PIPP: Promotion/Insertion Pseudo-Partitioning of Multi-Core Shared Caches

Yuejian Xie, Gabriel H. Loh
College of Computing
Georgia Institute of Technology
Atlanta, GA, USA
{corvarx,loh}@cc.gatech.edu

ABSTRACT

Many multi-core processors employ a large last-level cache (LLC) shared among the multiple cores. Past research has demonstrated that sharing-oblivious cache management policies (e.g., LRU) can lead to poor performance and fairness when the multiple cores compete for the limited LLC capacity. Different memory access patterns can cause cache contention in different ways, and various techniques have been proposed to target some of these behaviors. In this work, we propose a new cache management approach that combines dynamic insertion and promotion policies to provide the benefits of cache partitioning, adaptive insertion, and capacity stealing all with a single mechanism. By handling multiple types of memory behaviors, our proposed technique outperforms techniques that target only either capacity partitioning or adaptive insertion.

Categories and Subject Descriptors

C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures

General Terms

Design, Performance

Keywords

Multi-core, cache, contention, sharing, insertion, promotion

1. INTRODUCTION

Modern multi-core processors employ large last-level caches (LLC) shared between all of the cores. An unmanaged shared multi-core cache leads to an inefficient and under-utilized system. As a result, many researchers have proposed a variety of techniques to manage the LLC to provide better performance and fairness [6, 7, 15, 19, 23, 28, 36, 34, 37, 38, 39, 40]. Most of these schemes follow a pattern of observation, policy selection, and enforcement. The observation part tracks the memory reference behaviors of individual cores in an attempt to deduce the per-core behaviors. The policy selection decides how each core should behave, and the enforcement mechanism makes it happen.

In this work, we propose a new cache management policy called Promotion/Insertion Pseudo-Partitioning (PIPP). Instead of explicitly partitioning the cache by ways, sets or total occupancy, PIPP implicitly partitions (or pseudo-partitions) the cache by simply managing the insertion and promotion policies of the cache. The insertion policy determines where in the eviction priority (e.g., LRU stack) a line should initially be installed [19, 35]. The *promotion policy* determines how the eviction priority should be changed on a cache hit [25]. We show that targeted insertion and promotion can provide effects similar to explicit cache partitioning. Our mechanism's ability to not always insert cache lines at the top of the recency stack also provides benefits similar to adaptive insertion schemes. Furthermore, by not strictly enforcing hard partitions in the cache, our approach allows cores to "steal" cache capacity from other cores, thereby making better use of the total available resources.

2. MOTIVATION

The academic and industrial research communities have already made many efforts to manage shared caches in multi-core processors. We now discuss several of the most related prior works so that our contributions can be properly put into context. Additional related work is discussed in Section 7.

2.1 Capacity Management

Different programs, or different threads from the same program, executing on a multi-core processor can have different memory capacity requirements (i.e., working set sizes). Given limited cache capacity, the question is how should these resources be divided among the competing cores? One approach is to provide fixed allocations for each core. The amount of space allocated or per-core priority levels can be determined at the software level, for example by the operating system [8, 12, 17, 18, 37, 40], or it could be dynamically determined based on run-time observations of program requirements [7, 23, 36, 38]. Most of the prior works employ some form of *way-partitioning* where each way or column of a w-way set-associative cache is assigned to one of the cores. This allocation could be structurally enforced by physically constraining all cache lines belonging to a core to reside in a fixed subset of columns, or the allocation can be logically enforced by ensuring that core$_i$ occupies no more than π_i lines per set (although those π_i lines may physically reside in any of the actual columns of the cache).

One common approach for adaptive cache partitioning is to use hardware to monitor the benefit or *utility* of allocating different numbers of ways to each core, and then to choose a partitioning to optimize some performance metric, such as minimizing the global number of cache misses. Figure 1 shows the number of additional hits that could be achieved for different cache allocations for two example programs. Based on this information, an allocation of three ways for core$_0$ and five ways for core$_1$ would result in the maximum number of misses avoided. While several past

ISCA'09, June 20–24, 2009, Austin, Texas, USA.
Copyright 2009 ACM 978-1-60558-526-0/09/06 ...$5.00.

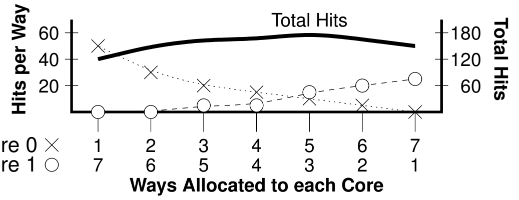

Figure 1: Example utility curves showing the number of additional hits provided for each additional way allocated to an LRU cache, and the total number of hits between the two cores for different partitionings of the cache.

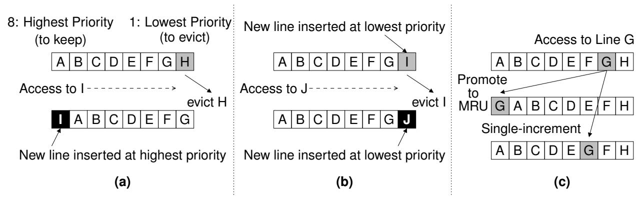

Figure 2: Example of (a) conventional insertion at the highest priority ("MRU") position and (b) insertion at the lowest priority ("LRU") position, and (c) different promotion policies.

techniques are similar in spirit, in this work we compare against the recently proposed utility-based cache partitioning (UCP) approach [36]. UCP uses a set of *shadow tags* to track what the contents of the cache would be if each core had sole ownership of the entire cache. A hit in the i^{th} most recent position in the LRU stack indicates that an i-way set-associative cache would have provided a hit, but a lower-associativity cache would not have. By counting the number of hits corresponding to each recency position, UCP can easily approximate associativity-benefit curves like those shown in Figure 1. Using this information, UCP then chooses an allocation to minimize global misses.

A potential limitation of strict way-partitioning of caches is that some cache capacity may be unutilized. If, for example, one cache set is not ever accessed by core$_i$, then the π_i cache lines allocated in that set go to waste. Another core$_j$ that does access that set with some regularity will still be trapped in its own partition of π_j lines, unable to ever make use of those wasted resources. Some previous approaches include some facilities for recouping these otherwise unused cache lines [12, 37].

2.2 Dead-Time Management

In a way-partitioned cache, the inefficient use of the unused cache lines discussed above are a direct artifact of the partitioning mechanism. Caches (whether for single- or multi-core processors) are not always efficiently used due to *dead* lines. Some cache lines are inserted into the cache, reused only a few times (perhaps never reused at all), and then eventually get evicted. There is an opportunity cost that from the time of a line's last access until it is evicted, the line consumes cache capacity without providing any benefit. In an LRU-managed cache, this "dead time" may last for hundreds or thousands of cycles as several other lines may need to be evicted before this dead line becomes the least recently used line.

In a way-partitioned cache, the opportunity loss can be magnified because dead lines may belong to other cores, but the partitioning mechanism prevents one core from stealing lines from other cores. Consider an example where two cores share a cache, and core$_0$ has one dead line. Under conventional way-partitioned cache management, when core$_1$ must make a replacement, it simply chooses a victim from among its own lines. In this example, however, if core$_1$ could choose core$_0$'s dead line, then core$_1$ could potentially improve its hit rate while having no adverse impact on core$_0$, since core$_0$ would have eventually evicted its dead line without having derived any more hits from it anyway.

One particularly important and common case of dead lines are those lines that are dead on arrival. The recently proposed *Dynamic Insertion Policy* (DIP) technique can insert lines directly into the least-recently-used position to minimize the residency time of such dead-on-arrival lines [35]. The thread-aware dynamic insertion policy (TADIP) uses dynamic monitoring of the policies combined with awareness of how these policy decisions interact in a multi-core context to reduce the amount of time dead lines take up the valuable shared cache resources [19].

3. INSERTION AND PROMOTION FOR CONTROLLING CACHE OCCUPANCY

In this section, we first review cache insertion policies, introduce the concept of promotion policies, and then detail how we combine these to manage a shared cache.

3.1 Cache Insertion and Promotion Policies

Traditional cache management has focused on cache *replacement* policies. When a new line must be installed, the replacement policy chooses a victim or evictee. Most conventional systems make use of a *Least-Recently Used* (LRU) replacement policy or an approximation thereof [11]. Figure 2(a) illustrates an example cache set with eight lines, logically organized left-to-right from highest priority (8: keep in the cache) to lowest priority (1: to be evicted).[1] For LRU replacement, the priority ordering is equal to the access recency (the least recently used line has the lowest priority for retention). An access to line I causes line H (in the lowest priority) to be chosen as the evictee. In a conventional LRU-managed cache, the newly installed line is inserted in the highest priority (MRU) position. It has been observed that there are cache lines that are accessed only once and then never accessed again [21, 35]. By installing these lines in the highest priority positions, LRU actually *maximizes* the amount of time that these lines occupy the cache. Qureshi et al. introduced the concept of separating a cache replacement policy into independent victim selection and insertion policies [35]. As shown in Figure 2(b), for no-reuse lines, insertion of this particular line into the lowest priority position is actually much better as this minimizes the amount of time that the line spends in the cache.

Independent of the victim selection and insertion policies, the conventional LRU-based policies all behave the same on a cache hit. That is, on any cache hit, the line is automatically moved or *promoted* to the highest priority position. We decompose cache management policies into three components: the victim selection and insertion decisions, as described earlier, and now the *promotion policy*. The promotion policy decides for a line that provides a cache hit how to update that line's position in the replacement priority order. Figure 2(c) illustrates both a traditional "promote-to-MRU" policy and a simple "single-increment" promotion policy.

[1] We use the terms lowest priority and highest priority instead of LRU and MRU, respectively, because once the insertion and promotion policies are modified, the ordering of the lines no longer strictly follows true "recency" and so calling a line "least recently used" when it is in fact *not* the least recently used line is inaccurate and can be confusing. Although the lines are shown left-to-right in priority order for illustration, the physical order in a cache may differ.

3.2 Basic PIPP

We now explain our algorithm for Promotion/Insertion Pseudo-Partitioning (PIPP) of shared caches. For n cores, we will assume that we are given a target partitioning Π=$\{\pi_1, \pi_2, ..., \pi_n\}$ such that $\sum_{i=1}^{n} \pi_i$=w, where w is the total set associativity of the cache. In this work, we make use of UCP's utility monitors to compute the target partitions, but they could potentially be specified by the operating system [37] or from many other approaches. The three policy decisions for insertion, promotion and eviction are described in turn. On insertion, core$_i$ simply installs all new incoming lines at priority position π_i. That is, a core's partitioning allocation determines its insertion position. On a cache hit, the promotion policy for PIPP promotes the line by a *single* priority position with a probability of p_{prom}, and the priority is *unchanged* with a probability of 1-p_{prom}. Finally, the victim selection always chooses the line in the lowest-priority position; this logic remains unchanged compared to conventional LRU.

PIPP does not strictly enforce the target partitioning, but the *combination* of targeted insertion and incremental promotion creates results *similar* to explicit partition enforcement, hence *pseudo*-partitioning. For systems with more than two cores, each core$_i$ still installs its lines into the priority position determined by π_i. While this insertion policy tends to cluster cache lines near the low-priority end of the ordering (for example, a quad-core system with a 16-way cache and Π=$\{6, 4, 4, 2\}$ results in no lines ever being inserted with a priority higher than 6), this approach can still produce the desired target partitioning. Core$_0$'s lines experience less promotion/demotion competition than the other cores, and so its lines tend to stay in the cache and are more likely to get promoted into higher priority positions. Core$_1$ and core$_2$ get inserted at the same position; while they will directly compete for cache resources, neither has a distinct advantage over the other, and both have a disadvantage compared to core$_0$ by being inserted at a lower priority position, so statistically both end up occupying fewer lines than core$_0$ but about the same lines as each other. Finally, core$_3$ must "swim upstream" against *all* of the traffic from the three other cores, and ends up occupying the fewest lines. Again, PIPP does not explicitly enforce the partitioning, but this example illustrates how partitioning-like behavior can be induced.

3.3 Example

Figure 3 illustrates a simple example for an eight-way cache shared between two cores with Π=$\{5, 3\}$. Core$_0$'s cache lines are represented by numerals in squares, and core$_1$ by letters in black circles. The figure also includes the insertion positions for each core as determined by the partitioning. Core$_1$ makes a request for line D, which misses and is inserted at position 3. Similarly, core$_0$'s request for lines 6 and 7 both miss and are in turn inserted at position 5. The next access is for core$_1$'s D, which hits in the cache. In a normal promote-to-MRU scheme, line D would be promoted to the highest priority position, but in our PIPP scheme, we promote the line by only a single position (for this example, we simply assume that p_{prom}=1). The example continues with several more misses (insertions) and hits (promotions).

The example contains a few interesting sections. Note that for most of the time, the actual cache occupancy for each of the two cores matches that of the target allocation Π=$\{5, 3\}$. There are intervals, however, where the instantaneous occupancy does deviate from the target partitioning, thus highlighting the fact that our scheme does not explicitly *enforce* partitions. To understand how the insertions and promotions can control capacity, consider a core with a marginal utility curve where most of the core's hits can be achieved with three ways, and that adding a fourth way does not provide many additional hits. If this core manages to grab a fourth way under PIPP, this cache line will not provide many additional hits (by assumption of the utility curve), the other more useful lines will continue to provide hits and get promoted above the extra line and therefore push the low-utility fourth line down the priority ordering where it will be quickly evicted, at which point the cache occupancy reconverges to that of the target allocation.

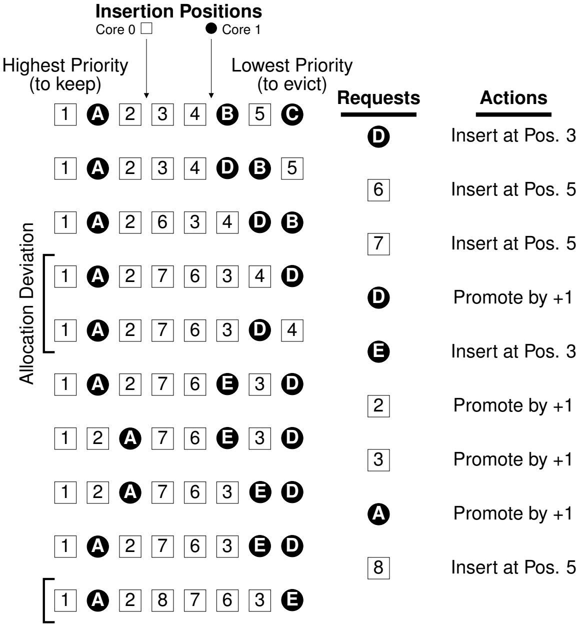

Figure 3: Example operation of PIPP for a variety of cache misses (insertions) and hits (promotions). Evictions always choose the lowest-priority cache line.

In addition to capacity management, the example in Figure 3 also illustrates some similarities and differences between PIPP and TADIP. Consider line E and assume that it does not exhibit any reuses. Inserting E in priority position 3 reduces the amount of time this dead line occupies cache space compared to conventional MRU insertion. Note that for line E, PIPP's approach is not as effective at eliminating dead time as TADIP, which would have inserted E directly into the lowest-priority position, thereby minimizing the amount of time the dead line spends in the cache. Consider line D which exhibits one reuse before becoming dead. On reuse, TADIP would promote the line directly to the highest priority position, but at this point the line is now dead and so this approach actually maximizes the residency time of the dead line. On the other hand, PIPP only promotes line D by a single position, thereby keeping D in a lower priority position which allows it to be evicted that much more quickly.

3.4 Stream-Sensitive PIPP

Poor cache performance due to inter-core interference is often caused by one or more programs that exhibit memory access patterns with very poor locality. Many of these situations have stream-like behaviors characterized by both a high access frequency as well as a large number of cache misses. These "low-utility" applications insert a large number of lines in the cache, often at a very high rate relative to the access frequencies of the other cores, and as a result they quickly flush out the cache lines used by the other cores. The worst part of it is that the useful lines evicted are replaced by useless lines that do not get reused.

Dual-Core Workloads		**Quad-Core Workloads**	
Workload Name	Application Suites, Names and Inputs	Workload Name	Application Suites, Names and Inputs
UCP2-0	f00:art, FP:ebgm.l	UCP4-0	i06:hmmer.r, f00:art, f00:equake, f06:soplex.r
UCP2-1	PB:continuous, f00:equake	UCP4-1	f00:art, Mi:dijkstra, FP:ebgm.l, f06:lbm
UCP2-2	i00:eon.k, md:mpeg4.d	UCP4-2	i00:crafty, i06:hmmer.r, f00:art, i06:omnetpp
UCP2-3	i06:h264ref.f, i06:gcc.g	UCP4-3	i06:hmmer.r, i00:bzip2.g, i06:astar.r, f06:bwaves
UCP2-4	i06:perl.s, i06:hmmer.n	UCP4-4	FP:ebgm.e, i06:h264ref.f, md:jpg2000.d, f00:art
UCP2-5	md:mpeg2.d, f06:sphinx3	UCP4-5	i06:hmmer.n, i06:bzip2.p, f06:soplex.p, f06:bwaves
UCP2-6	i06:perl.s, f06:sphinx3	UCP4-6	i06:h264ref.f, i06:gcc.g, md:pegwit.e, i06:hmmer.r
DIP2-0	i00:eon.k, FP:ebgm.e	DIP4-0	FP:ebgm.l, md:jpg2000.d, i00:eon.c, md:jpeg.d
DIP2-1	FP.ebgm.e, BP:clustalw.c	DIP4-1	BP:clustalw.c, Mi:adpcm.d, MN:bayes, FP:ebgm.e
DIP2-2	md:h264.e, FP:ebgm.l	DIP4-2	i06:h264ref.s, i00:gcc.s, f00:equake, i00:mcf
DIP2-3	FP:ebgm.l, md:jpg2000.d	DIP4-3	md:pegwit.d, FP:ebgm.e, i00:eon.k, md:pegwit.k
DIP2-4	i00:eon.k, i00:mcf	DIP4-4	i00:eon.k, i00:mcf, md:pegwit.d, md:adpcm.d
DIP2-5	i06:mcf, md:jpg2000.d	DIP4-5	i06:sjeng, i00:eon.r, FP:ebgm.e, i00:eon.k
DIP2-6	f00:art, md:jpg2000.d	DIP4-6	i06:mcf, md:h264.e, md:jpg2000.d, Mi:adpcm.d

The prefix before the colon identifies the benchmark suite; f00/f06/00/i06 are SPECcpu fp2000, fp2006, int2000 and int2006, respectively; BP is BioPerf [3]; FP is FacePerf [5]; md is MediaBench [27]; and Mi is MiBench [13]; MN is MineBench [32]; PB is PhysicsBench [41].

Table 1: Dual-core and quad-core workloads used in our evaluation. In the UCPx workloads, UCP performs better than TADIP, and visa-versa for the DIPx workloads.

Fortunately, such cache-unfriendly behaviors are often easy to detect. We propose a simple modification to PIPP that accounts for applications that exhibit stream-like behaviors. For each core$_i$, we make use of the Utility Monitor shadow tags to track (1) A_i the total number of accesses by core$_i$, and (2) m_i the number of misses the core would experience *if it had access to the entire cache for itself.* If the total number of misses exceeds a certain threshold $m_i \geq \theta_m$ or if the miss rate $\frac{m_i}{A_i} \geq \theta_{mr}$, then PIPP assumes the core is running a stream-like application. The intuition is that a large number of absolute misses will likely cause significant thrashing and interference with other cores, and a high relative miss rate means that even if more lines could be obtained, there would be very little return on the investment.

When PIPP detects that a core is running a stream-like application, it modifies its behavior as follows. First, all insertions for the streaming core$_s$ are made at priority position π_{stream}, independent of the target partition π_s. We set this insertion position to equal the current number of stream-like applications, effectively "allocating" a single way to each such program. The idea is that since the streaming accesses are very unlikely to exhibit reuse, there is no point in inserting them with priority greater than π_{stream}. Next, promotion for hits due to core$_s$ only occur with a reduced probability of $p_{stream} \ll p_{prom}$. The greatly reduced promotion probability ensures that *only* those cache lines that can demonstrate significant reuse will have a reasonable probability of getting promoted to higher priorities where they have a better chance of staying in the cache. This has similar properties to statistical filtering of caches [4]. For the corner case where *all* programs simultaneously exhibit streaming behavior, PIPP reverts to inserting all lines at the highest-priority position since there are no non-streaming applications to hurt.

4. EXPERIMENTAL EVALUATION

This section first briefly explains our simulation methodology, and then presents our main performance results.

4.1 Simulation Methodology

Our cycle-level model is built on top of the SimpleScalar toolset for x86 [2, 30]. We base our processor configuration on the Intel Core 2 processor clocked at 3.2GHz [9]. The pipeline has a minimum branch mispredict penalty of 14 cycles, the in-order portions (decode/commit) are four-wide, and the out-of-order core can issue up to six μops per cycle. We use a 96-entry ROB, 32-entry RS, 32-entry LDQ and a 20-entry STQ. Each core has 32KB, 8-way, 3-cycle level one instruction and data caches, and all cores share a 4MB, 16-way, 11-cycle LLC. All caches have 64-byte lines, and all are equipped with hardware prefetchers. We model a SDRAM memory with 9-9-9 timing on a 800MHz front-side bus (effective 1.6GHz with DDR2).

We use multi-programmed workloads created from a mix of applications including SPECcpu with the reference input sets as well as a wide variety of other suites. We used SimPoint 3.2 to choose representative samples [14]. The workloads are classified into two groups: those for which UCP performs better than TADIP, and those for which TADIP works better than UCP. Table 1 lists all of the dual- and quad-core workloads.

We also evaluated many additional workloads with less interesting behaviors (e.g., working sets mostly fit in the LLC, high DL1 hit rates resulting in low LLC activity) to verify that the techniques do not inadvertently cause any significant performance slowdowns; for brevity, these results are omitted since they do not show any unexpected results.

For each workload, we warm the caches and branch predictors for 500 million instructions, and then perform detailed simulation for 250 million instructions per benchmark. When an application reaches its instruction limit, it continues executing to compete for cache resources, but the statistics that we report only correspond to the original 250M instruction sample; this methodology is consistent with prior studies [19, 36]. Since one benchmark may take a lot longer to reach its instruction limit than others, the total number of simulated cycles is typically much larger than what might be expected for 250 million instructions per core. On average, each experiment simulated over one billion cycles (not including warm-up activities) with a maximum of about 9.5 billion cycles for one of the quad-core workloads with substantial memory accesses. For all experiments, we report weighted speedup (i.e., SMT speedup) $\sum_{i=1}^{c} IPC[i]/IPC_{sa}[i]$, where IPC_{sa} is the $s$tand-$a$lone IPC when the core has exclusive access to the entire processor [31], IPC throughput $\sum_{i=1}^{c} IPC[i]$, and the harmonic mean of weighted speedups $\frac{c}{\sum_{i=1}^{c}(IPC_{sa}[i]/IPC[i])}$ which accounts for both fairness and performance [7]. Unless otherwise specified, "performance" will refer to weighted speedup.

4.2 Results

For all performance comparisons, we use a conventional unmanaged, shared cache with LRU replacement as the baseline. We com-

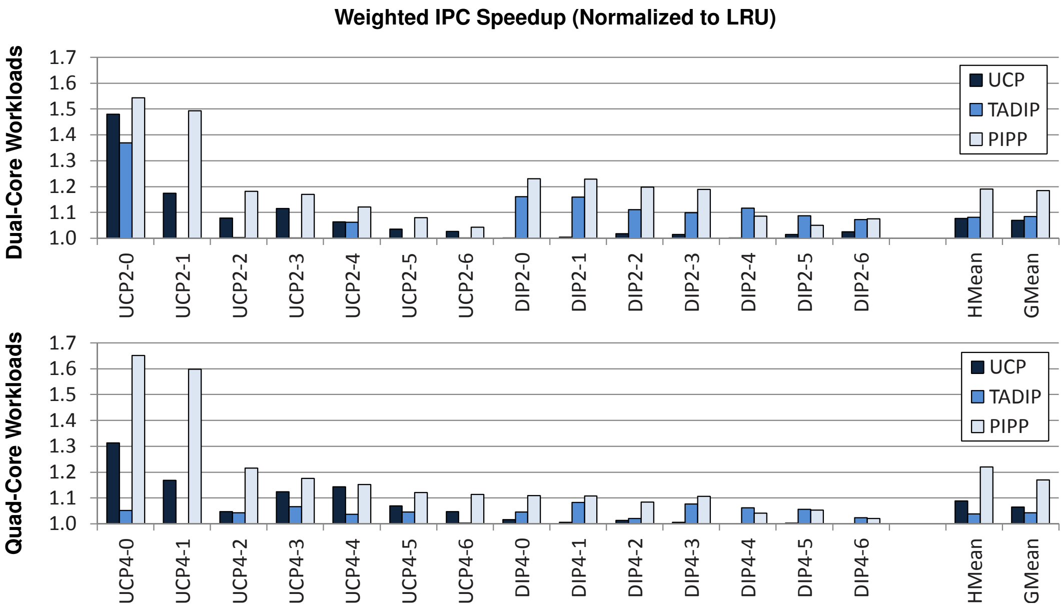

Figure 4: Performance as measured by the weighted speedups of IPC for UCP, TADIP and PIPP normalized to an LRU-managed cache for dual-core and quad-core workloads.

pare against UCP, TADIP (specifically TADIP-F) and PIPP. Both UCP and PIPP make use of shadow tags with dynamic set sampling to track per-core utility curves (32 sets per shadow tag) [33]; both UCP and PIPP use this information to feed to the same partitioning algorithm (optimal for dual-core, and Lookahead [36] for quad-core) to select the target partition [36]. All PIPP results here make use of the stream-sensitive version of PIPP with probabilities p_{prom}=$\frac{3}{4}$ and p_{stream}=$\frac{1}{128}$. The probability p_{prom} simply requires generating a two-bit pseudo-random number and testing that the result is not equal to zero, and p_{stream} requires generating a seven-bit pseudo-random number and testing that it *is* equal to zero. The stream-detection thresholds were likewise chosen for easy implementation. We use $\theta_m \geq 4095$, which simply requires testing that a 12-bit saturating counter has reached its maximum value. Similarly, θ_{mr} can be selected for easy computation; we used the value 12.5%: $\frac{m_i}{A_i} \geq \frac{1}{8}$ is equal to $m_i \geq \frac{A_i}{8}$ (right-shift A_i by three and compare with m_i).

Figure 4 shows the performance impact of the different cache management techniques for the weighted IPC speedup. For the dual-core simulations, PIPP consistently outperforms unmanaged LRU by a large margin (19.0% on the harmonic mean), and also outperforms both UCP and TADIP (10.6% and 10.1%, respectively). Similar results hold for the quad-core case where PIPP is 21.9% better than LRU, 12.1% better than UCP and 17.5% better than TADIP.

Figure 5 shows the results measured by total IPC throughput and fair speedup, relative to LRU. The trends are very similar to the weighted speedup results, demonstrating that PIPP also provides higher raw throughput and better fairness. Due to the similarity of the overall trends, we only deal with weighted speedup in the rest of this paper.

PIPP consistently outperforms UCP for both dual-core and quad-core workloads on all of the performance metrics (with the one exception of UCP4-4 for the fair speedup metric where the performance of PIPP is still very close to UCP). PIPP's strong performance comes from its effective capacity management combined with it not being strictly bound to the partition allocations and its abilities to exploit DIP-friendly behaviors. These attributes will be further explored in the next section.

There are a few workloads where, while still performing well compared to LRU, PIPP still gets beat by TADIP. An interesting pair of workloads to contrast are DIP2-3 and DIP2-5, where PIPP performs better than TADIP on DIP2-3 and TADIP is superior on DIP2-5, and each responds well to LRU insertion. For each workload, one of the benchmarks contained a large number of lines that observe a single reuse in an unshared cache; contention in a shared cache shortens the lifetimes of these lines such that they become zero-reuse lines that TADIP takes advantage of.

In the case of DIP2-3, there are many lines with just a few uses, but very few lines with many uses. It would be desirable to have a cache management policy that can keep the lines resident in the cache just long enough to expose these additional hits, but then quickly evict them after they become dead. PIPP can provide this type of effect because it inserts the lines with slightly higher priority than the lowest possible which provides a short window for additional hits to manifest. PIPP's incremental promotion policy discourages these lines from staying in the cache for too long after they become dead. TADIP on the other hand may not keep the lines in the cache long enough to expose the extra hits, and when it does, the lines are directly promoted to the highest priority position. Since most lines in this program have only a single reuse, TADIP's promotion policy actually ends up maximizing the dead time of these lines.

Compared to the DIP2-3 workload, DIP2-5 (where TADIP performs better) has many more lines with many more reuses (in addition to the many zero-reuse lines). In this scenario, PIPP's incremental promotion policy is too cautious leading to situations where lines with more uses in the near future do not get promoted high enough in the priority ordering for them to evade eviction before their next uses. TADIP's more aggressive promotion policy does a better job at keeping these lines in the cache to expose many more hits. This suggests that perhaps PIPP could be further enhanced by providing some facilities to dynamically tune the aggressiveness of the promotion policies.

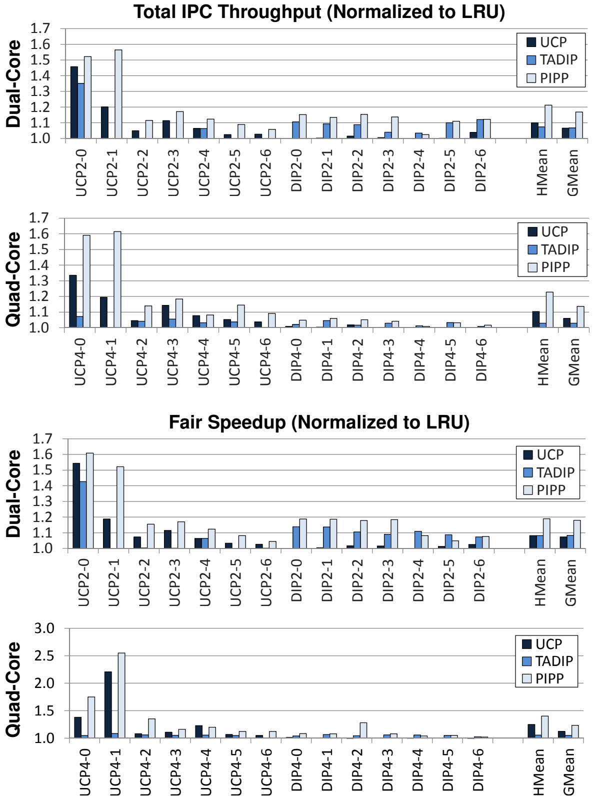

Figure 5: Performance results for the IPC throughput and fair speedup metrics.

5. ANALYSIS

In this section, we first provide some additional quantitative results to demonstrate PIPP's abilities to deal with different types of memory behaviors. We then provide results from our sensitivity analysis to show the importance of the different design choices for PIPP.

5.1 Why Does PIPP Work?

Occupancy Control

We have claimed that PIPP is effective at controlling the occupancy of shared caches, but so far we have only demonstrated that the performance of PIPP is as good as or better than UCP. To measure the effectiveness of PIPP's capacity management, we measured the difference between each core's actual cache occupancy and the target partitioning. We define a core's partitioning deviation as the absolute difference between the average number of ways occupied and the target number of ways allocated. For example, if core$_i$ has a target allocation of π_i=3 in an eight-way cache, but actually occupies half of the cache's total capacity, then on average core$_i$ occupies four out of every eight lines in the cache, and so the partitioning deviation is equal to one. During simulation, every one million cycles we measure the partitioning deviation of each core and then we average all of the samples over the entire simulation.

Figure 6 shows the average partitioning deviation for all of the workloads. For the vast majority of the workloads, the partitioning deviation is within 1.0 of the target allocation. This shows that PIPP can create aggregate conditions that are similar to those created by UCP. The imperfect partitioning is a direct result of the fact that PIPP only *pseudo*-partitions the cache but does not explicitly enforce allocations. Nevertheless, PIPP still manages to balance per-core capacities in a way that comes reasonably close to the target allocations. It is also important to note that PIPP's partitioning deviation does not necessarily result in lower performance, for example due to the theft of a dead line from another core.

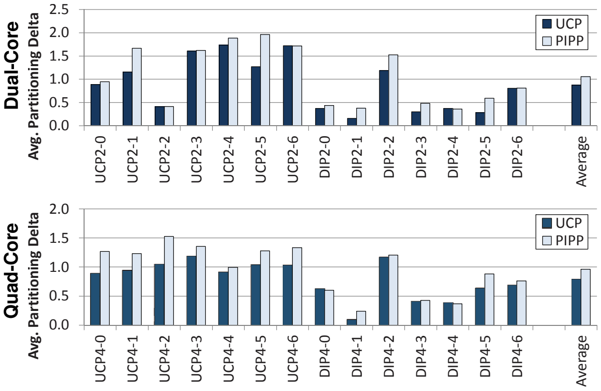

Figure 6: Average partitioning deviation for all of the dual-core and quad-core workloads for PIPP.

Insertion Behavior

PIPP can reduce the amount of time that dead-on-arrival lines reside in the cache by simply inserting them in a position of lower priority. Of particular interest are the workloads DIP2-0 and DIP2-1 where UCP provides absolutely no benefit and TADIP is still able to deliver about 16% speedup for both workloads. These "pure TADIP" workloads exhibit a large number of lines with no-reuse, and this is reflected by the fact that on average, TADIP inserts lines with a priority of 1.683 and 1.685 (where a priority of 1 is "LRU" insertion, and a priority of 16 is "MRU" insertion) for DIP2-0 and DIP2-1, respectively. For the same workloads, PIPP has average insertion priorities of 1.330 and 1.329, effectively showing that PIPP can mimic TADIP's LRU-insertion behavior.

Pseudo-Partitioning Benefit

One of the qualitative differences between PIPP and conventional way-partitioned cache management schemes is that the partitioning is not rigid which allows cores to obtain more cache resources than would be normally allowed. In particular, we say that a core "steals" a cache line when it inserts a line into the LLC, but inserting this line causes the core to exceed its target partition (in this cache set). For example, if a core has a target allocation of π=3 ways, inserts a line and then the core now occupies four or more ways in this set, then we say that this insertion stole a cache line.

For each workload, we monitor every LLC insertion and record whether the insertion resulted in a line theft. We then monitor two types of events. The first is for every line which was stolen, how many times was that line subsequently reused? This provides an estimate of the benefit of stealing capacity from other cores. Second, for every stolen line, we remember the previous occupant's address. If we later observe a miss on this address, but we find the address in one of the stolen line's previous-address fields, then we record this as a miss that was caused by stealing the line ("forced miss"). Figure 7 shows the averages (across accesses by all cores) for these metrics along with markers indicating the net impact accounting for both additional hits and misses caused by line thefts. Note that these metrics are not necessarily directly proportional to performance impact because, for example, the number of hits credited for a stolen line include all reuses of that line. In a conventional cache, it is possible that after the first access, the line is reinstalled

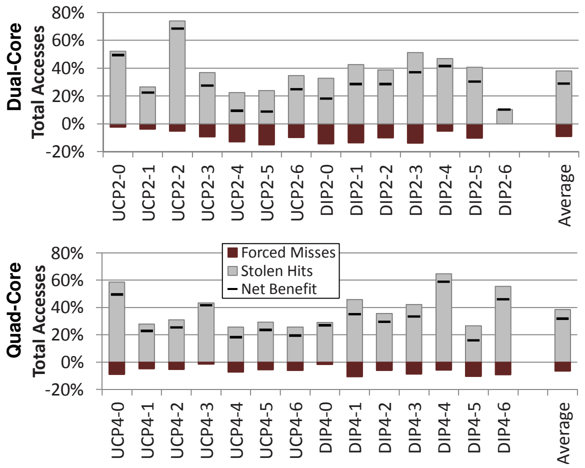

Figure 7: Number of hits on lines stolen from other cores, number of misses induced by having lines stolen by other cores, and the difference.

Configuration	p_{prom}	p_{stream}	Promotion
Baseline PIPP	3/4	1/64	0/+1
50-50 Probability	1/2	1/2	0/+1
Always MRU Promotion	1	1	MRU
Stochastic MRU Promotion	3/4	1/64	0/MRU
Always Promote by +1	1	1	+1
No-Stream	3/4	3/4	0/+1

Table 2: Variants on PIPP for studying the importance of different components of the algorithm.

in the cache and the remaining hits would have occurred anyway. Nevertheless, the results show that the lines that are stolen are indeed useful, and that the theft of lines from other cores do not introduce that many more misses. This result is intuitive in that when a line gets stolen, the evicted line is already in the lowest priority position in the eviction order, which tends to be correlated with a lack of near-future reuses. That is, when one core steals a line from another core, there is a good chance that the line would have been evicted soon anyway. PIPP can effectively make use of these dead lines to improve the efficiency of the cache.

5.2 PIPP Parametric Sensitivity Analysis

Our PIPP mechanism contains several parameters that can be chosen by the computer architect. These include decisions about promotion policies, whether to make PIPP aware of streaming application behaviors, and setting the two different probabilities p_{prom} and p_{stream}. We first consider several variants on the PIPP mechanism. These are summarized in Table 2, with the original PIPP included for reference. These variants were chosen to demonstrate the importance of different design choices for PIPP. Figure 8(a) shows the performance degradation (higher is worse) for the harmonic mean of the weighted IPC speedup normalized to the stream-sensitive PIPP. Omission of any of these features can lead to an 11.6% performance penalty.

We also explored the sensitivity of the results to different choices of p_{prom} and p_{stream}. It would not be desirable to have to carefully retune these probabilities for different workloads or new processor designs. Figure 8(b) shows the performance results when all parameters for the original stream-sensitive PIPP are held constant except for p_{prom}. The results are normalized to the original case where $p_{prom} = \frac{3}{4}$. Note that for any similar values for p_{prom}, the overall performance change is less than 1%. Similarly, Figure 8(c) modifies p_{stream} while keeping all other parameters the same. The original value for p_{stream} was $\frac{1}{128}$, but these result show that one can change this probability over a reasonably wide range and the overall performance changes are quite small. These results are good in that they suggest that the probability parameters need not be chosen too carefully; any reasonable values will result in good performance.

6. IMPLEMENTATION ISSUES

In this section, we briefly discuss some of the remaining hardware overhead required to implement PIPP, and then present a simple optimization to eliminate much of the remaining costs.

6.1 Hardware Overhead

One of the critical components of many previously proposed cache partitioning approaches is the mechanism for estimating or predicting what the benefit/cost would be of providing a core with more/fewer ways. This information provides the input for the partitioning mechanism to make its decisions. In particular, the implementation of PIPP evaluated thus far in this paper simply makes use of the same Shadow Tag approach used in the UCP work and discussed in Section 2. As described earlier, the *Utility Monitor* (UMON) maintains one set of shadow tags per core to track what the cache's contents would be if each core had exclusive access to the cache. Depending on the recency positions of the hits observed in the shadow tags, utility monitoring counters are updated to create the corresponding utility curves such as those shown in Figure 1.

The UMON shadow tags represent additional overhead that would not be required in a conventional unmanaged cache, although the use of Dynamic Set Sampling (DSS) reduces the overhead by a significant amount. For example, a cache with 4096 sets shared by four cores requires 1.1MB to store the unsampled shadow tags (assuming 36-bit tag entries), whereas with DSS the overhead is reduced to only 9.1KB (assuming 32 sampled sets). Suh et al. proposed to estimate marginal utility based on the actual hit position in the shared cache [40]. A hit in recency position i in the real cache causes the i^{th} marginal utility counter to get incremented. For example, a set containing the lines {A, B, 1, C} where A is the most recently used line and C is the least, and A, B and C belong to core$_0$, a hit on line C causes core$_0$'s fourth counter to be incremented because C is located in the fourth most-recently used position, but if core$_0$ had the entire cache to itself, C would only be in the third most-recently used spot. It is clear that while this approach does not incur any additional storage overhead for shadow tags, the contents of the marginal gain counters may be compromised, leading to inaccurate utility curves [36].

6.2 Elimination of Shadow Tags

We propose In-Cache Estimation Monitors (ICEmon) which can be thought of as a hybrid of Suh et al.'s approach, Qureshi and Patt's UMON, and Qureshi et al.'s *leader set* idea [35]. A small number of sets use a modified cache management policy to facilitate utility tracking. Figure 9 shows an example cache with eight sets, where one set is reserved for tracking core$_0$ and another for core$_1$. Set zero estimates the utility for core$_0$. For all accesses by core$_0$, this set is managed in a conventional LRU manner (evict the LRU line, insert at the MRU position, promote to the MRU position on a hit). We then enforce a *private monitoring boundary* (PMB) where lines inserted/accessed by other cores are not permitted to obtain a recency position greater than PMB. Core$_1$ continues to follow its insertion and promotion policies, with the modification

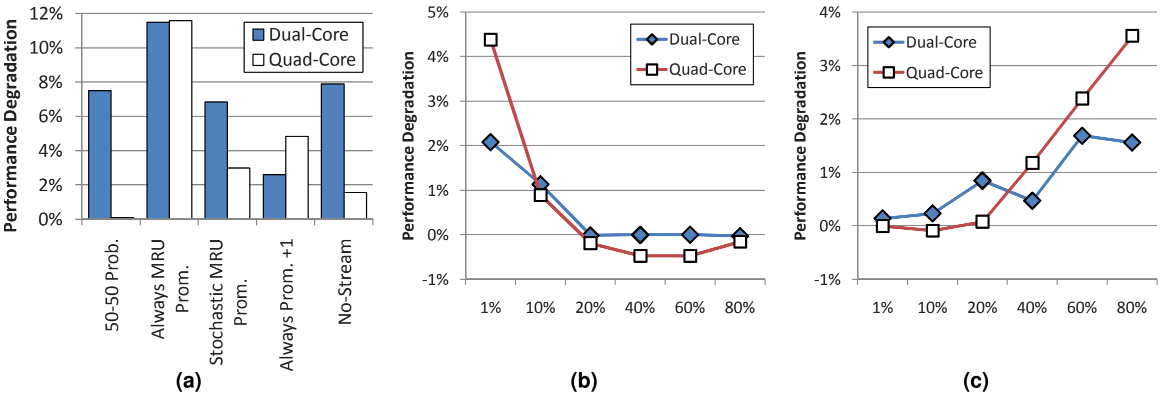

Figure 8: Performance sensitivity to different (a) PIPP design choices, (b) values for p_{prom} and (c) values for p_{stream}.

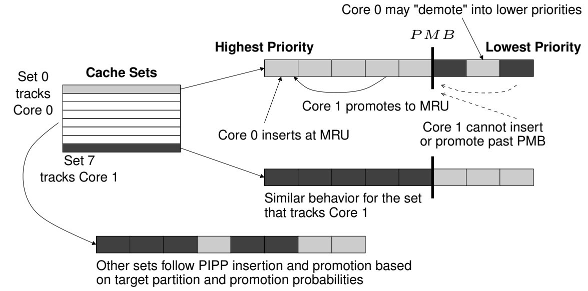
Core 0 may "demote" into lower priorities
PMB
Highest Priority
Lowest Priority
Set 0 tracks Core 0
Cache Sets
Core 1 promotes to MRU
Core 0 inserts at MRU
Core 1 cannot insert or promote past PMB
Set 7 tracks Core 1
Similar behavior for the set that tracks Core 1
Other sets follow PIPP insertion and promotion based on target partition and promotion probabilities

Figure 9: Example organization of the In-Cache Estimation Monitors (ICEmon) for two cores sharing an eight-set, eight-way cache.

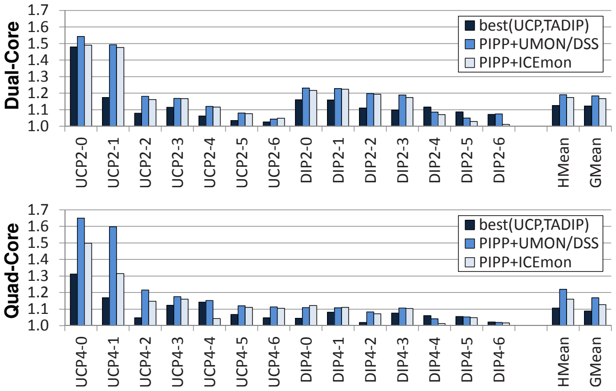

Figure 10: Weighted speedup of PIPP with different utility tracking mechanisms.

that insertion and promotion are capped at PMB, as shown in Figure 9. Set seven monitors core$_1$ in a similar fashion with core$_0$ not being able to cross the PMB.

By dedicating w–PMB ways (in a w-way cache) to the monitored core, our ICEmon is able to accurately track the marginal gain updates for the first w–PMB ways. These ways are typically the most important in that they account for the majority of the area underneath the marginal utility curves (i.e., the first few ways usually provide the most hits). For the remaining w ways, the utility tracking may have some errors because the lines accessed by the other cores will pollute and perturb the recency ordering of the core being tracked. This can create problems similar to Suh et al.'s monitoring scheme, but the impact on ICEmon is much less because these tracking errors only impact the PMB-least recently used lines which typically account for a much smaller number of the total accesses.

Figure 10 shows the performance of PIPP using UMON with DSS, and PIPP using our proposed ICEmon scheme. For reference, we also include the performance of the *best* of UCP and TADIP. The PIPP+ICEmon configuration uses 32 leader sets per core to estimate the marginal gain counters. For each leader set, PMB is set to four ways.

Our ICEmon mechanism for estimating the marginal utilities does introduce some error in the partitioning decisions, as shown by the difference in the PIPP+UMON/DSS and PIPP+ICEmon results. For the dual-core configurations, the small amount of error introduced by ICEmon, as compared to UMON/DSS, results in an average performance loss of only 1.4% (for both means); for the quad-core results, the performance loss is a more modest 3.6%/4.8% for the harmonic/geometric mean. Note that even with the reduction in performance due to the less accurate ICEmon scheme, the average performance across all of the dual-core and quad-core workloads still exceeds the best of both UCP and TADIP.

The main reason that the quad-core results are not better is that we did not recalibrate the PMB setting, and so in the leader sets, *three* cores worth of cache lines are forced to share the four ways which results in more error in tracking the marginal utility of the last few ways. We were also concerned that a hot set that mapped to a leader set could cause significant pathological behaviors. We considered a variant of PIPP/ICEmon where during each sampling interval, the leader sets are rotated so *all* sets in the cache take turns as leaders. It turns out that this hot-set phenomenon is not important, at least for our workloads, and this rotating leader-set approach made very little difference on performance. Table 3 summarizes the storage overhead required for the UMON and ICEmon approaches for estimating marginal utility.

7. RELATED WORK

Apart from the most relevant studies already discussed in the earlier sections of this paper, there exists a large body of additional work on way-partitioned caches and other techniques for managing shared caches.

	Shadow Tag Storage	UMON Counters	Storage 4MB/16-way L2, 4 cores
UMON (*no DSS*)	$swtN$	wN	1.1 MB
UMON (*w/ DSS*)	$\alpha swtN$	wN	9.1 KB
ICEmon	0	wN	20 bytes

Table 3: Summary of overheads for different marginal utility estimation schemes. Example storage overhead assumes s=4096 sets, w=16 ways, N=4 cores, t=36 bits per shadow tag entry, $\alpha=\frac{1}{128}$ (DSS sampling rate), UMON counter size=10 bits.

Several research efforts by multiple groups have proposed variants of way-partitioned caching to provide Quality of Service (QoS) in shared-cache multi-cores [12, 17, 18, 24]. PIPP in of itself does not provide or enforce any explicit QoS guarantees. It may be possible to allow the operating system to specify partitions and/or export utility monitoring information back to software to provide higher-level control over managing QoS among cores, but such work is beyond the scope of this paper.

There have also been a few proposed schemes that relax the strict "way-partitioned" organization of the cache. Rafique et al. proposed a mechanism for enforcing OS-specified partitions or *quotas* [37], and they introduce the concept of *reluctance* that effectively enables cores to steal lines from other cores (possibly exceeding their quotas) when the other cores are not making effective use of their own cache allocations. Chang and Sohi proposed Multiple Time-sharing Partitions (MTP) where the partitioning decisions can be adjusted in both space and time [7]. For example, consider two cores sharing an eight-way cache where each core needs six ways to hold the majority of their working sets. Chang and Sohi observe that rather than providing a "fair" allocation of four ways per core, it is sometimes better to simply allow one core to receive an unfair allocation (e.g., six ways), and then later allow the other core an *equally* unfair allocation. Although on a per-timeslice basis, the partitioning is unfair, across multiple timeslices fairness is still maintained. Such an approach could potentially be adapted to PIPP by simply varying the target partition allocations from one timeslice to the next. Srikantaiah et al. observed that in shared caches, interference between cores can induce "compulsory inter-cache" misses (compulsory misses due to unrelated accesses from other cores) [38], and they propose dynamic set pinning as a means to prevent such misses.

Dead-line prediction (also called dead-block prediction) attempts to anticipate the last touch to a given line [1, 16, 22, 26, 29]. If the system is confident that the line will not be accessed again, then the line can be given the highest priority for replacement, thereby minimizing the amount of time the otherwise useless line resides in the cache. Other proposals predict the deadness of a line to save power by either turning off dead lines or putting them into a drowsy state [1, 10, 20]. PIPP does not target dead lines in a fashion as explicit as these other works, but part of PIPP's benefit effectively comes from the reduction of the residency times of dead cache lines.

8. CONCLUSIONS

Many previous studies have proposed a variety of mechanisms to improve the performance of shared last-level caches by exploiting a variety of common memory access behaviors. In this work, we have introduced a single unified technique that can provide the benefits of capacity management, adaptive insertion and inter-core capacity stealing. By covering multiple types of memory behaviors, our proposed PIPP scheme delivers higher performance than previously proposed techniques.

This work opens several future directions for research. First, there are likely many additional promotion policies that can be explored, and more work is needed to provide a deeper understanding of how promotion policies interact with insertion and evictee selection policies. Despite the fact that the ICEmon eliminates the shadow tag overhead, there exists the overhead of the partitioning logic itself (the hardware that converts the utility curves into actual partition allocations). Generalizations of ICEmon, perhaps incorporating more Set Dueling concepts, may enable the complete elimination of the partitioning logic.

Acknowledgments

Funding and equipment were provided by a grant from Intel Corporation, and funding was provided from the National Science Foundation under Grant No. 0702275. We also thank the anonymous reviewers for their constructive feedback and Aamer Jaleel for assistance with TADIP.

9. REFERENCES

[1] J. Abella, A. González, X. Vera, and M. F. P. O'Boyle. IATAC: A Smart Predictor to Turn-Off L2 Cache Lines. *Trans. on Architecture and Code Optimization*, 2(1):55–77, Mar. 2005.

[2] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Micro Magazine*, pages 59–67, Feb. 2002.

[3] D. A. Bader, Y. Li, T. Li, and V. Sachdeva. BioPerf: A Benchmark Suite to Evaluate High-Performance Computer Architecture of Bioinformatics Applications. In *Proc. of the IEEE Int. Symp. on Workload Characterization*, pages 163–173, Austin, TX, USA, Oct. 2005.

[4] M. Behar, A. Mendelson, and A. Kolodny. Trace Cache Sampling Filter. In *Proc. of the 14th Int. Conference on Parallel Architectures and Compilation Techniques*, pages 255–266, St. Louis, MO, USA, Sep. 2005.

[5] D. S. Bolme, M. M. Strout, and J. R. Beveridge. FacePerf: Benchmarks for Face Recognition Algorithms. In *Proc. of the IEEE Int. Symp. on Workload Characterization*, Boston, MA, USA, Oct. 2007.

[6] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting Inter-Thread Cache Contenton on a Chip Multi-Processor Architecture. In *Proc. of the 11th Int. Symp. on High Performance Computer Architecture*, pages 340–351, San Francisco, CA, USA, Feb. 2005.

[7] J. Chang and G. Sohi. Cooperative Cache Partitioning for Chip Multiprocessors. In *Proc. of the 21st Int. Conference on Supercomputing*, pages 242–252, Seattle, WA, June 2007.

[8] D. Chiou. *Extending the Reach of Microprocessors: Column and Curious Caching*. PhD thesis, Massachusettts Institute of Technology, 1999.

[9] J. Doweck. Inside Intel Core Microarchitecture and Smart Memory Access. White paper, Intel Corporation, 2006. http://download.intel.com/technology/architecture/sma.pdf.

[10] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy Caches: Simple Techniques for Reducing Leakage Power. In *Proc. of the 29th Int. Symp. on Computer Architecture*, pages 148–157, Anchorage, AK, USA, May 2002.

[11] H. Ghasemzadeh, S. Mazrouee, and M. R. Kakoee. Modified Pseudo LRU Replacement Algorithm. In *Proc. of the Int. Symp. on Low Power Electronics and Design*, pages 27–30, Potsdam, Germany, Mar. 2006.

[12] F. Guo, Y. Solihin, L. Zhao, and R. Iyer. A Framework for Providing Quality of Service in Chip Multi-Processors. In *Proc. of the 40th Int. Symp. on Microarchitecture*, Chicago, IL, Dec. 2007.

[13] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A Free, Commerically Representative Embedded Benchmark Suite. In *Proc. of the 4th Workshop on Workload Characterization*, pages 83–94, Austin, TX, USA, Dec. 2001.

[14] G. Hamerly, E. Perelman, J. Lau, and B. Calder. SimPoint 3.0: Faster and More Flexible Program Analysis. In *Proc. of the Workshop on Modeling, Benchmarking and Simulation*, Madison, WI, USA, June 2005.

[15] L. R. Hsu, S. K. Reinhardt, R. R. Iyer, and S. Makineni. Communist, Utilitarian, and Capitalist Cache Policies on CMPs: Caches as a Shared Resource. In *Proc. of the 15th Int. Conference on Parallel Architectures and Compilation Techniques*, pages 13–22, Seattle, WA, USA, Sep. 2006.

[16] Z. Hu, M. Martonosi, and S. Kaxiras. Timekeeping in the Memory System: Predicting and Optimizing Memory Behavior. In *Proc. of the 29th Int. Symp. on Computer Architecture*, pages 209–220, Anchorage, AK, USA, May 2002.

[17] R. Iyer. CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms. In *Proc. of the Int. Conference on Supercomputing*, Saint-Malo, France, June 2004.

[18] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. QoS Policies and Architecture for Cache/Memory in CMP Platforms. In *Proc. of the ACM SIGMETRICS*, San Diego, CA, USA, June 2007.

[19] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. S. Jr., and J. Emer. Adaptive Insertion Policies for Managing Shared Caches. In *Proc. of the 17th Int. Conference on Parallel Architectures and Compilation Techniques*, 2007.

[20] S. Kaxiras, Z. Hu, and M. Martonosi. Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power. In *Proc. of the 28th Int. Symp. on Computer Architecture*, pages 240–251, Göteborg, Sweden, June 2001.

[21] M. Kharbutli and Y. Solihin. Counter-Based Cache Replacement Algorithms. In *Proc. of the Int. Conference on Computer Design*, pages 61–68, San Jose, CA, USA, Oct. 2005.

[22] M. Kharbutli and Y. Solihin. Counter-Based Cache Replacement and Bypassing Algorithms. *Trans. on Computers*, 57(4):433–447, Apr. 2008.

[23] S. Kim, D. Chandra, and Y. Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *Proc. of the 13th Int. Conference on Parallel Architectures and Compilation Techniques*, pages 111–122, Antibes Juan-les-Pins, France, Sep. 2004.

[24] S. Kim, D. Chandra, and Y. Solihin. Fair Caching in a Chip Multi-Processor Architecture. In *Proc. of the IBM P=AC^2 Conference*, Yorktown Heights, NY, USA, Oct. 2004.

[25] J. D. Kron, B. Prumo, and G. H. Loh. Double-DIP: Augmenting DIP with Adaptive Promotion Policies to Manage Shared L2 Caches. In *Proc. of the Workshop on Chip Multiprocessor Memory Systems and Interconnects*, Beijing, China, June 2008.

[26] A.-C. Lai, C. Fide, and B. Falsafi. Dead-Block Prediction & Dead-Block Correlating Prefetchers. In *Proc. of the 28th Int. Symp. on Microarchitecture*, pages 144–154, Göteborg, Sweden, June 2001.

[27] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communication Systems. In *Proc. of the 30th Int. Symp. on Microarchitecture*, pages 330–335, Research Triangle Park, NC, USA, Dec. 1997.

[28] J. Lin, Q. Lu, X. Ding, Z. Zhang, and P. Sadayappan. Gaining Insights into Multicore Cache Partitioning: Bridging the Gap between Simulation and Real Systems. In *Proc. of the 14th Int. Symp. on High Performance Computer Architecture*, pages 367–378, Salt Lake City, UT, USA, Feb. 2008.

[29] H. Liu, M. Ferdman, J. Huh, and D. Burger. Cache Bursts: A New Approach for Eliminating Dead Blocks and Increasing Cache Efficiency. In *Proc. of the 41st Int. Symp. on Microarchitecture*, pages 222–233, Lake Como, Italy, Nov. 2008.

[30] G. H. Loh, S. Subramaniam, and Y. Xie. Zesto: A Cycle-Level Simulator for Highly Detailed Microarchitecture Exploration. In *Proc. of the Int. Symp. on Performance Analysis of Systems and Software*, Boston, MA, USA, Apr. 2009.

[31] K. Luo, J. Gummaraju, and M. Franklin. Balancing Throughput and Fairness in SMT Processors. In *Proc. of the 2001 Int. Symp. on Performance Analysis of Systems and Software*, pages 164–171, Tucson, AZ, USA, Nov. 2001.

[32] R. Narayanan, B. Ozisikyilmax, J. Zambreno, G. Memik, and A. N. Choudhary. MineBench: A Benchmark Suite for Data Mining Workloads. In *Proc. of the IEEE Int. Symp. on Workload Characterization*, pages 182–188, San Jose, CA, USA, Oct. 2006.

[33] M. K. Qureshi, , D. Lynch, O. Mutlu, and Y. N. Patt. A Case for MLP-Aware Cache Replacement. In *Proc. of the 33rd Int. Symp. on Computer Architecture*, pages 167–178, Boston, MA, USA, June 2006.

[34] M. K. Qureshi. Dynamic Spill-Accept for Scalable High-Performance Caching in CMPs. In *Proc. of the 15th Int. Symp. on High Performance Computer Architecture*, Raleigh, NC, USA, Feb. 2009.

[35] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. S. Jr., and J. Emer. Adaptive Insertion Policies for High-Performance Caching. In *Proc. of the 34th Int. Symp. on Computer Architecture*, pages 381–391, San Diego, CA, USA, June 2007.

[36] M. K. Qureshi and Y. N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proc. of the 39th Int. Symp. on Microarchitecture*, pages 423–432, Orlando, FL, Dec. 2006.

[37] N. Rafique, W.-T. Lin, and M. Thottethodi. Architectural Support for Operating System-Driven CMP Cache Management. In *Proc. of the 15th Int. Conference on Parallel Architectures and Compilation Techniques*, pages 2–12, Seattle, WA, USA, Sep. 2006.

[38] S. Srikantaiah, M. Kandemir, and M. J. Irwin. Adaptive Set-Pinning: Managing Shared Caches in Chip Multiprocessors. In *Proc. of the 13th Symp. on Architectural Support for Programming Languages and Operating Systems*, Seattle, WA, USA, Mar. 2009.

[39] H. S. Stone, J. Tuerk, and J. L. Wolf. Optimal Paritioning of Cache Memory. *Trans. on Computers*, 41(9):1054–1068, Sep. 1992.

[40] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic Partitioning of Shared Cache Memory. *Jour. of Supercomputing*, 28(1):7–26, 2004.

[41] T. Y. Yeh, P. Faloutsos, S. J. Patel, and G. Reinman. ParallAX: an Architecture for Real-Time Physics. In *Proc. of the 34th Int. Symp. on Computer Architecture*, pages 232–243, San Diego, CA, USA, June 2007.