block belonging to a cache-missing thread whenever the thread has reached its allocation limit in the set.

Critical to the success of cache partitioning is the selection of the per-thread partition sizes. Most existing techniques select partitions using stack distance profiles (SDPs). An SDP counts the number of memory references a program exhibits at different stack distance values, and can compute a program's cache miss count as a function of stack depth, N, by simply summing the references in the SDP at stack positions beyond

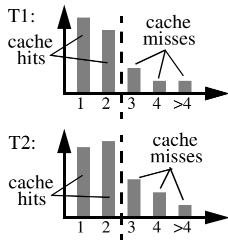


Figure 1. SDPs.

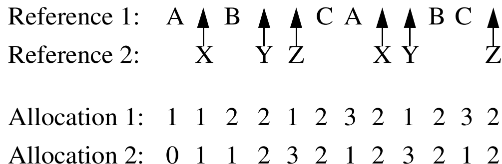


Figure 2. Fine-grain interleaving.

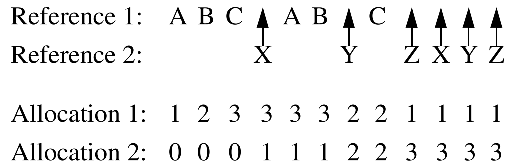


Figure 3. Coarse-grain interleaving.

N.[1] Hence, given the SDPs for all programs in a workload, the cache misses for different partitionings can be assessed, and the one that minimizes cache misses globally can be identified. To illustrate, Figure 1 shows the SDPs for two hypothetical threads. As shown by the dotted lines, the partitioning that minimizes cache misses for a 4-way set-associative cache gives each thread 2 cache ways.

Cache partitioning sacrifices references with stack distances beyond the partition boundary to ensure they do not interfere with references that have stack distances within the partition boundary, thus guaranteeing the latter will cache hit. For example, in Figure 1, thread T1's references at stack positions 3 and 4 are sacrificed (become cache misses) to guarantee resources for thread T2's references at stack positions 1 and 2, and vice versa. Notice, this pessimistically assumes the former will interfere with the latter. But in many cases, such interference may not occur. The next section explains why.

2.1 Interleaving Granularity and Cache Interference

Simultaneous threads interfere in a shared cache due to memory reference interleaving. Figure 2 illustrates this. In Figure 2, 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 cache hit, 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 2. Then, the cache capacity is divided amongst the two programs as a consequence of the LRU replacements. (The numbers in Figure 2 report the per-program cache allocation for each memory reference). Due to the memory interleaving, each reuse reference's stack distance increases to 6, so the LRU policy is unable to provide sufficient resources to each program at the

[1] This assumes a cache managed using an LRU replacement policy.

time of reuse. Hence, all reuse references cache miss.

Cache partitioning mitigates this degradation in intra-program locality by guaranteeing cache resources to threads. In Figure 2, cache partitioning would provide 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 cache miss, performance improves overall due to the additional cache hits. In this case, cache partitioning is necessary and improves performance because the threads exhibit actual cache intereference.

Unfortunately, cache partitioning can 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 3. 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 cache hit because the requisite cache capacity can be flexibly time-multiplexed between the two programs, as indicated by the cache allocation counts in Figure 3. 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 cache miss. These cache misses directly degrade performance since the other program receives no added benefit from its larger partition. In this case, cache partitioning is unnecessary–the cache interference it tries to mitigate doesn't occur.

This examples illustrates a key point: 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 isolation via partitioning. 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 in LRU.

2.2 Natural Spatial and Temporal Isolation

What determines the granularity of memory reference interleaving in a shared cache? For fine-grain interleaving to occur, the memory references in question must access the *same cache sets* during *overlapping time intervals*. Because most caches are set associative, references that map to disjoint sets cannot interfere with

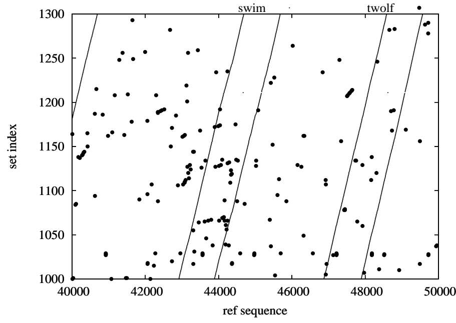


Figure 4. Memory reference pattern for the swim-twolf workload.

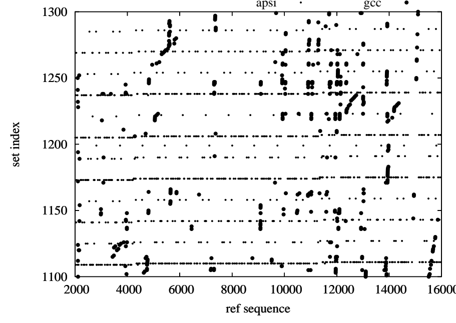


Figure 5. Memory reference pattern for the apsi-gcc workload.

$$\Delta = miss_1 * lru_2 - miss_2 * lru_1$$

$$0 = miss_1 * \frac{A_2}{C} - miss_2 * \frac{A_1}{C}$$

$$A_1 = \frac{miss_1 * C}{miss_1 + miss_2}$$

$$A_2 = \frac{miss_2 * C}{miss_1 + miss_2}$$

Figure 6. Flow equations for 2-program case.

each other. In this degenerate case, the memory references are *spatially isolated* and exhibit no interleaving. Even when memory references access the same cache sets, if the references occur during predominantly distinct time intervals, they are *temporally isolated* and exhibit coarse-grain interleaving. We say inter-thread memory references can experience varying degrees of natural isolation in both space and time. The presence of natural isolation yields coarse-grain interleaving, while the absence of natural isolation yields fine-grain interleaving.

To illustrate different spatial and temporal isolation, Figures 4 and 5 plot the memory references performed by two workloads, swim-twolf and apsi-gcc. For each plotted point, the Y axis indicates which cache set the reference maps to, while the X axis indicates the sequence ID (time) of the reference. Heavy versus light dots distinguish different programs' references. In Figure 4, swim (light) exhibits a scanning pattern, while twolf (heavy) accesses the cache uniformly. Swim's scans touch the entire cache, so they share many common sets with twolf (little spatial isolation). Moreover, the scans frequently cut in between twolf's references (little temporal isolation). Hence, swim-twolf exhibits fine-grain interleaving. In Figure 5, gcc (heavy) exhibits a uniform pattern, while apsi (light) references a few sets. Apsi's limited use of the sets means it does not interact with most of gcc's references (spatial isolation). Moreover, apsi's references occur in bursts, so there is very little chance for interleaving with gcc's references in the few sets it uses (temporal isolation). Hence, apsi-gcc exhibits coarse-grain interleaving.

Figures 4 and 5 illustrate workloads can exhibit a variety of different reference interleaving granularities. Given the discussion in Section 2.1, this suggests neither cache partitioning nor LRU can effectively support all workloads, implying there is the potential for new policies to achieve higher performance.

2.3 Rate-Based Cache Allocation

We propose *probabilistic replacement* (PR), a new cache allocation policy that effectively addresses both high and low cache interference. Essentially, PR integrates an explicit cache allocation mechanism into the LRU policy. Since PR is based on LRU, its approach to cache allocation is LRU-like. In LRU, cache allocation happens implicitly as a consequence of its replacement decisions. In particular, anytime a cache-missing thread replaces a block belonging to a *different* thread (*i.e.*, an inter-thread replacement), the cache-missing thread's allocation increases by 1 cache block. Over time, numerous cache blocks can flow back and forth between threads via inter-thread replacements (*e.g.*, Figures 2 and 3).

Within a given time interval, a thread's net cache gain, Δ, is simply the difference between the cache blocks it acquires and the cache blocks it loses. This can be expressed in terms of the number of cache misses thread i incurs in the time interval, $miss_i$ (*i.e.*, the events triggering resource transfers), and the probability that the LRU block belongs to thread i, lru_i (*i.e.*, the rate of inter-thread replacement per cache miss). Figure 6 shows Δ for a two-program workload. From this "flow equation," we can solve for thread i's cache allocation, A_i, in steady state ($\Delta = 0$) if we make the simplifying assumption that $lru_i = \frac{A_i}{C}$, where C is the total cache capacity. Figure 6 shows A_1 and A_2 in the two-program case.[2]

In Figure 6, we see LRU cache allocation is determined by the balance of inter-thread replacement rates between programs. Hence, cache allocation can be changed by tipping this balance. PR does this by altering replacement decisions. Rather than always replace the global LRU block, on certain cache misses, PR replaces the LRU block belonging to a different thread *probabilistically.* By tuning the probability for deviating from the LRU choice, the rate of inter-thread replacements can be modulated.

Because PR works by tuning the rate of inter-thread replacements, it can effectively support different levels of cache interference. During low cache interference, coarse-grain interleaving occurs, so there are very few inter-thread replacements. In this case, PR is activated infrequently, and reverts to LRU-like, thus enabling LRU's flexible sharing benefit. During high cache interference, fine-grain interleaving occurs, so there are a large number of inter-thread replacements. In this case, PR is activated frequently, and provides explicit isolation control. Later, Section 4 will give more insight into how PR works and performs.

[2] Note, we also assume $A_1 + A_2 = C$.

3 Memory Reference Interleaving Study

Having presented our insights qualitatively, we now perform quantitative studies. This section evaluates the impact of memory reference interleaving on the performance of different cache allocation policies. First, we describe the allocation policies considered in our study. Then, we discuss experimental methodology. Lastly, we present results.

3.1 Cache Partitioning Techniques

Besides the basic LRU policy, our study considers two cache partitioning techniques: a recent technique, called utility-based cache partitioning (UCP) [1], and an ideal cache partitioner, which we call iPART. UCP and iPART perform partitioning dynamically, as shown in Figure 7. CMP execution is divided into fixed time intervals, called *epochs*. At the beginning of each epoch, a partitioning of the cache is selected across threads. Both UCP and iPART perform way partitioning (see Section 2), so each thread is allocated some number of cache ways. During execution in the epoch, the cache replacement logic enforces the way partitioning. This process repeats for every epoch until the workload is completed.

UCP and iPART differ in how they select the partitionings. Like most partitioning techniques, UCP computes partitionings from SDPs (see Section 2) 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 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 UCP implementation analyzes SDPs for all possible partitionings at every epoch (the analysis is performed in hardware at zero runtime cost). Although very aggressive, we consider our UCP implementation representative of state-of-the-art cache partitioners.

iPART is an ideal technique that omnisciently selects the best partitioning every epoch. iPART is physically unrealizable, but can be studied for a limited number of cases via simulation (see Section 3.2). Our simulator checkpoints the entire CMP state at the beginning of each epoch from which it simulates every possible partitioning of the cache for 1 epoch. Before each exhaustive trial, the simulator rolls back to the

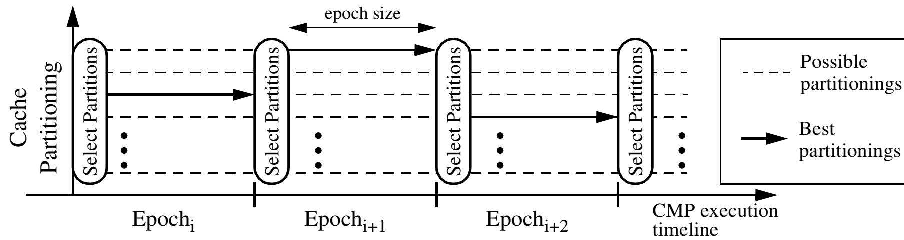


Figure 7. Epoch-based dynamic cache partitioning.

checkpoint so that every trial begins from the same architectural point. After trying all partitionings, the best-performing one is identified, and the simulator advances to the next epoch using this partitioning. At the end of the simulation, the cumulative execution time for the best partitionings is the execution time of the workload, while the overhead for all the exhaustive trials is ignored. iPART represents an upper bound on cache partitioning performance.

3.2 Experimental Methodology

We use M5 [11], a cycle-accurate event-driven simulator, to quantify the performance of different cache allocation policies. We configured M5 to model both a dual-core and 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 L1 caches split between instructions and data, 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. The latency to the L1s, L2, and main memory is 2, 10, and 200 cycles, respectively. Table 1 lists the detailed simulator parameters.

We apply cache partitioning to the shared L2 cache. To model UCP, we modified M5 to implement the UMON-global profiler for acquiring SDPs, as well as the analysis to determine the partitioning from the SDPs. To model iPART, we modified M5 to acquire simulator checkpoints every epoch, enabling the checkpoint-based simulation methodology discussed in Section 3.1. In each epoch, we try all possible allocations of the 8 ways in our L2 cache to different threads. While 2-program workloads only require trying 7 allocations (each program is allocated at least 1 cache way), simulations become prohibitive for larger workloads (*e.g.*, there are 455 different allocations per epoch for 4 threads). Due to the combinatorial num-

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

Table 1. Simulator parameters.

ber of allocations, we only simulate iPART for 2-program workloads. Our performance objective is average weighted IPC (WIPC) [12]. This is the metric we optimize when searching for the best partitioning, as well as the metric we report in our results. For both UCP and iPART, the epoch size is 1 million cycles, which is comparable to what's used in other dynamic cache partitioning techniques.

To drive our simulations, we employ multiprogrammed workloads created from the complete set of 26 SPEC CPU2000 benchmarks shown in Table 2. 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 3.[3] For the benchmarks, we use the pre-compiled Alpha binaries provided with the SimpleScalar tools [13] which have been built with the highest level of compiler optimization.[4] All our benchmarks use the reference input set. Before performing our detailed simulations, we fast forward the benchmarks in each workload to their representative simulation regions; the amount of fast forwarding is reported in the columns labeled "Skip" of Table 2. These were determined by SimPoint [14], and are posted on the SimPoint website.[5] After fast forwarding, detailed simulation is turned on, and the workload is simulated for 500 million cycles (for iPART, this does not include the exhaustive partitioning trials). On average, we simulate over 1 and 2 billion instructions for the 2- and 4-program workloads, respectively.

Finally, in addition to measuring performance, we also analyze memory reference interleaving. For some of these experiments, we use trace-driven simulation. We instrumented memory reference tracing in M5, and ran all of our workloads on the instrumented simulator assuming an LRU policy. Because our traces collectively consume significant disk storage, we acquire traces over a smaller simulation window of 100M

[3]In creating the 4-thread workloads, we ensured that each benchmark appears in 2 workloads.
[4]The binaries we used are available at http://www.simplescalar.com/benchmarks.html.
[5]Simulation regions we use are published at http://www-cse.ucsd.edu/ calder/simpoint/multiple-standard-simpoints.htm.

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

Table 2. SPEC CPU2000 benchmarks used to drive our study (B = Billion).

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

Table 3. 4-program workloads used in the evaluation.

cycles rather than the 500M cycles used for our performance simulations. After acquiring the traces, we replay them on a cache simulator that models LRU and iPART. For iPART, the exhaustive search technique looks for the partitioning that minimizes cache misses rather than WIPC.

3.3 Performance Results

We begin by comparing the performance of cache partitioning and LRU. In particular, we study whether workloads perform best using partitioning instead of LRU, or vice versa (we refer to these as "partitioning workloads" and "LRU workloads," respectively). Figure 8 studies our 2-thread workloads, comparing iPART against LRU. Out of the 325 2-thread workloads, 109 (37.5%) do not show any appreciable performance difference ($\leq$ 1%) between iPART and LRU, as shown by the "PART = LRU" bar in Figure 8. In these workloads, the allocation policy is irrelevant, for example, because the working sets of both benchmarks fit in cache. Of the remaining 216 workloads where performance is sensitive to the allocation policy, 154 (71%) are partitioning workloads while 62 (29%) are LRU workloads, as shown by the last two bars in Figure 8. Because there are a larger number of partitioning workloads, cache partitioning outperforms LRU overall. Figure 9 reports the WIPC of LRU and iPART averaged across the 216 policy-sensitive workloads. In Figure 9, we see iPART holds a 3.73% advantage over LRU on average. Nonetheless, Figure 8 shows there exists a mixture of partitioning and LRU workloads, a result consistent with previous work.

Figure 10 studies our 4-thread workloads by comparing UCP against LRU (since we cannot simulate iPART for 4 threads). The graph reports the WIPC of UCP normalized to the WIPC of LRU. Looking at the UCP bars in Figure 10, we see every workload but one (apsi-gcc-ammp-swim) is a partitioning workload.

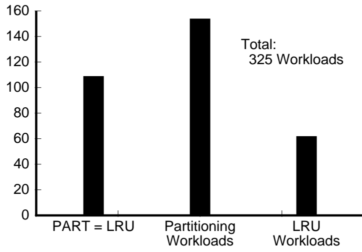


Figure 8. LRU and partitioning workload breakdown (2-thread workloads).

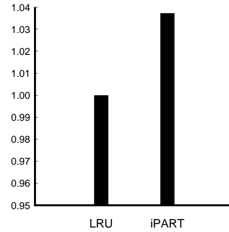


Figure 9. Normalized WIPC (2-thread workloads).

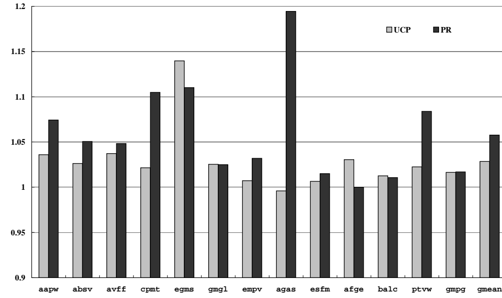


Figure 10. Normalized WIPC for UCP and PR (4-thread workloads).

This permits UCP to outperform LRU by 2.9% on average. (Note, iPART would achieve an even larger gain over LRU). These results suggest the frequency of partitioning workloads increases with thread count.

Having identified the partitioning and LRU workloads, we now study policy sensitivity across *different cache sets* within the same workload–*i.e.*, we differentiate "partitioning sets" from "LRU sets." To facilitate this study, we use trace-driven simulation, as described in Section 3.2. (Although event-driven simulation is more accurate, traces fix memory ordering, and thus permit us to correlate the performance study in this section to the memory interleaving study in the next section.) For each cache set and for each epoch in a workload's execution, we count the number of cache misses incurred by LRU and cache partitioning. If partitioning achieves a lower miss count, we label the set as a partitioning set. Otherwise, if LRU achieves a lower miss count, we label the set as an LRU set.

The first two rows in Table 4 report the percentage of cache misses that occur in partitioning sets versus LRU sets, broken down by workload type: the first column reports percentages averaged across the partitioning workloads, while the second column reports percentages averaged across the LRU workloads. Table 4 shows partitioning workloads incur 72.0% of their misses in partitioning sets, while LRU workloads incur 78.4% of their misses in LRU sets. In other words, the dominant preference at the cache set level determines whether the workload prefers partitioning or LRU overall. A more surprising result, however, is regardless of the workload policy preference, there exists a non-trivial number of cache sets that prefer the *other* policy: 28.0% and 21.6% of cache misses occur in LRU and partitioning sets for partitioning and LRU workloads, respectively. This demonstrates an important point: the preference for partitioning or LRU not only varies across workloads, it also varies across cache sets within a single workload.

	Partitioning Workloads	LRU Workloads
Misses in Partitioning Sets	72.0%	21.6%
Misses in LRU Sets	28.0%	78.4%

Table 4. Percentage of misses incurred in LRU and partitioning sets for partitioning and LRU workloads.

	Spatial	Coarse-Grain	Fine-Grain
2-Thread LRU Workloads	56.2%	22.2%	21.6%
2-Thread Partitioning Workloads	17.3%	10.7%	72.0%
4-Thread Partitioning Workloads	3.9%	5.6%	90.5%

Table 5. Percentage of memory references performed in spatially isolated, coarse-grain interleaved (temporally isolated), and fine-grain interleaved sets for 2-core/4-core LRU/partitioning workloads.

3.4 Memory Interleaving Results

We now investigate the connection between memory reference interleaving and partitioning / LRU performance. In particular, we measure the degree of spatial and temporal isolation of memory references to determine their interleaving granularity, and hence, the extent to which they interfere. Then, we correlate the observed interleaving granularity to the preference for partitioning or LRU. Our study is performed on the memory traces used in the previous section. While we consider all 4-thread workloads, for the 2-thread workloads, we only consider the 216 identified in Figure 8 as being policy sensitive.

To study spatial isolation, we examine the memory references performed to each set within each epoch. If 100% of the memory references from each "set-epoch" are performed by a single thread, we say the references are spatially isolated. Across all our workloads, we identified some 450,000 set-epochs in which spatial isolation occurs. In *all* of these set-epochs, we find LRU outperforms partitioning (*i.e.*, the set-epochs occur entirely in the LRU sets from Table 4). As discussed in Sections 2.1 and 2.2, spatially isolated memory references retain all of their intra-thread locality. Due to the lack of cache interference, performance is best under flexible cache sharing. Our results confirm LRU is always better than partitioning for the spatial isolation case.

How significant is spatial isolation? The second column of Table 5, labeled "Spatial," reports the percentage of memory references in different workloads that are spatially isolated. Table 5 shows spatial isolation occurs frequently. For the 2-thread workloads, it accounts for 56.2% and 17.3% of memory references in the LRU and partitioning workloads, respectively. For the 4-thread workloads, it accounts for fewer memory references, 3.9%, in the partitioning workloads.[6]

[6]Table 5 doesn't report results for 4-thread LRU workloads since there's only one of them

Figure 11. Average per-thread reference runlength sorted by partitioning's gain over LRU.

a). Aggressor Cache Miss:

LRU = tid → replace CB_{LRU}

LRU != tid → replace CB_{tid} with prob p_{tid} otherwise replace CB_{LRU}

b). Non-Aggressor Cache Miss:

LRU = tid → replace $CB_{!tid}$ with prob p_{tid} otherwise replace CB_{LRU}

LRU != tid → replace CB_{LRU}

LRU: thread ID of owner of LRU cache block

tid: thread ID of cache-missing thread

CB_i: most LRU cache block belonging to thread i

$CB_{!tid}$: next most LRU cache block not belonging to thread tid

p_{tid}: replacement probability for thread tid

Figure 12. The PR policy for explicitly controlling cache allocation.

Next, we study temporal isolation. For the memory references that are not spatially isolated, inter-thread reference interleaving occurs. We characterize the granularity of this interleaving by measuring the *average per-thread reference runlength*, *i.e.* the number of consecutive memory references a thread performs before another thread performs an interleaving reference. For each set, we exclude the set-epochs containing spatially isolated memory references, and compute the average runlength for each thread among the remaining (interleaving) references. We then average the runlengths across all the sets to obtain a single runlength value for the workload. All averages use the geometric mean.

Figure 11 plots the runlength value (Y-axis) for each workload (X-axis), with workloads sorted in ascending order of partitioning's benefit from left to right (*i.e.*, from largest gain of LRU over partitioning to largest gain of partitioning over LRU). The vertical line separates the LRU and partitioning workloads. As Figure 11 shows, there is a strong correlation between runlength and the preference for partitioning or LRU. At large runlengths (coarse-grain interleaving / temporal isolation), LRU usually outperforms partitioning, while at small runlengths (fine-grain interleaving / no temporal isolation), partitioning usually outperforms LRU. In particular, a runlength of 7.4 correctly classifies the partitioning and LRU workloads 89% of the time. As discussed in Sections 2.1 and 2.2, interleaving granularity determines cache interference, and hence, the efficacy of different allocation policies. Coarse-grain interleaving makes flexible cache sharing important while fine-grain interleaving makes explicit cache allocation important. Our results confirm LRU is better in the coarse-grain case; otherwise, partitioning is better.

How significant is temporal isolation? In Table 5, the "Coarse-Grain" column reports the percentage

of memory references participating in runs with length ≥ 7.4, while the "Fine-Grain" column reports the percentage of memory references participating in runs with length < 7.4. Table 5 shows temporal isolation (coarse-grain interleaving) occurs fairly frequently. For the 2-thread workloads, it accounts for 22.2% and 10.7% of memory references in the LRU and partitioning workloads, respectively. For the 4-thread partitioning workloads, it accounts for fewer references, 5.6%.

4 Probabilistic Replacement

This section presents probabilistic replacement (PR) in detail. Section 4.1 elaborates on the mechanism for providing allocation control from Section 2.3. Then, Section 4.2 describes an on-line technique for determining the parameters required by the PR mechanism. Finally, Section 4.3 evaluates PR.

4.1 Controlling Resource Transfer Rates

As discussed in Section 2.3, PR controls the inter-thread replacement rate between simultaneous threads. For workloads with frequent inter-thread replacements (due to fine-grain interleaving), LRU does not allocate cache resources to threads effectively, as discussed in Section 2.1. Typically, some threads are over-allocated, thus negatively impacting other threads. PR identifies the over-allocated threads, which we call "aggressors," and uses probabilistic replacement to reduce their allocation.

Figure 12 illustrates PR. On a cache miss by an aggressor thread `tid` (Figure 12a), if the LRU block belongs to `tid`, then it is replaced as normal. However, if the LRU block belongs to another thread, then with probability, p_{tid}, the most LRU block belonging to `tid` is replaced instead. This reduces the rate at which aggressor `tid` takes blocks away from other threads. On a cache miss by a non-aggressor thread `tid` (Figure 12b), if the LRU block does not belong to `tid`, then it is replaced as normal. However, if the LRU block belongs to `tid`, then with probability, p_{tid}, the most LRU block not belonging to `tid` is replaced instead. This increases the rate at which non-aggressor `tid` takes blocks away from other (hopefully aggressor) threads. Together, the replacement algorithms in Figures 12a and b reduce the cache allocation of aggressor threads while increasing the cache allocation of non-aggressor threads.

Figure 13 shows the cache allocation control provided by PR on the ammp-applu workload. The experiment was performed on the M5 simulator from Section 3.2 modified to implement PR. In Figure 13, we plot the fraction of cache allocated to ammp as a function of time (applu receives the remaining cache). The

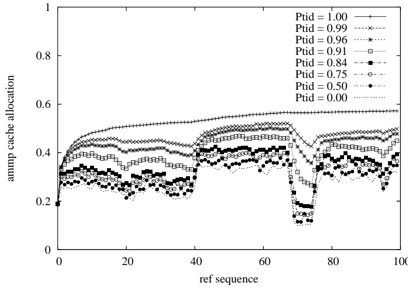


Figure 13. Cache allocation under PR for ammp-applu.

```
/* T = number of threads */
/* wipc_i = multithread_IPC_i / singlethread_IPC_i */
/* WIPC = Σ wipc_i */
/* do_epoch(p_1, ..., p_T): execute 1 epoch using p_i's */
/* NewPhase(): true if wipc_i order changes, else false */
main() {
  while (1) {
    Sample();
    do {
      do_epoch(p_1, p_2, ..., p_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) {
      p_i = +0.5;
      WIPC_i++ = do_epoch(0, ... +0.99 ..., 0);
      if (WIPC_i++ > WIPC_i+) p_i = +0.99;
      continue;}
    WIPC_i- = do_epoch(0, ... -0.5 ..., 0);
    if (WIPC_i- > WIPC_LRU) {
      p_i = -0.5;
      WIPC_i-- = do_epoch(0, ... -0.99 ..., 0);
      if (WIPC_i-- > WIPC_i-) p_i = -0.99;}
  }
}
```

Figure 14. Sample-based algorithm for identifying aggressor threads, and picking p_i values.

curves show cache allocation for different p_{tid} between 0 and 0.99. (Note, PR defaults to LRU for $p_{tid} = 0$). In the ammp-applu workload, applu is the aggressor. So, as p_r increases, applu's cache allocation is reduced. Notice, the allocation boundary is *not fixed*–it ebbs and flows as programs request resources over time, just like in LRU. In fact, if either ammp or applu were to stop requesting resources, the other could take over the entire cache. Yet, explicit cache allocation is achieved during simultaneous execution by controlling the rate at which the boundary can move.

4.2 On-Line Sampling Algorithm

PR requires identifying the workload's threads that are aggressors, and then choosing the per-thread replacement probabilities, p_i, to achieve the best performance. We use on-line sampling to select these PR parameters. Figure 14 shows the pseudocode for our sample-based technique. The algorithm alternates between two operation modes, as shown by the "main" function in Figure 14. Normally, it executes epochs using the current p_i values (the "do_epoch" function). When a workload phase change occurs, the algorithm transitions to a sampling mode to determine new p_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 parameters.

In the sampling mode, the algorithm first executes 1 epoch using LRU ($p_i = 0$, $\forall i$). Then, the algorithm considers each thread in the workload one at a time to determine its p_i value. While p_i can take on any value between 0 and 1, we only permit 0, 0.5, and 0.99 to mitigate sampling overhead. (Trying too many

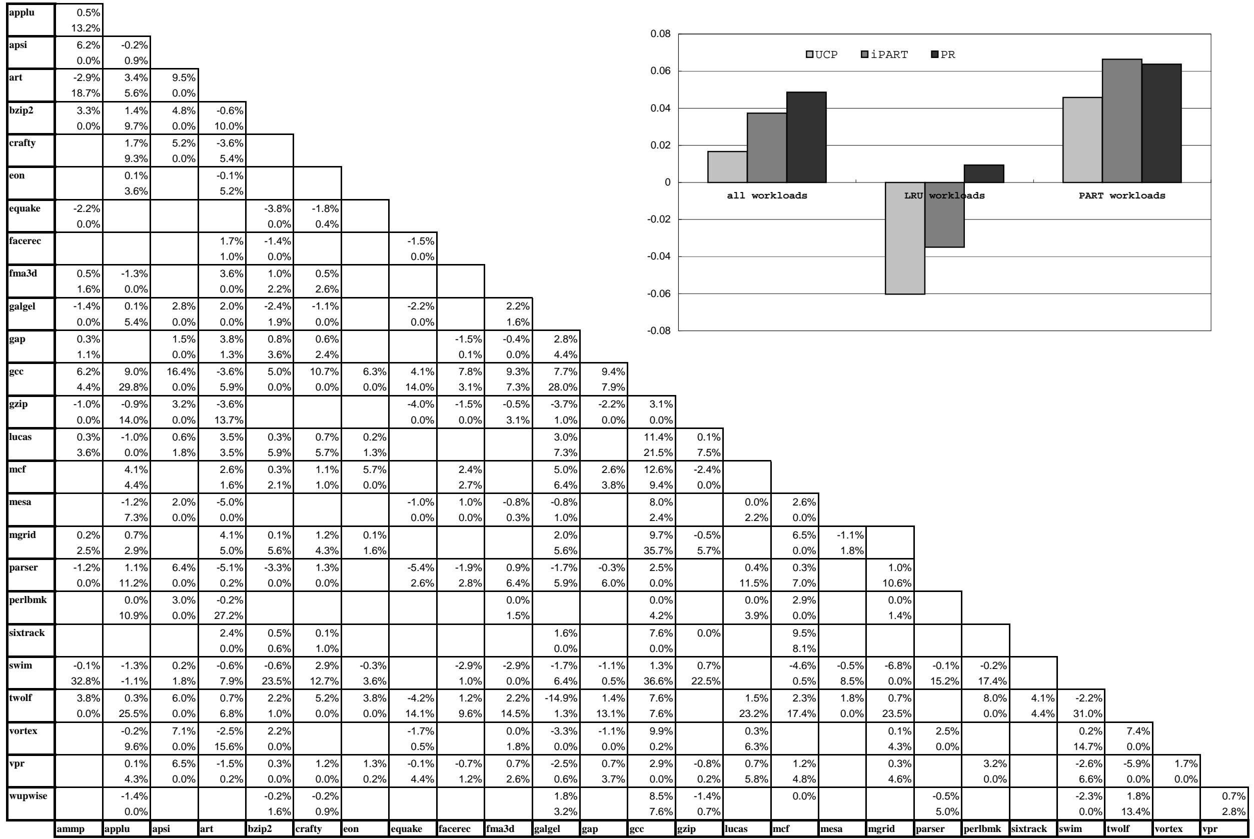
samples reduces performance since each sample can potentially perform poorly). Since $p_i = 0$ is tried during the LRU sample, the algorithm only needs to try $p_i = 0.5$ and $p_i = 0.99$, once assuming the thread is an aggressor, and a second time assuming the thread is a non-aggressor (indicated as +0.5/+0.99 and -0.5/-0.99, respectively, in Figure 14). After sampling, the thread's p_i is set accordingly, and the algorithm moves onto the next thread. In total, the algorithm performs at most $4 \times T + 1$ samples, where T is the number of threads. Each sample lasts 1 epoch.

Implementation Cost. PR requires supporting the probabilistic replacement mechanism and the sampling algorithm. Probabilistic replacement requires maintaining a global LRU list and a per-thread LRU list per cache set (similar hardware is also required for partitioning). In addition, it requires a random number generator, and a comparator to determine if the generated random number meets a p_i threshold. The randomization logic is only accessed on cache misses, so it is off the hit path. We must also store a single p_i threshold value per thread. The sampling algorithm can be implemented in a software handler, invoked once each epoch. Given epochs are 1M cycles, we believe the runtime cost would be negligible. We do not model this overhead in our PR evaluation.

4.3 PR Performance

We modified our M5 simulator to implement the PR policy and sampling algorithm presented in Sections 4.1 and 4.2. Then, we evaluated PR on our 2-thread and 4-thread workloads. These experiments use the same 500M-cycle simulation windows used in Section 3.3, except the first few epochs are not timed to permit our sampling algorithm to determine the initial aggressor threads and p_i values.

The matrix in Figure 15 presents our detailed PR results for the 2-thread workloads. Each matrix element reports PR's percentage gain over iPART (top value) and LRU (bottom value) under the WIPC metric for a single workload. (The workload's benchmarks are specified by the matrix element's row and column). Positive values indicate PR outperforms LRU/iPART, while negative values indicate LRU/iPART outperform PR. Empty matrix elements denote policy-insensitive workloads–*i.e.*, the 109 "PART = LRU" workloads in Figure 8. The matrix shows PR is often the better technique. Out of the 216 policy-sensitive workloads, PR outperforms LRU and iPART in 215 and 140 workloads, respectively.



	ammp	applu	apsi	art	bzip2	crafty	eon	equake	facerec	fma3d	galgel	gap
applu	0.5% / 13.2%											
apsi	6.2% / 0.0%	-0.2% / 0.9%										
art	-2.9% / 18.7%	3.4% / 5.6%	9.5% / 0.0%									
bzip2	3.3% / 0.0%	1.4% / 9.7%	4.8% / 0.0%	-0.6% / 10.0%								
crafty		1.7% / 9.3%	5.2% / 0.0%	-3.6% / 5.4%								
eon		0.1% / 3.6%		-0.1% / 5.2%								
equake	-2.2% / 0.0%				-3.8% / 0.0%	-1.8% / 0.4%						
facerec				1.7% / 1.0%	-1.4% / 0.0%			-1.5% / 0.0%				
fma3d	0.5% / 1.6%	-1.3% / 0.0%		3.6% / 0.0%	1.0% / 2.2%	0.5% / 2.6%						
galgel	-1.4% / 0.0%	0.1% / 5.4%	2.8% / 0.0%	2.0% / 0.0%	-2.4% / 1.9%	-1.1% / 0.0%		-2.2% / 0.0%		2.2% / 1.6%		
gap	0.3% / 1.1%		1.5% / 0.0%	3.8% / 1.3%	0.8% / 3.6%	0.6% / 2.4%			-1.5% / 0.1%	-0.4% / 0.0%	2.8% / 4.4%	
gcc	6.2% / 4.4%	9.0% / 29.8%	16.4% / 0.0%	-3.6% / 5.9%	5.0% / 0.0%	10.7% / 0.0%	6.3% / 0.0%	4.1% / 14.0%	7.8% / 3.1%	9.3% / 7.3%	7.7% / 28.0%	9.4% / 7.9%
gzip	-1.0% / 0.0%	-0.9% / 14.0%	3.2% / 0.0%	-3.6% / 13.7%				-4.0% / 0.0%	-1.5% / 0.0%	-0.5% / 3.1%	-3.7% / 1.0%	-2.2% / 0.0%
lucas	0.3% / 3.6%	-1.0% / 0.0%	0.6% / 1.8%	3.5% / 3.5%	0.3% / 5.9%	0.7% / 5.7%	0.2% / 1.3%				3.0% / 7.3%	
mcf		4.1% / 4.4%		2.6% / 1.6%	0.3% / 2.1%	1.1% / 1.0%	5.7% / 0.0%		2.4% / 2.7%		5.0% / 6.4%	2.6% / 3.8%
mesa		-1.2% / 7.3%	2.0% / 0.0%	-5.0% / 0.0%				-1.0% / 0.0%	1.0% / 0.0%	-0.8% / 0.3%	-0.8% / 1.0%	
mgrid	0.2% / 2.5%	0.7% / 2.9%		4.1% / 5.0%	0.1% / 5.6%	1.2% / 4.3%	0.1% / 1.6%				2.0% / 5.6%	
parser	-1.2% / 0.0%	1.1% / 11.2%	6.4% / 0.0%	-5.1% / 0.2%	-3.3% / 0.0%	1.3% / 0.0%		-5.4% / 2.6%	-1.9% / 2.8%	0.9% / 6.4%	-1.7% / 5.9%	-0.3% / 6.0%
perlbmk		0.0% / 10.9%	3.0% / 0.0%	-0.2% / 27.2%						0.0% / 1.5%		
sixtrack				2.4% / 0.0%	0.5% / 0.6%	0.1% / 1.0%					1.6% / 0.0%	
swim	-0.1% / 32.8%	-1.3% / -1.1%	0.2% / 1.8%	-0.6% / 7.9%	-0.6% / 23.5%	2.9% / 12.7%	-0.3% / 3.6%		-2.9% / 1.0%	-2.9% / 0.0%	-1.7% / 6.4%	-1.1% / 0.5%
twolf	3.8% / 0.0%	0.3% / 25.5%	6.0% / 0.0%	0.7% / 6.8%	2.2% / 1.0%	5.2% / 0.0%	3.8% / 0.0%	-4.2% / 14.1%	1.2% / 9.6%	2.2% / 14.5%	-14.9% / 1.3%	1.4% / 13.1%
vortex		-0.2% / 9.6%	7.1% / 0.0%	-2.5% / 15.6%	2.2% / 0.0%			-1.7% / 0.5%		0.0% / 1.8%	-3.3% / 0.0%	-1.1% / 0.0%
vpr		0.1% / 4.3%	6.5% / 0.0%	-1.5% / 0.2%	0.3% / 0.0%	1.2% / 0.0%	1.3% / 0.2%	-0.1% / 4.4%	-0.7% / 1.2%	0.7% / 2.6%	-2.5% / 0.6%	0.7% / 3.7%
wupwise		-1.4% / 0.0%			-0.2% / 1.6%	-0.2% / 0.9%					1.8% / 3.2%	

	gcc	gzip	lucas	mcf	mesa	mgrid	parser	perlbmk	sixtrack	swim	twolf	vortex	vpr
applu													
apsi													
art													
bzip2													
crafty													
eon													
equake													
facerec													
fma3d													
galgel													
gap													
gcc													
gzip	3.1% / 0.0%												
lucas	11.4% / 21.5%	0.1% / 7.5%											
mcf	12.6% / 9.4%	-2.4% / 0.0%											
mesa	8.0% / 2.4%		0.0% / 2.2%	2.6% / 0.0%									
mgrid	9.7% / 35.7%	-0.5% / 5.7%		6.5% / 0.0%	-1.1% / 1.8%								
parser	2.5% / 0.0%		0.4% / 11.5%	0.3% / 7.0%		1.0% / 10.6%							
perlbmk	0.0% / 4.2%		0.0% / 3.9%	2.9% / 0.0%		0.0% / 1.4%							
sixtrack	7.6% / 0.0%	0.0%		9.5% / 8.1%									
swim	1.3% / 36.6%	0.7% / 22.5%		-4.6% / 0.5%	-0.5% / 8.5%	-6.8% / 0.0%	-0.1% / 15.2%	-0.2% / 17.4%					
twolf	7.6% / 7.6%		1.5% / 23.2%	2.3% / 17.4%	1.8% / 0.0%	0.7% / 23.5%		8.0% / 0.0%	4.1% / 4.4%	-2.2% / 31.0%			
vortex	9.9% / 0.2%		0.3% / 6.3%			0.1% / 4.3%	2.5% / 0.0%			0.2% / 14.7%	7.4% / 0.0%		
vpr	2.9% / 0.0%	-0.8% / 0.2%	0.7% / 5.8%	1.2% / 4.8%		0.3% / 4.6%		3.2% / 0.0%		-2.6% / 6.6%	-5.9% / 0.0%	1.7% / 0.0%	
wupwise	8.5% / 7.6%	-1.4% / 0.7%		0.0%			-0.5% / 5.0%			-2.3% / 0.0%	1.8% / 13.4%		0.7% / 2.8%

Figure 15. Comparison of PR against LRU, UCP, and iPART on our 2-program workloads.

The bar graph in Figure 15 shows overall performance. In particular, the group of bars labeled "All Workloads" reports the WIPC of iPART, UCP, and PR averaged across the 215 policy-sensitive workloads.[7] The three bars are normalized to the WIPC achieved by LRU across the same 215 workloads. This bar graph shows PR outperforms LRU and UCP by 4.86% and 3.15%, respectively. PR also outperforms iPART by 1.09%. Although this is a modest margin, it is particularly significant given iPART represents the best that conventional partitioning can do.

Figure 10 presents our PR results for the 4-thread workloads. The bars labeled "PR" report the WIPC achieved by PR for each workload normalized to the WIPC achieved by LRU. Figure 10 shows PR outperforms LRU in 12 workloads, and matches it in 1. Overall, PR achieves an 5.77% gain over LRU. Comparing the "PR" and "UCP" bars in Figure 10, we see PR outperforms UCP in 9 workloads while UCP outperforms PR in 4 workloads. Overall, PR achieves a 2.84% gain over UCP.

[7]We ran UCP simulations across all 2-thread and 4-thread workloads even though Section 3.3 only reports the 4-thread results.

2-Thread Workloads			4-Thread Workloads		
	PR vs Part	PR vs LRU		PR vs Part	PR vs LRU
LRU Sets	15.66%	2.24%	LRU Sets	2.83%	1.88%
Partitioning Sets	-0.56%	1.93%	Partitioning Sets	1.05%	4.38%

Table 6. Improvement in cache miss count achieved by PR over partitioning and LRU in the partitioning and LRU sets for 2- and 4-thread workloads.

4.4 Analysis

To provide insight into PR's performance advantage, we simulated PR on the same memory traces from Section 3.4, and compare the miss count PR achieves against the misses incurred by LRU and partitioning. Table 6 reports these results. In Table 6, the columns labeled "PR vs Part" and "PR vs LRU" report PR's cache miss improvement over partitioning and LRU, respectively, for 2- and 4-thread workloads (note, for partitioning, the 2-thread workloads use iPART while the 4-thread workloads use UCP). The results are broken down by cache set type: the top row shows PR's improvement in the LRU sets while the bottom row shows PR's improvement in the partitioning sets. As discussed in Section 3.4, LRU and partitioning sets exhibit different granularities of memory reference interleaving, and hence, different degrees of cache interference. So, our results study PR's effectiveness as cache interference varies.

The top row of Table 6 shows PR is quite effective in the LRU sets. For the 2-thread workloads, PR improves miss count over partitioning and LRU by 15.66% and 2.24%, respectively, and for the 4-thread workloads, PR improves miss count over partitioning and LRU by 2.83% and 1.88%, respectively. LRU sets exhibit coarse-grain interleaving (low cache interference) due to spatial and temporal isolation of memory references. In this case, inter-thread replacements are infrequent, so PR acts like LRU and permits threads to flexibly share the cache. In contrast, cache partitioning enforces a fixed boundary in the cache, and sacrifices flexible sharing. It pays a particularly high price for this in the 2-thread case where isolation, especially spatial, is highly prevalent, as shown in Section 3.4. Not only does PR outperform partitioning in the LRU sets, it also slightly outperforms LRU as well. Because PR favors non-aggressor threads, it tends to install them into the cache faster than LRU. This can provide a performance boost anytime a non-aggressor thread performs a burst of references.

The bottom row of Table 6 shows PR is also effective in the partitioning sets. For the 2-thread workloads,

PR essentially matches the miss count compared to partitioning (a degradation of 0.56%) and improves the miss count over LRU by 1.93%. For the 4-thread workloads, PR improves miss count over partitioning and LRU by 1.05% and 4.38%, respectively. Partitioning sets exhibit fine-grain interleaving (high cache interference) due to lack of spatial and temporal isolation. In this case, inter-thread replacements are frequent, so PR limits the aggressor threads via probabilistic replacement, providing spatial isolation between threads' working sets, like partitioning does. In contrast, LRU over-allocates the aggressor threads, and performs poorly. Not only does PR outperform LRU in the partitioning sets, it also slightly outperforms partitioning for the 4-thread workloads. As illustrated in Figure 1, partitioning pessimistically sacrifices *all* memory references with stack distance beyond the partition boundary. Despite fine-grain interleaving, we find references with stack distance slightly beyond the partition boundary *can often coexist*. In many cases, PR permits these short-stack references to flexibly share the cache, thus outperforming partitioning.

PR's improvements in Table 6 for LRU and partitioning cache sets translate into improvements at the workload level. To illustrate this, we sub-set our performance results by workload type. The two groups of bars in Figure 15, labeled "LRU Workloads" and "PART Workloads," break down the performance of the 203 policy-sensitive workloads into LRU and partitioning workload groups, respectively. These bars use the same format as the "All Workloads" bars. Among LRU workloads, we see LRU outperforms UCP and iPART by 6% and 3.5%, respectively, due to the dominance of LRU sets in these workloads. Because PR effectively handles LRU sets, it achieves a 0.94% gain over LRU (as well as a 7.41% and 4.59% gain over UCP and iPART, respectively). Among partitioning workloads, we see UCP and iPART outperform LRU by 4.58% and 6.64%, respectively, due to the dominance of partitioning sets in these workloads. Because PR effectively handles partitioning sets, it achieves a 1.71% gain over UCP (as well as a 6.36% gain over LRU). Unfortunately, PR is slightly worse than iPART (by 0.35%) in the partitioning workloads.

5 Related Work

Previous research on CMP cache management has already noted partitioning and LRU are cache allocation techniques that address different types of cache interference [2, 10]. However, prior art explains the cache interference variation in terms of locality. For example, Moreto et al [10] measure per-thread locality, and then uses the locality measurements to predict cache interference. Our work shows per-thread locality is

only part of the picture. In addition, how threads' memory references map and interleave in the cache is also extremely important in determining the efficacy of different allocation policies.

Because we analyze interleaving, our work is close to Chandra et al [15] who demonstrate that inter-thread reference interleaving can degrade intra-thread locality. They develop a model based on interleaving probability to predict the miss rate of LRU given per-thread SDPs. Our work differs in that we use interleaving analysis to reason about the performance of different cache allocation policies. To our knowledge, we are the first to investigate interleaving's impact on cache allocation.

Previous research has also investigated adapting the allocation policy to different levels of cache interference in each cache set. Adaptive set pinning [16] profiles cache sets exhibiting high interference, and provides additional cache capacity to eliminate the conflicts in those hot sets. Cache-partitioning aware replacement [17] profiles the utility of giving each thread 1 more cache block per set by using per-set shadow tags, and drives partition selection using the profiles. Unfortunately, profiling the behavior in each cache set can be costly given the large number of sets. In contrast, PR is a rate-based allocation mechanism that controls cache allocation in proportion to the inter-thread replacement rate. Because inter-thread replacements directly reflect cache interference in each set, there is no need for per-set profiling. Hence, PR provides a simple way to achieve per-set adaptation.

Cooperative cache partitioning (CCP) [2] is a hybrid technique that switches between a partitioning-like and LRU-like policy. Unfortunately, as our analysis shows, cache interference varies at the cache set level. Hybrid techniques like CCP only adapt at the workload level, and thus miss opportunities for optimization across cache sets. In contrast, PR can adapt at the per-set level.

Finally, a large body of research has studied cache partitioning [3, 1, 5, 4, 8, 6, 7, 9]. PR borrows from all of this prior research in that it provides another form of cache allocation control. To our knowledge, however, we are the first to propose a rate-based control mechanism, and to demonstrate its benefits.

6 Conclusion

This paper demonstrates the degree of cache interference a multiprogrammed workload exhibits is intimately tied to the granularity of interleaving amongst inter-thread memory references. Coarse-grain interleaving occurs when there is spatial and/or temporal isolation of threads' memory references; otherwise,

fine-grain interleaving occurs. In addition to quantifying the degree of spatial and temporal isolation in several 2- and 4-thread workloads, we also demonstrate LRU and cache partitioning's efficacy is highly correlated to the interleaving granularity, with coarse-grain and fine-grain interleaving preferring LRU and partitioning, respectively. We show this correlation exists at the cache set level, as well as workload wide. Motivated by our interleaving insights, we also develop probabilistic replacement (PR), a new cache allocation policy. PR controls the rate at which inter-thread replacements transfer cache resources between threads. Because PR is a rate-based technique, it applies allocation control in proportion to the degree of cache interference. This permits adaptation to different sharing patterns across cache sets, as well as across workloads.

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*, 2006.

[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.

[3] 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), September 2006.

[4] S. Kim, D. Chandra, and Y. Solihin, "Fair cache sharing and partitioning in a chip multiprocessor architecture," in *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, (Washington, DC, USA), pp. 111–122, IEEE Computer Society, 2004.

[5] H. S. Stone, J. Turek, and J. L. Wolf, "Optimal Partitioning of Cache Memory," *IEEE Transactions on Computers*, vol. 41, September 1992.

[6] G. E. Suh, S. Devadas, and L. Rudolph, "Analytical Cache Models with Applications to Cache Partitioning," in *Proceedings of the 15th International Conference on Supercomputing*, (Sorrento, Italy), 2001.

[7] G. 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*, 2002.

[8] G. E. Suh, L. Rudolph, and S. Devadas, "Dynamic Cache Partitioning for Simultaneous Multithreading Systems," in *Proceedings of the 13th IASTED International Conference on Parallel and Distributed Computing Systems*, 2001.

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

[10] M. Moreto, F. J. Cazorla, A. Ramirez, and M. Valero, "Explaining Dynamic Cache Partitioning Speed Ups," *IEEE Computer Architecture Letters*, vol. 6, 2007.

[11] 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, pp. 52–60, July/August 2006.

[12] A. Snavely, D. M. Tullsen, and G. Voelker, "Symbiotic Jobscheduling with Priorities for a Simultaneous Multithreading Processor," in *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, June 2002.

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

[14] T. Sherwood, E. Perelman, and B. Calder, "Basic block distribution analysis to find periodic behavior and simulation points in applications," in *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[15] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture," in *Proceedings of the International Symposium on High Performance Computer Architecture*, February 2005.

[16] 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), pp. 135–144, March 2008.

[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*, 2006.