The Gradient-Based Cache Partitioning Algorithm

WILLIAM HASENPLAUGH, PRITPAL S. AHUJA, AAMER JALEEL, SIMON STEELY JR., and JOEL EMER, Intel

This paper addresses the problem of partitioning a cache between multiple concurrent threads and in the presence of hardware prefetching. Cache replacement designed to preserve temporal locality (e.g., LRU) will allocate cache resources proportional to the miss-rate of each competing thread irrespective of whether the cache space will be utilized [Qureshi and Patt 2006]. This is clearly suboptimal as applications vary dramatically in their use of recently accessed data. We address this problem by partitioning a shared cache such that a global goodness metric is optimized. This paper introduces the Gradient-based Cache Partitioning Algorithm (GPA), whose variants optimize either hitrate, total instructions per cycle (IPC) or a weighted IPC metric designed to enforce Quality of Service (QoS) [Iyer 2004]. In the context of QoS, GPA enables us to obtain the maximum throughput of low-priority threads, while ensuring high performance on high-priority threads. The GPA mechanism is robust, low-cost, integrates easily with existing cache designs and improves the throughput of an in-order 8-core system sharing an 8MB L3 cache by ∼14%.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles—*Cache memories*

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Cache replacement, insertion policy, dynamic cache partitioning, dynamic control, hill climbing, gradient descent, chernoff bound, adaptive caching

ACM Reference Format:
Hasenplaugh, W., Ahuja, P. S., Jaleel, A., Steely Jr., S., and Emer, J. 2012. The gradient-based cache partitioning algorithm. ACM Trans. Architec. Code Optim. 8, 4, Article 44 (January 2012), 21 pages.
DOI = 10.1145/2086696.2086723 http://doi.acm.org/10.1145/2086696.2086723

1. INTRODUCTION

Last level cache replacement mechanisms designed to preserve temporal locality (e.g., LRU) will allocate cache resources proportional to the miss-rate of each competing thread irrespective of whether the cache space will be utilized [Qureshi and Patt 2006]. This is suboptimal, as applications vary dramatically in their use of recently accessed data, and is due to a variety of reasons: the extraction of all useful temporal locality by upper level caches, effective prefetching, streaming access patterns, etc. Unregulated shared caches, employing a temporal replacement algorithm (e.g., LRU), are also subject to other hazards like prefetcher pollution and unfairness [Kim et al. 2004], as the cache partition for each thread is linear in the relative rates of replacement. Consider the example of a system administrator with a high-priority task and a multicore system. The administrator may be tempted to run the high-priority task in isolation because she is powerless to stop a background thread from thrashing the shared cache and, unwittingly, slowing down her high-priority thread. Alternatively, she may opt to buy a machine with private L3 caches and thus protect her high-priority thread,

Authors' address: W. Hasenplaugh, P. S. Ahuja, A. Jaleel, S. Steely Jr., and J. Emer, Intel Corporation, 77 Reed Road, Hudson, MA 01749; email: william.c.hasenplaugh@intel.com.

DOI 10.1145/2086696.2086723 http://doi.acm.org/10.1145/2086696.2086723

though at the expense of not being able to allocate most of the total cache area to the high-priority thread. If she were designing her own shared-cache multi-core system, she may be tempted to implement a version of *way partitioning* [Suh et al. 2004], where each thread has some dedicated ways at their disposal, and there is a communal pool of ways allocated like a normal shared cache. Again, she will guarantee some modicum of Quality of Service (QoS) to her high-priority thread as with a private cache, though she will have no power to grant the high-priority thread the extra shared system resources. Making matters worse, the *way partitioning* approach allows fewer threads than ways (8-32 in typical systems and holding), whereas the number of threads increases with Moore's law or greater.

We present an approach to this problem that nominally optimizes the allocation of cache capacity such that a *global goodness* metric is optimized, while exposing QoS controls to the system administrator. Further, our algorithm utilizes an adaptive *insertion policy* and a simple temporal replacement algorithm, *One-Bit LRU* (OBL) [Al-Zoubi 2004], as opposed to using a complex replacement algorithm. We do this to avoid allocating additional state, on which a complex replacement decision would be made, and to be more sensitive to the replacement decision circuitry on the critical path. Cache designs which select victims upon insertion would be quite sensitive to the complexity of the circuit that selects the victim. By contrast, our use of an adaptive insertion policy allows us to insert a cache block such that it will be replaced soon if we suspect that it is useless data. For instance, if we suspect that a prefetcher is polluting the cache, we can mark those prefetches as next to be replaced, or *vulnerable*, and allow other useful data to live longer in the cache. In this way we indirectly control the partition of each thread; by controlling the rate at which each thread is chosen for replacement via vulnerable insertion.

1.1. Contributions and Related Work

There are a number of papers that have influenced this work, giving insight into dynamic behavior of shared caches and application behavior, just a few of which are Guo et al. [2007], Dhodapkar and Smith [2002], Qureshi et al. [2007], Chiou [1999] and Stone et al. [1992]. There are also several excellent cache replacement schemes aimed at improving cache hitrate and/or reducing cache latency in shared caches: Chang et al. [2006] makes proximity-aware caching decisions, Qureshi et al. [2006] exploits memory-level parallelism and Jaleel et al. [2008] opportunistically evicts cache blocks that are unlikely to be useful, making room for more useful data. More recently, Jaleel et al. [2010] have presented a baseline replacement policy, RRIP, which is more robust and higher-performance than true LRU, while maintaining a modest hardware cost. Albonesi [1999] evaluates the performance of an application as a function of cache space online and adaptively turn off ways, when appropriate, in order to save power. Another branch of cache management that has influenced this work is related to Quality of Service (QoS) enforcement: Guo et al. [2006], Rafique et al. [2006], Guo et al. [2007], Iyer et al. [2007] and Nesbit et al. [2007]. The QoS literature is largely dominated by calls for architectural support to guarantee minimum levels of performance or cache space in order to provide QoS. These techniques are primarily motivated by Virtual Machine environments, where system administrators are faced with unpredictable interactions of many unrelated applications sharing system resources.

We propose the Gradient-based Cache Partitioning Algorithm (GPA) in an effort to provide a scalable, robust, low-complexity adaptive mechanism for partitioning a shared cache between competing threads and prefetchers. In addition, we expose controls to the System Administrator to customize the performance metric to optimize, including provisions to complement OS-level QoS mechanisms. Finally, there are three approaches which most closely resemble GPA: Modified LRU (mLRU) by Suh

et al. [2004], Utility Cache Partitioning (UCP) by Qureshi and Patt [2006] and Cache-Partitioning Aware Replacement (CPAR) by Dybdahl et al. [2006]. Each of these three replacement policies periodically allocate cache space in quanta of ways, such that they optimize some fixed performance metric (e.g., Reduction in Misses, Throughput etc.). In addition, each requires a modification to the replacement decision such that the victim selection circuit is dependent on the state of the set and the identity of the thread requesting the cache block, which is a consequence of the method of maintaining each thread's cache allocation. We will take the opportunity to contrast GPA with these three approaches.

(1) *High Performance*. GPA can match the throughput performance of an in-order 8 core system with 4MB private L3 caches with a shared 8MB L3 cache across several heterogeneous mixes of SPEC2006 workloads.
(2) *Continuous Adjustment*. As opposed to being updated periodically at a constant rate, GPA reacts immediately to program changes and potential performance gains.
(3) *Acceleration*. GPA is a control system, adjusting sooner when there is a larger potential performance gain, due to a dynamic threshold governed by the Chernoff bound.
(4) *Data Classes*. GPA can delineate between each thread's demand and prefetch reference stream, individually controlling the amount of cache space each receives.
(5) *Continuous Partition Granularity*. GPA allocates partitions on cache block granularity, rather than on a per way granularity, without requiring full-associativity or LRU replacement. In addition, GPA is able to allocate cache space quite deliberately, whereas Adaptive Insertion Policies: Set Dueling Thread-Aware (SD-TA) by Jaleel et al. [2008] make a binary decision about whether a thread is useful, allowing the cache space to be greedily allocated among threads deemed to be useful.
(6) *Scalability*. GPA only requires more sets (and not more ways) to scale with number of cores, while UCP, mLRU and CPAR need more ways. Cache sets typically grow linearly with the number cores via banking, so GPA promises to scale indefinitely.
(7) *Flexible Performance Metric Optimization*. GPA can optimize a variety of different metrics (e.g., IPC, MissRate, Weighted IPC etc.) with common hardware. In addition, the weights in a weighted IPC performance metric can be controlled by a system administrator to enable an OS-level QoS system.
(8) *Robust*. In none of the experiments in this paper does GPA fail to outperform an unregulated shared cache in Harmonic Mean of Relative Speedups, generally regarded as a strong metric of fairness [Luo et al. 2001].
(9) *Low Complexity*. GPA is composed of simple logic which passively observes the reference stream and consumes $\sim$5 Bytes of register state per thread, which is negligible, particularly in the face of 1MB (or more) L3 cache per core; Only Qureshi et al. [2007] and Jaleel et al. [2008] have comparably low complexity. In addition, GPA does not impact the victim selection circuit, only adds one mux stage to the install policy circuit and all calculations can be performed well ahead of the install policy decision.

In Section 2, we will describe the GPA algorithm and a hardware implementation of it. Section 3 will detail our simulation infrastructure and results, highlighted by the result that using GPA in an 8-core shared L3 cache configuration with 1MB per core achieves nearly the effective throughput of a system with a private L3 cache system with 4MB per core. Section 4 is a case-study of one mix, illustrating what effect GPA has on cache partitions and how it is able to incorporate QoS controls. Finally, we include an appendix where we derive the concepts underlying GPA more rigorously.

2. GRADIENT-BASED CACHE PARTITIONING ALGORITHM

We present the Gradient-based Cache Partitioning Algorithm (GPA) which addresses the problem of cache partitioning between different threads and prefetchers by applying the gradient descent algorithm, otherwise known as hill-climbing. The hardware will continuously monitor whether it is better for the whole system to allocate extra cache space to one thread by taking it from the others, or vice-versa. This does not require a large data structure; rather, the hardware will evaluate the state of the cache itself. Each thread will have a dynamically controlled *throttle* associated with it which dictates the fraction of that thread's memory references that will be installed *vulnerably*, or in a deprecated manner (equivalent to β from the Bimodal Fill Policy [Qureshi et al. 2007]). Recall that a cache block installed vulnerably will be a candidate for replacement earlier than if it had been installed normally. So, all other things being equal, if a thread's throttle is lower, the thread's effective share of the cache (or cache partition) will tend to be smaller. In addition, each thread will have $\sim$5 Bytes of register state associated with it. The GPA hardware passively monitors the sequence of cache references to decide when each thread is over- or underallocated, moving the associated throttle in the appropriate direction. Whenever a cache block is inserted into the cache, the insertion mechanism is advised by the GPA module as to whether the block's replacement state should be vulnerable or not, according to the throttle. Quite conveniently, the hardware continuously and indirectly adjusts the partition for each thread without having to intimately comprehend what the other threads are doing, yet it manages to navigate the partition of the cache to a globally optimum state.

The value of any thread's references can be artificially changed to enforce Quality of Service (QoS). Thus, we are able to re-allocate cache space if the high-priority threads are not making good use of it, but they will more easily maintain a larger partition if they are indeed utilizing it. Finally, GPA achieves excellent cache partitioning among a practically unbounded number of threads while being very conservative in hardware and non-invasive to the critical paths of existing typical cache designs.

2.1. Determining the Throttles

We have discussed the idea that each thread has a throttle which dictates what fraction of its cache misses will be installed vulnerably, indirectly controlling the cache allocation for that thread. So, how do we learn what the throttle should be for each thread? We could search for the optimal set of throttle values; though, with each coming from a continuous range, the combinatorics would be staggering. Instead, we take a different approach. To understand how we determine the throttles, consider the following example. Imagine we have 2 threads, *t1* and *t2*, running on a system with a shared cache. We have predetermined (whether randomly, by profiling, or some other means), that thread *t1* should get a fraction f of the cache and that *t2* should get the rest, $1 - f$, of the cache. Now, we do not know if f and $1 - f$ are the fractions that would optimize the combined hitrate of *t1* and *t2*. Thus, we would like to perform two experiments, one in which *t1* would get slightly more cache, say $f + \varepsilon$, and another experiment in which *t1* would get slightly less cache, say $f - \varepsilon$. Thread *t2* would get $1 - f - \varepsilon$ and $1 - f + \varepsilon$, respectively, in the two experiments. After running each experiment for a "large" number of cycles, we analyze the overall hitrate of the cache in each experiment. If the overall hitrate is better in the experiment in which *t1* received more cache, we may conclude that *t1* should be given a little more cache than its previous allocation, and *t2,* slightly less. On the other hand, the experiments may indicate that overall hitrate improves when *t1* gets slightly less cache. In either case, we could at this point adjust how much cache to give each thread, and then repeat the experiments, again skewing slightly more and slightly less than the new allocation. Eventually, using this hill-climbing approach, we would arrive at some optimal partitioning.

```
bool InsertVulnerably(Set, Thread, Throttle)
{
  if Set is not part of Thread's Gradient Region
    return rand() > Throttle

  else
    if set is in +ve half of Gradient Region
      return rand() > Throttle + grDelta
    else
      return rand() > Throttle - grDelta
}
```

Fig. 1. Pseudocode illustrating how to skew the throttle and thus the partition of cache.

In practice, we would like to perform these experiments while the threads are running, invisible to the user, and have the shared cache adjust its allocation for each thread dynamically and automatically. We start with the premise that the reference stream to each set in the cache has approximately the same statistics [Qureshi et al. 2006]. Given that assumption, we assign a collection of cache sets, called a *gradient region*, to each thread. The *gradient regions* are non-overlapping (i.e., a cache set may be a member of at most 1 gradient region). Each thread conducts experiments in its own gradient region, skewing the effective throttle a bit higher in one half (we refer to this half as the *positive* half, or *+ve* half) and a bit lower in the other half (the *negative*, or *–ve,* half). Figure 1 lists the pseudocode describing how the hardware would skew the throttle—and thus the partition of cache—when performing the experiments for a thread. Notice that when inserting a new cache block into a set that is not in the requesting thread's gradient region, the hardware does not skew the throttle. Only when inserting a block into a thread's gradient region is the probability of a vulnerable insertion skewed (by a constant value, *grDelta*). Deciding whether a set belonging to a gradient region is in the positive or negative half can be as simple as calculating the set number modulo 2.

Given a thread T, the behavior of all other threads in thread T's gradient region is constant (i.e., there is no skewing of their throttles). So the predominant difference in performance of the two halves of a gradient region can be attributed to the skewing of the corresponding thread's throttle. It is worth emphasizing that the performance we are measuring is that of every thread combined. After all, the experiment we are hoping to conduct in a gradient region is whether the cache space is better spent on the thread under test or on the other threads. For each gradient region, we will keep track of which half is doing better via a counter, the *randomWalkCounter* (see Section 2.2). If one half performs much better than the other, the *randomWalkCounter* will become very negative or very positive. We use the Chernoff Bound (see Appendix E. The Chernoff Bound) to decide when the *randomWalkCounter* is sufficiently extreme to warrant moving the throttle.

The Chernoff Bound gives GPA a kind of acceleration; it adapts the throttle more quickly when there is a strong difference in performance (see Figure 2). The converse is that it is less turbulent when there is a weak difference in performance, providing robustness and stability. While it may seem complicated, the Chernoff Bound resolves to a very simple calculation between *randomWalkCounter* and the *referenceCounter* (see Section 2.2) and does not require information from any other thread. In this way, we are able to achieve a globally optimum cache partition while only performing local calculations.

```
void Reference(Set, Hit, Weight, Thread)
{
  if Set is not in Thread's Gradient Region
    return

  if Set is in +ve half of Gradient Region AND Hit==true
    randomWalkCounter += Weight

  elsif Set is in -ve half of Gradient Region AND Hit==false
    randomWalkCounter += Weight

  else
    randomWalkCounter -= Weight

  referenceCounter += Weight^2

  // test Chernoff Bound, K = sensitivity
  if (randomWalkCounter^2 > K*referenceCounter)
  {
    if ( randomWalkCounter > 0 )
      throttle += updateDelta
    else
      throttle -= updateDelta;

    randomWalkCounter = 0
    referenceCounter = 0
  }
}
```

Fig. 2. Pseudocode illustrating how the hardware would measure the performance of each set map and use the Chernoff Bound to move the throttle. Reference() would only be called for threads whose gradient region corresponds to the set index of the reference address.

2.2. GPA Hardware

Each thread will be equipped with three counters: a *throttle* (∼4 bits), a *referenceCounter* (∼20 bits) and a *randomWalkCounter* (∼12 bits). The *referenceCounter* counts the number of demand references that are made by any thread to its corresponding gradient region. The *randomWalkCounter* is a differential counter, measuring the difference of the performance between the two halves of the gradient region. Figure 3 illustrates the update logic for the three counters and Figure 4 depicts the logic that governs the fill priority decision between normal and vulnerable.

3. EXPERIMENTAL SETUP

We simulate a single-issue, in-order, x86 multi-core processor with three levels of cache, using the same simulation methodology as other comparable algorithms [Qureshi et al. 2007; Jaleel et al. 2008]. The L1 instruction cache is 2-way associative and the L1 data cache is 4-way associative; both are 32KB. The private L2 cache is 12 cycle load-to-use, 8-way associative with 256KB capacity. The L1 and L2 caches use One-Bit-LRU (OBL) replacement in all configurations. We simulate 27 cycle load-to-use 16-way L3 caches of 1MB per core in non-inclusive private and shared configurations of 4 and 8 cores, with memory at a constant 180 cycles. We use OBL as the baseline replacement policy as opposed to true LRU for two reasons. First, it is realistic; true LRU is implausible as a hardware replacement, whereas OBL is used widely in practice and performs nearly as well. Second, we are not testing the replacement policy itself, but rather a dynamic

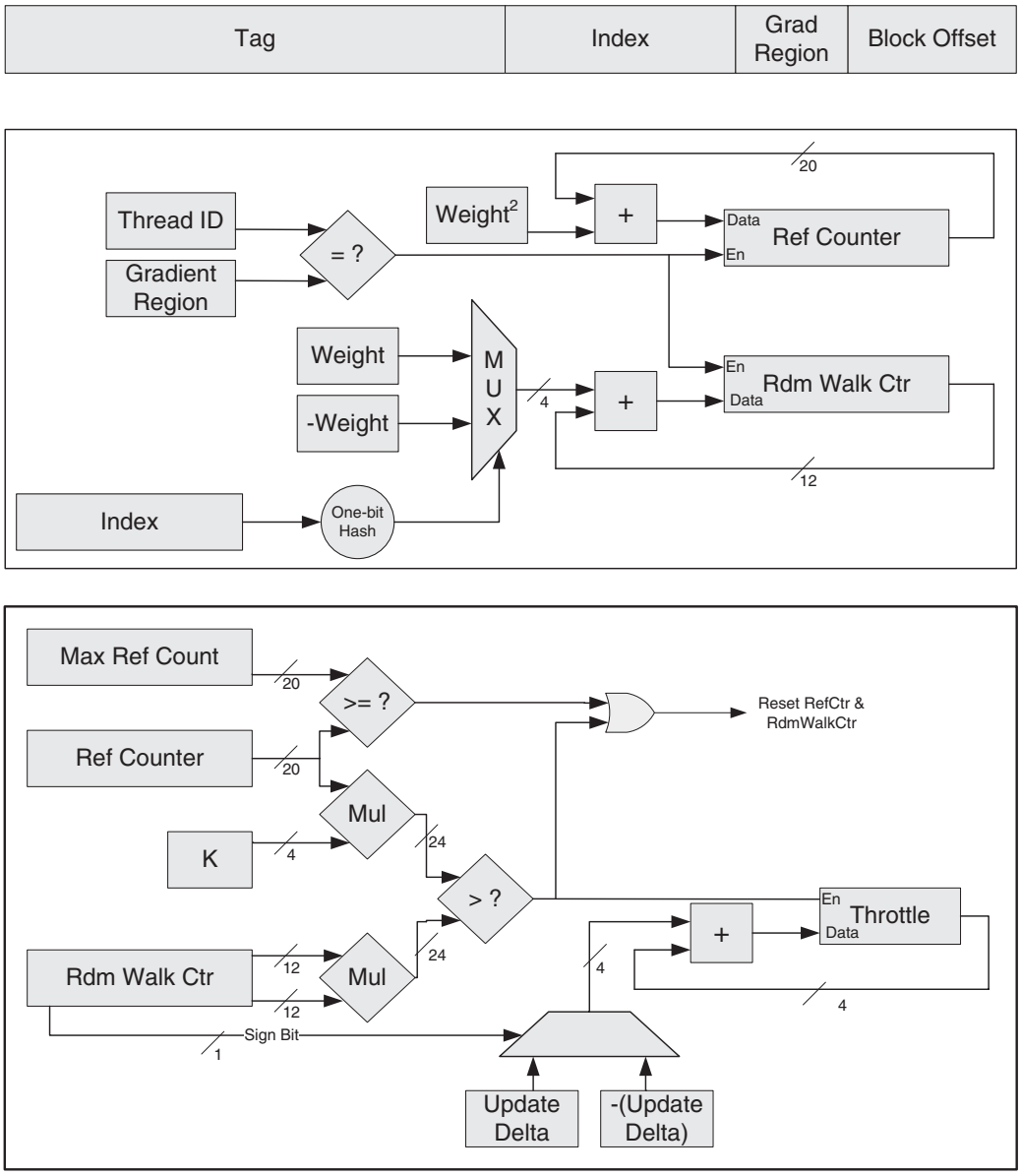

Fig. 3. *referenceCounter* and *randomWalkCounter* update logic.

addition to it, so we wanted to be able to demonstrate a high-performance dynamic replacement scheme with a solid and plausible foundation; after all, a replacement scheme that is built on top of true LRU is not terribly interesting in practice. In addition to the three level cache hierarchy, there is a standard L2 streaming prefetcher [Palacharla et al. 1994], with 10 streams per thread, which installs prefetches at the L2 and L3 levels. We do not model interconnect congestion or a memory controller, as we are primarily investigating the dynamics of cache sharing, though such effects will be

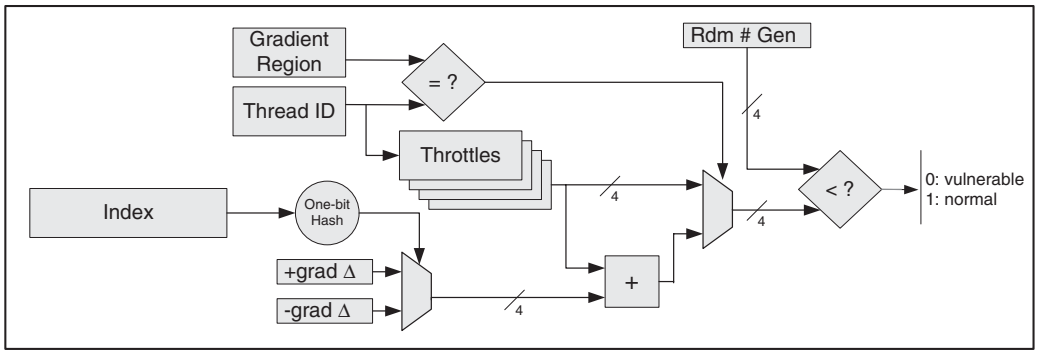

Fig. 4. Vulnerable/normal fill decision logic.

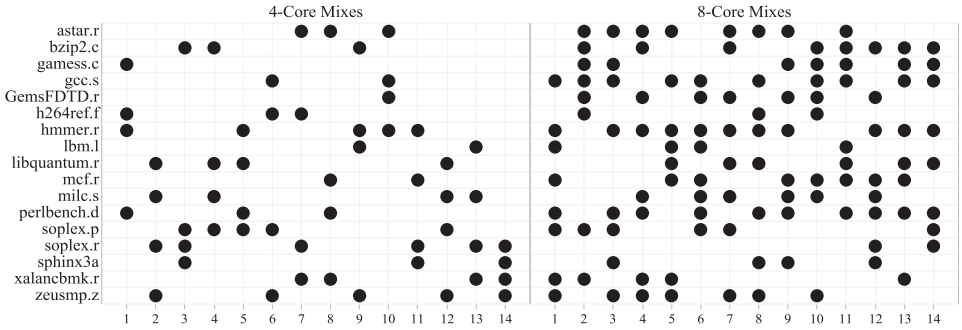

Fig. 5. Workload mixes for 4-core and 8-core experiments, respectively. A black dot denotes the inclusion of a workload (on the corresponding row) in a mix (on the corresponding column).

comprehended in future work. We test various L3 cache replacement strategies with 14 heterogeneous workload mixes each drawn randomly from SPEC2006: {astar.r, bzip2.c, games.c, gcc.s, GemsFDTD.r, h264ref.f, hmmer.r, lbm.l, libquantum.r, mcf.r, milc.s, perlbench.d, soplex.p, soplex.r, sphinx3.a, xalancbmk.r, zeusmp.z}. The specific combinations for 4 and 8-core mixes are denoted in Figure 5. Each mix is run until every thread has completed 500 million instructions, taking the first half of the run (in cycles) as warmup and collecting statistics on the second half. As with all timing-based multi-core simulations, the instruction composition of each mix varies from algorithm to algorithm. We rely on the fact that 500 million instructions are generally sufficient to find a consistent steady-state. The rationale behind using a heterogeneous mix of SPEC2006 workloads is that it is a reasonable proxy for a VM system, running totally unrelated workloads. Extending to cooperative parallel applications and other commercial workloads will be left to future work.

3.1. Throughput Optimization

The baseline for the following experiments is a Private 16-way L3 with 1MB per core under OBL replacement and we compare a variety of shared cache alternatives to it. In particular, we evaluate GPA – MissRate Optimal, which implies that the *weights* from Section 2 (and treated analytically in the appendix) equal 1, and GPA – IPC Optimal, which implies that the *weights* equal T_i, the instantaneous Throughput of the ith thread. For these particular 4-core and 8-core CMP configurations we found the best GPA parameters ($gradientDelta = .5$, $updateDelta = .05$, $K = \ln(\varepsilon^{-2}) = 8$,

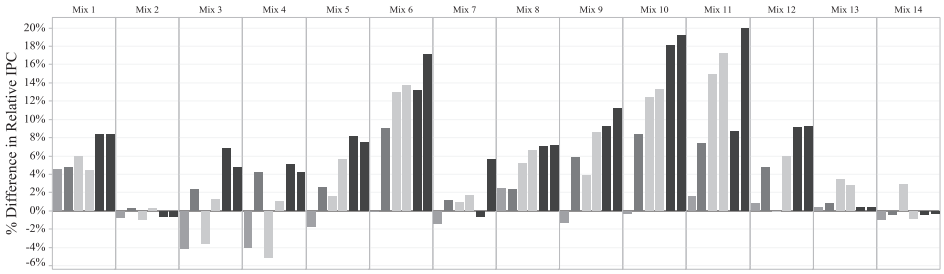

Fig. 6. Total Throughput Speedup (as compared to Private L3 Cache 1MB per core) by workload mix and replacement algorithm from (left to right: OBL, UCP, SD-TU, SD-TA, GPA-MissRate Optimal, GPA-IPC Optimal) for the 4-core configuration.

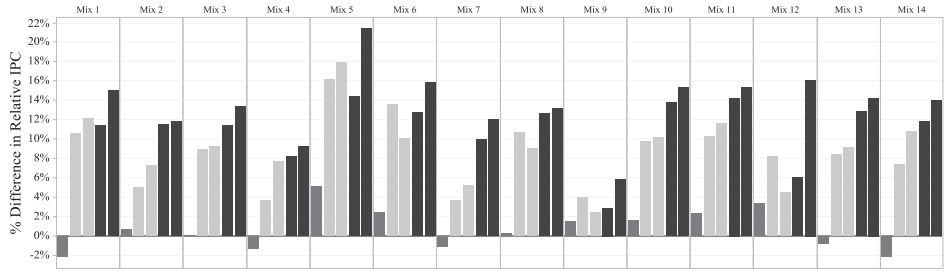

Fig. 7. Total Throughput Speedup (as compared to Private L3 Cache 1MB per core) by workload mix and replacement algorithm (from left to right: OBL, SD-TU, SD-TA, GPA-MissRate Optimal, GPA-IPC Optimal) for the 8-core configuration.

gradientRegions = 8) via a directed exhaustive search. GPA appears to be fairly robust to the choice of those parameters, as more than 50% of the combinations are within 5% throughput performance of the peak combination. Any particular CMP architecture will require such an optimization procedure and we will not go into the details of it here for the sake of brevity, but consider that in silicon these parameters can be optimized on-line and are not bound at design time. In these studies, we assign the demand misses and prefetcher for each thread to its own *throttle*, as though the L2 prefetcher for each thread is a separate thread and GPA manages its cache capacity accordingly. This enables GPA to stop a thread's prefetchers from polluting the cache when necessary.

The results for the 4 thread mixes can be found in Figure 6, broken out by each cache partitioning algorithm managing a 4MB Shared L3 cache. In addition, the 8 thread mixes can be found in Figure 7, though notice that UCP is not present due to the fact that it did not scale to 8 threads in our simulation environment. We use two performance metrics to compare each replacement algorithm: Relative Throughput and Harmonic Mean of Relative Speedups, which are particularly relevant for measuring raw performance and fairness [Luo et al. 2001], respectively. Relative Throughput is defined as the total throughput (i.e., IPC) of the shared cache configuration divided by the total throughput of the baseline private cache configuration. The Harmonic Mean of Relative Speedups is defined as $n/[\sum_{i=1}^{n} s_i^{-1}]$ where s_i is the relative speedup of the ith thread in the shared cache configuration over its performance in the baseline configuration. It generally penalizes an algorithm which tremendously slows down a thread in order to accelerate another.

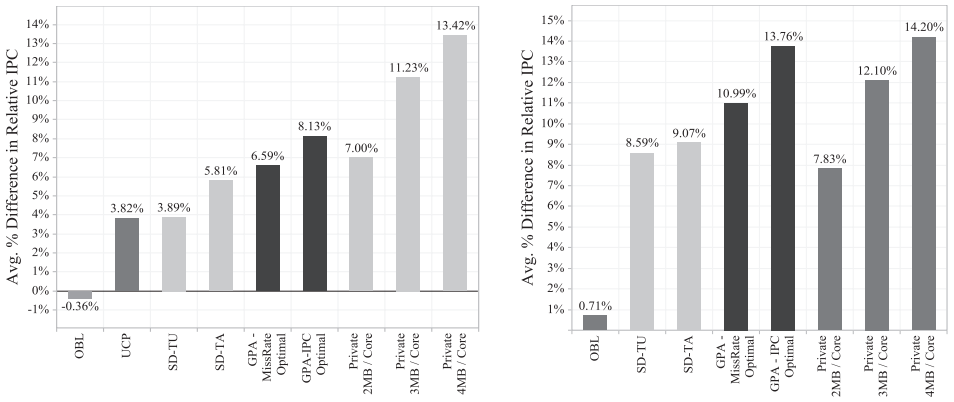

Fig. 8. Summary Speedups of the 4-core (left) and 8-core (right) shared cache experiments, respectively.

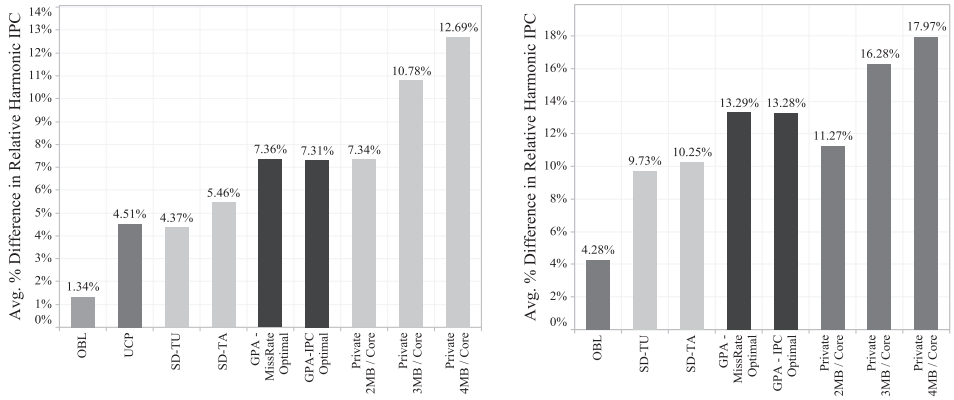

Fig. 9. Summary Speedups of the 4-core (left) and 8-core (right) shared cache experiments, respectively.

Figure 6 shows that GPA, either IPC or MissRate Optimal, is the best throughput performer in 11 of 14 of the 4-core mixes, though in 3 of those 11 mixes, MissRate Optimal slightly outperforms IPC Optimal. We can see in Figure 7 that the 8-core configuration looks particularly promising for the scalability of GPA: whereas GPA is only the best throughput performer in 11 of the 14 4-core mixes, GPA-IPC Optimal is the best performer in every 8-core mix.

A summary of the 4- and 8-core simulations can be found in Figure 8 and Figure 9, plotting average relative throughput and average harmonic mean of relative speedups, respectively, averaged across the 14 mixes. Here, we have also included Private L3 caches of 2MB, 3MB and 4MB per core using OBL replacement, as a point of reference. It is interesting to note that not only does GPA effectively coax the performance of a much larger cache out of 1MB per core, but that much larger caches do not improve performance very much with temporal replacement (e.g., OBL): Quadrupling the cache to get a 14% speedup would seem to be a dubious return on investment.

In addition, the performance of the adaptive schemes (Set Dueling and GPA) appear to be better with 8 cores than with 4, which might suggest that they are harnessing the load-balancing potential of a large shared cache. Even the 8-core shared cache managed by OBL shows a modest improvement over the 4-core case ($\sim$1% throughput

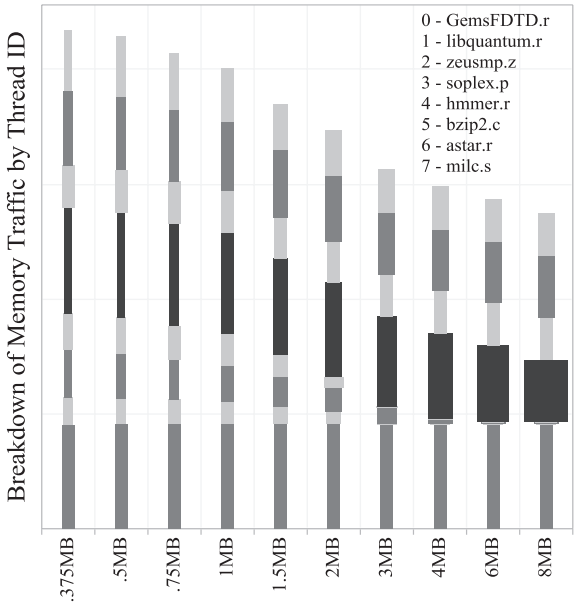

Fig. 10. Breakdown of Memory Traffic by Thread (in the same order from top to bottom as the legend). The segment width represents speedup compared to a 1MB Private L3 cache (1.78 max: soplex.p @ 8MB).

improvement with improvement in harmonic mean of relative speedups of 4%). Further, this data suggests that indeed SD-TU is an improvement over OBL, SD-TA is an improvement over SD-TU and that GPA-IPC Optimal is an improvement over each. Certainly, GPA is a much cheaper way to improve performance than doubling or even quadrupling the L3 cache size, which are the only points that are superior to GPA in Figure 8 and Figure 9.

4. WEIGHTED THROUGHPUT OPTIMIZATION: A CASE-STUDY

Now, we turn to a case-study on the dynamics of cache sharing, to illustrate how GPA can generally eradicate cache pollution and usefully expose QoS controls. Our case study is of an 8-core system running 8-core mix # 7: GemsFDTD.r, libquantum.r, zeusmp.z, soplex.p, hmmer.r, bzip2.c, astar.c and milc.s. We choose soplex.p (shown in black in the following figures) to be designated a "high-priority" thread to illustrate some interesting cache sharing phenomena. We run this mix across multiple Private L3 cache configurations {.375MB, .5MB, .75MB, 1MB, 2MB, 3MB, 4MB, 6MB, 8MB} and an 8MB Shared L3 Cache managed by multiple replacement algorithms {OBL, Set Dueling – Thread Unaware (SD-TU), Set Dueling – Thread Aware (SD-TA), GPA – Miss-Rate Optimal, GPA – IPC Optimal W = {1, 2, 4, 8, 16}}. When using *weights* in a QoS context, it is helpful to think of the trade-off intuitively: A high-priority thread with weight 4, for instance, is worth 4 times as much of the performance metric (e.g., IPC) as the low-priority threads. That is, we must gain 4 times as much incremental throughput to justify moving a cache block from the high-priority thread to the low-priority threads. For the weighted GPA runs, only the high-priority thread, soplex.p, has a non-unity weight, the others have weights of 1. Notice that Figure 10 suggests that soplex.p would benefit considerably from much extra cache and generates comparatively high memory traffic. However, the absolute IPC of soplex.p is quite low, as shown in Figure 11. If it were given an 8MB cache it would only improve in throughput by 0.1 IPC,

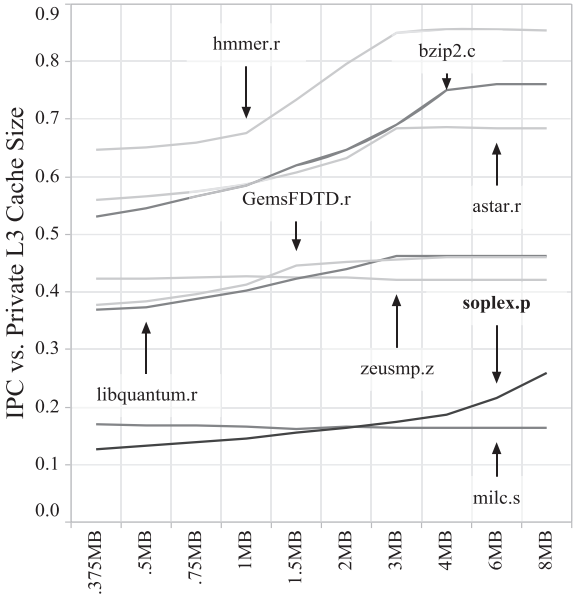

Fig. 11. IPC vs. Private L3 Cache Size.

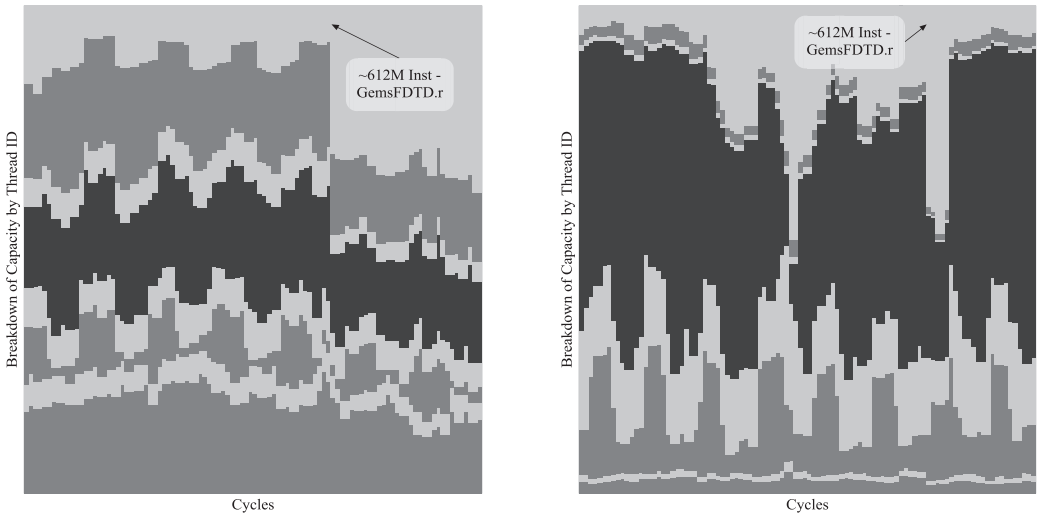

Fig. 12. Breakdown of Shared Cache Capacity (total 8MB) by Thread vs. Cycles (1 Billion) under OBL (left) and SD-TA (right) replacement. The 612 million instruction point for GemsFDTD is noted illustrating a prominent phase change.

whereas hmmer.r and bzip2.c would improve by improve by 0.2 in absolute IPC. Also worth noting is that astar.r, bzip2.c and hmmer.r (2nd, 3rd and 4th applications from the bottom) all generate comparatively little memory traffic given 2 or 3MB of cache yet have high IPC potential, a seemingly good return on investment for throughput performance.

It is also interesting to note that milc.s generates a large proportion of memory traffic and seems entirely unaffected by cache size, yet in Figure 12, milc.s appears to

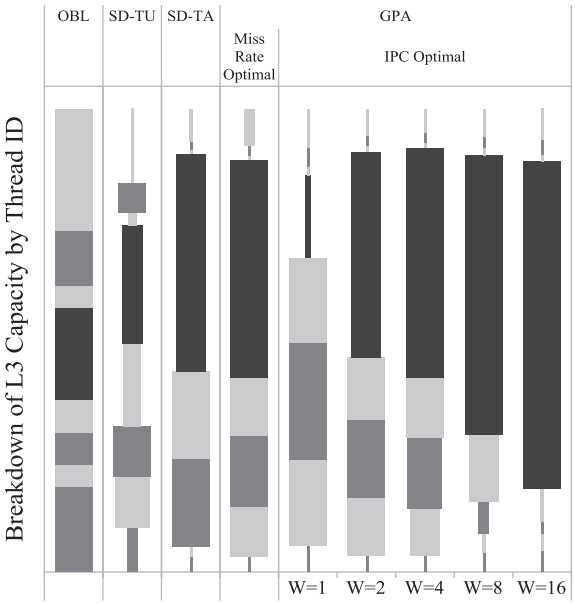

Fig. 13. Breakdown of Cache Capacity by Thread vs. various replacement algorithms in an 8MB Shared L3 cache. The segment width represents the fraction of installs that are normal, the rest are vulnerable.

be allocated a large cache partition under OBL replacement, supporting the observation that cache partitions under temporal replacement are proportional to the relative miss-rate. Sadly, the applications that would benefit the most from extra cache, hmmer.r, bzip2.c, and astar.r, are comparatively under-allocated. However, as designed, SD-TA manages to keep milc.s from obtaining a large partition and allocates more cache to hmmer.r and bzip2.c, though not astar.r. SD-TA allocates a very large partition to soplex.p, which is a consequence of the binary nature of SD-TA, if a thread is deemed to be useful, its relative miss-rate will govern its cache partition. That is, all threads deemed to be useful under SD-TA compete for cache space greedily, as in OBL. Also of interest, is the observation that at 612 million instructions GemsFDTD.r very aggressively strides through data and is thus allocated a large partition under OBL replacement. SD-TA appears to react very quickly to this phase change, restricting GemsFDTD.r to a small partition immediately after the phase change.

We can see in Figure 13 that GPA is able to favor soplex.p with increasing weights, though progressively at the expense of hmmer.h, bzip2.c and astar.r. Clearly, the overall performance is suffering, as we see in Figure 14, though the performance of soplex.p indeed improves. Also, we see that GPA-IPC Optimal is able to exploit the fact that hmmer.h, bzip2.c and astar.r all benefit more in throughput from additional cache than soplex.p, despite the fact that soplex.p indeed improves. As such, GPA-IPC Optimal gives it a small cache partition, though GPA-MissRate Optimal favors it more, due to its large reference rate. However, unlike SD-TA, GPA-MissRate Optimal is able to favor astar.r, which benefits greatly from cache (Figure 11).

In addition, we can see from the capacity graphs in Figure 15 that even with a weight of 2, the weighted version of GPA gives considerably more cache space to soplex.p, while still preserving reasonable capacity for hmmer.h, bzip2.c and astar.r and holding back the streaming access pattern of milc.s (the bottom trace). Notice that in both GPA

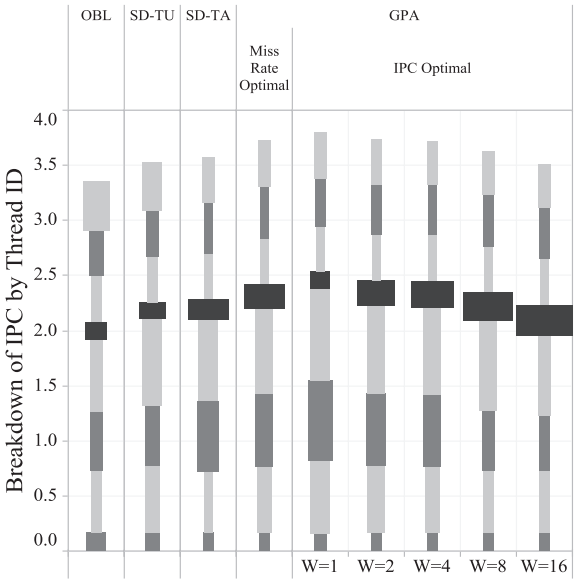

Fig. 14. Breakdown of IPC by Thread vs. various replacement algorithms in an 8MB Shared L3 cache. The segment widths represent the fraction of capacity.

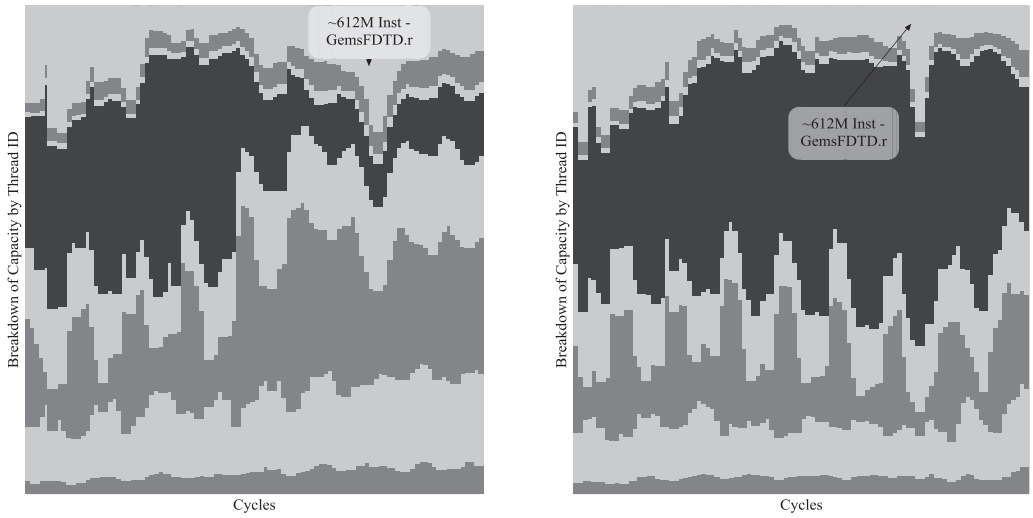

Fig. 15. Breakdown of Shared Cache Capacity (total 8MB) by Thread vs. Cycles (1 Billion) under GPA – IPC Optimal replacement with high-priority (soplex.p, in black) weights of 1 and 2, respectively.

capacity graphs (Figure 15), they are able to recover quickly from the phase change in GemsFDTD, whereas OBL allocates roughly a third of the cache to it.

Finally, we see in Figure 16 that GPA can span a continuum from optimizing total throughput to protecting any particular thread(s) via the QoS weights. In this, admittedly contrived, example we are able to get soplex.p within 10% of its performance given all the system resources and only take a slowdown of 3% off of the peak throughput

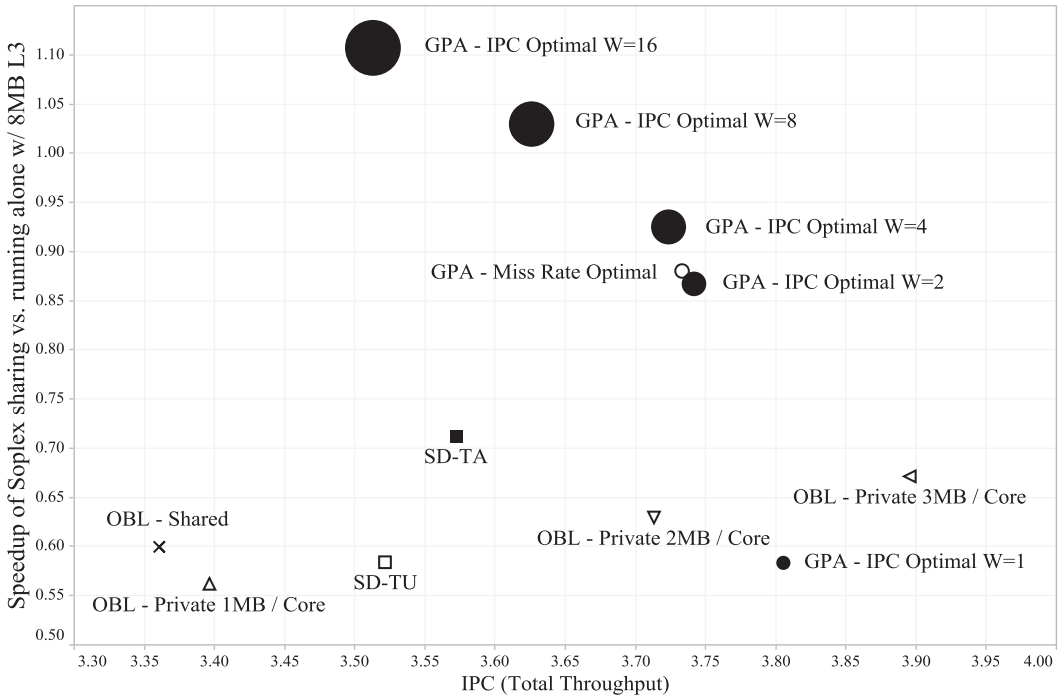

Fig. 16. QoS Summary depicting on the Y-axis the relative performance of soplex.p running in a shared environment vs. running alone with the entire 8MB L3 cache to itself. The X-axis is the total throughput of the whole system.

with GPA – IPC Optimal. This is a very reasonable choice, since running soplex.p in isolation yields only ∼0.26 IPC and we are able to perform an additional ∼3.5 IPC of background work while suffering only a minor slowdown for soplex.p. It is interesting to notice that with weights of 4 and 8, GPA – IPC Optimal improves the performance of soplex.p in a shared environment over the private environment, where it has the entire cache available to it. While this may seem counterintuitive, it is because we have a separate throttle for the main thread and the corresponding prefetcher. Stream-based hardware prefetchers commonly found in modern processors tend to be rather inaccurate for soplex.p. This leads GPA to reduce the throttle associated with soplex.p's prefetcher which results in less cache pollution and more performance. To reiterate, running soplex.p as a high-priority thread using GPA's QoS feature, allows the system administrator to get more performance for soplex.p running in isolation using OBL and get ∼3.25 IPC worth of throughput from 7 other background threads simultaneously.

5. CONCLUSION

GPA is a low-cost, scalable, robust and high performance mechanism for dynamically partitioning a shared cache. GPA – IPC Optimal improves Throughput of 4-core and 8-core CMPs by 8% and 14% respectively, which is roughly equivalent to doubling the L3 cache for 4 cores and quadrupling it for 8. GPA generally always improves performance, though suffers a 1% degradation in system throughput on two 4-core mixes as compared to a Private L3 cache with 1MB per core. The worst slowdown of any thread across all mixes is 13.9% and 16.3% for the 4-core and 8-core configurations,

respectively. However, the Average Harmonic Mean of Relative Speedups is 7% and 11% for 4-core and 8-core configurations, respectively, implying that it achieves a net improvement in total throughput without suffering unfair individual slowdowns as a result. Further, we can expose controls to the System Administrator which enable an artificial inflation of importance via the weights to the thread of their choosing. This tool can be used in conjunction with performance counters to implement an OS-level QoS system. Further, GPA could be used within a QoS-enabled architecture [Iyer et al. 2007; Rafique et al. 2006; Nesbit et al. 2007; Guo and Solihin 2007] to sensibly govern allocation in a shared pool of otherwise 'greedily' allocated cache. GPA is cheap ($\sim$5 Bytes of register state per thread), easy to build and has minimal impact to the critical path of the cache install path.

5.1. Future Work

We intend to pursue other uses of GPA: adapting parameters of prefetchers, adapting GPA parameters like *gradientDelta* and *updateDelta* and optimizing for power or performance per watt. We will apply GPA to RRIP replacement presented by Jaleel et al. [2010] in two ways. First, we will let the throttles control the fraction of installs at a value of 1 (the rest at 0). Second, we will let the throttle control the actual insertion value on a continuum – on a 3-bit counter, we continuously adjust the insertion value from 0 to 7. Additionally, we intend to apply GPA to upper levels of cache in SMT configurations and to comprehend memory bandwidth saturation effects. Future work will include some study of cooperative parallel applications and other commercial workloads. Finally, we intend to expose GPA registers to the operating system as a performance counter for code tuning and feedback directed compilation.

APPENDIXES

A. Constrained Optimization

In this work we attempt to start from first principles and cast cache partitioning as a constrained optimization problem. That is, given a finite amount of cache, C, how do we partition it to optimize some goodness metric? We will consider a few metrics, but let us begin with $m_i(c_i)$, the miss-rate (misses per cycle) of the ith thread, given c_i cache. Suppose we wished to find the c_i's that minimize the overall miss-rate of the cache, $M(C)$, subject to the constraint that the c_i's sum to C. We can cast this optimization in terms of the Method of Lagrange Multipliers.

$$\begin{aligned} M(C) &= \sum_{i=1}^{n} m_i(c_i) + \lambda \left(C - \sum_{i=1}^{n} c_i \right) \\ \frac{dM(C)}{dc_i} &= \frac{dm_i(c_i)}{dc_i} - \lambda = 0 \\ \frac{dm_i(c_i)}{dc_i} &= \frac{dm_j(c_j)}{dc_j} \quad \forall i, j \end{aligned}$$

This result implies that the conditions for optimality occur when the derivative of the miss-rate curves are equal. Of course, this is not a new result; it has been known for at least twenty years [Stone et al. 1992].

B. Gradient Descent and Convexity

Unfortunately, our ability to detect when the optimality condition holds does not necessarily imply that there is an efficient algorithm for finding it. However, it is well known that the Gradient Descent (Hill-Climbing) algorithm will converge to this optimality

condition. The astute reader may notice that a globally optimum partition implies that the optimality condition above holds, but the reverse is not necessarily true (i.e., a local optimum) unless all $m_i(c_i)$ are convex. Stone et al. [1992] assumed that all $m_i(c_i)$ are convex, and while it is not strictly true, for most applications in the presence of prefetching, it typically is.

C. Gradients in Memory Space

There is a potential pitfall to the time-multiplexed Gradient Descent strategy described above: programs have phases. It might be difficult to tell the difference between a performance benefit due to a change in cache policy or a cache-friendly phase of a program. Alternatively, we could divide some subset of the cache sets into two parts or *set maps*, S_0 and S_1, mapping each cache set to one or the other pseudo-randomly. This could be accomplished by implementing a one-to-one hash on the set index and using the least significant bit to select S_1 or S_0. Then, experiments can be carried out simultaneously, by allocating more cache to the ith thread in S_1 and less in S_0, measuring the system performance of each. This requires that the access patterns of two non-contiguous, randomly selected subsets of the memory space are equivalent. Compiler writers attempt to map data structures to physical pages such that the probably of accessing any particular set in the cache is equal. In fact, it is critical for good performance that references are uniformly distributed among sets in a set-associative cache; after all, the performance of a program that uses only one or a few sets of the cache would be too horrible to imagine. Further, our strategy is validated by the fact that the Dynamic Insertion Policy from Qureshi et al. [2007] relies even more heavily on this assumption, as it further sub-samples the memory space (i.e., it measures the performance of very few sets), yet it yields high performance.

D. Bimodal Insertion Policy

Thus far, our strategy requires that we enforce a particular cache partition in S_1 and S_0, skewed such that the ith thread (i.e., the thread that is currently being adjusted), has slightly more cache in S_1 than in S_0. Certain details complicate this process. For instance, suppose the appropriate partition has been achieved and we need to install a block for the ith thread, but there are no blocks from the ith thread in that particular set. How do we choose the victim and still maintain the appropriate partition?

We define a throttle per thread, T_i, equivalent to β of the Bimodal Fill Policy [Qureshi et al. 2007], which is the relative proportion of installs for the ith thread that are installed normally. The rest are installed in a vulnerable or deprecated manner. This deprecation can take many forms, but the minimum requirement is that the expected lifetime of a block installed vulnerably is shorter than the lifetime of a block that is installed normally. Ultimately, the larger the proportion of vulnerable insertions a thread has, the more other threads will evict them and the smaller the partition will become. Thus, we assert that the value of throttle, T_i, is monotonic in cache partition, c_i: $T_i < T_i' \rightarrow c_i \leq c_i'$. Exploiting this observation, we can apply Gradient Descent to the T_i's, by adding and subtracting an offset, delta, to the throttle in S_1 and S_0, respectively. This obviates the need to keep track of the partition of cache allocated to each thread. More importantly, it obviates the need to keep track of which thread each cache block belongs to, which requires additional state, and keeps the victim selection decision free from having to comprehend that data, which would likely lengthen the critical path of the victim selection circuit.

E. The Chernoff Bound

Up to this point, we have demonstrated how we can create a gradient in memory space using a throttle and the concept of vulnerable insertion. That is, we have a mechanism

to enforce that the ith thread has a larger partition of cache in S_1 than in S_0. Next, we face the problem of measuring the relative performance impact to S_1 and S_0, and in a timely fashion. Ideally, we would observe a string of references and when we see that either S_1 or S_0 clearly has the advantageous partition, move T_i in the direction of the gradient, increasing if S_1 is better, decreasing if not. This is very much like the classic example of the Chernoff Bound; how many times must one flip a biased coin before it can be declared with some probability of error the side that is biased? Chernoff tells us that n flips is sufficient: $n \geq (2p-1)^{-2} \ln\left(\varepsilon^{-2}\right)$, where p is the probability of heads (the bias) and ε is the probability of error. After n flips, one merely subtracts the number of tails from the number of heads and if the result is positive, the coin is biased to heads, tails otherwise. Similarly, we can accumulate the results of references to the cache; we increment a counter, A_i (i.e., *randomWalkCounter*, a state variable in GPA), for a hit to S_1 or a miss to S_0, decrement it for a hit to S_0 or a miss to S_1. The probability of an increment (i.e., heads) is $\frac{1}{2}\left(H\left(S_1\right) + \left(1 - H\left(S_0\right)\right)\right)$, where $H\left(S_1\right)$ is the probability that a reference to S_1 is a hit (likewise for S_0), and we assume that the references are evenly divided between S_1 and S_0. Before declaring that S_1 or S_0 is superior, again we must observe at least n references: $n \geq \left(H\left(S_1\right) - H\left(S_0\right)\right)^{-2} \ln\left(\varepsilon^{-2}\right)$. Further, after n references, the expected value of A_i is:

$$\begin{aligned} E[A_i] &= \left[\frac{1}{2}(H(S_1) + (1 - H(S_0))) - \left[1 - \frac{1}{2}(H(S_1) + (1 - H(S_0)))\right]\right] n \\ &= [H(S_1) - H(S_0)][(H(S_1) - H(S_0))^{-2} \ln(\varepsilon^{-2})] \\ &= (H(S_1) - H(S_0))^{-1} \ln(\varepsilon^{-2}) \end{aligned}$$

In addition, we will accumulate the total number of references (i.e., n) in a counter, R_i (i.e., *referenceCounter*, another state variable in GPA). So, right at the moment when the Chernoff Bound would suggest that we can confidently declare the direction of the bias, the condition: $A_i^2 \geq \ln(\varepsilon^{-2}) R_i$ holds. Because we do not know a priori what the bias is, we test for this condition after each reference and when it is met, we increment or decrement T_i, depending on the sign of A_i (as in Figure 2).

We call the amount by which we increment or decrement, the *updateDelta*, which is a free parameter to GPA. Incidentally, the weight in the pseudo-code segment (Figure 2) is used to optimize other metrics and will be discussed at the end of this section. There is a curious acceleration that occurs with this mechanism; when the bias is strong (i.e., not close to .5), $A_i^2 \geq \ln(\varepsilon^{-2}) R_i$ for small values of R_i, which allows us to adapt quickly. Conversely, when the bias is weak (i.e., close to .5), we will judiciously observe many more references before we update T_i.; This is illustrated in Figure 17.

F. Simultaneous Experimentation

The algorithm we have assembled so far allows us to find a value of T_i that is optimal given the current values of the other $T_j \forall j \neq i$. We could potentially use time-multiplexing to successively adjust each throttle in turn, though we run the risk of reacting slowly to a phase change (e.g., we are experimenting with thread 2, while thread 3 is going through a phase change, etc.). Alternatively, we can perform experiments on all threads simultaneously by assigning non-overlapping collections of sets or gradient regions to each thread. Nominally, one hashes the set index and then uses the result to assign the gradient region to each thread. For instance, with 16 threads, we use the lowest four bits of the set index hash to identify if a set belongs to the gradient region of a particular thread and a one-bit hash of the set index could determine the *set map*. In a distributed L3 cache, a hash on the address determines which L3 bank

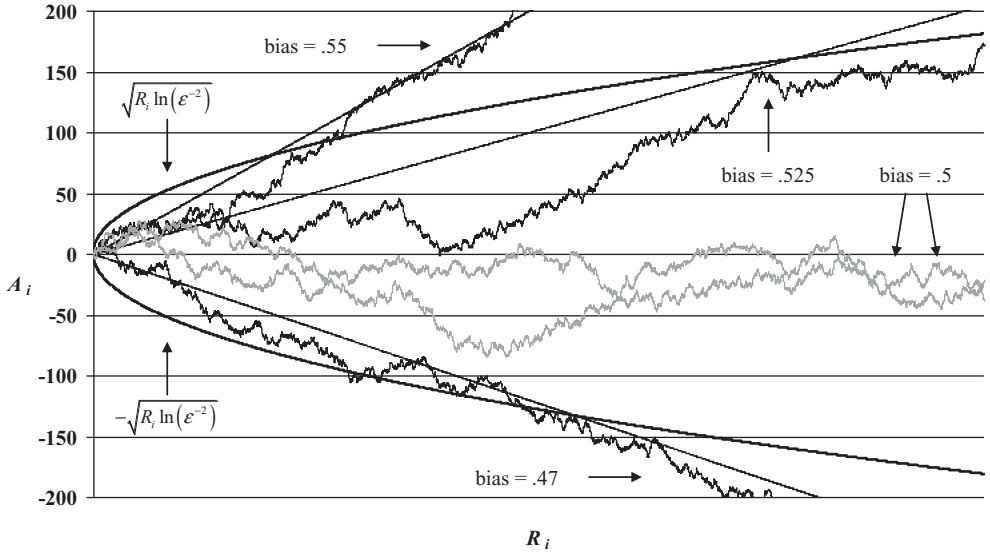

Fig. 17. Examples of random walks with various biases. The curved lines represent the Chernoff Bounds.

is relevant; this hash is appropriate for assigning gradient regions (i.e., one per bank), though a protocol for notifying other banks of throttle updates would be required in such a situation.

G. Weighted Throughput Optimization

Now we turn to our original promise to optimize throughput and yet cede QoS controls to the system administrator. We do this by optimizing weighted throughput, where the *weights* are configurable and represent the relative value of a particular thread compared with the others. For instance, if a thread has a weight of 4 (a high-priority thread) and the others have a *weight* of 1 (a background thread), GPA would need to observe 4 times as much incremental throughput from the background threads as compared to the high-priority thread to justify moving in that direction (i.e., taking cache capacity away from the high-priority thread and giving it to the background threads). In this way, we still manage to partition the cache sensibly, according to the needs and abilities of each thread, but with the flexibility to artificially modulate the value of any thread.

We start with an approximation to Throughput, T_i, or instructions per cycle (IPC), which is based on the misses per reference in the cache, $M_i(c_i)$, given c_i cache capacity. In addition, the IPC approximation relies on R_i, the fraction of instructions from the ith thread that are memory references, L_M and L_C, the latency to memory and the cache, respectively, D_i, all other sources of latency that do not depend on the partition of cache to the ith thread. Then, we use the Lagrange Multipliers approach to optimize T, the weighted throughput of the ensemble of threads, with the ith thread receiving a *weight* of W_i. Very fortunately, R_i combines with T_i to convert the derivative of misses per reference, $M_i(c_i)$, to the derivative of misses per cycle, $m_i(c_i)$, which we know how to optimize from Appendix A. Thus, we can use the previous mechanism from the MissRate Optimal formulation, with the enhancement that we weight the increments and decrements to the *randomWalkCounter* and the *referenceCounter* by W_i times the

IPC of the particular thread.

$$T_i = \left[R_i M_i\left(c_i\right)\left(L_M - L_C\right) + D_i\right]^{-1}$$

$$T = \sum_{i=1}^{n} W_i \left[R_i M_i\left(c_i\right)\left(L_M - L_C\right) + D_i\right]^{-1}$$

$$\frac{d\left(T + \lambda\left(C - \sum_{i=1}^{n} c_i\right)\right)}{dc_i} = W_i R_i \left(L_M - L_C\right) T_i^2 \frac{dM_i\left(c_i\right)}{dc_i} + \lambda$$

$$\rightarrow W_i T_i \frac{dm_i\left(c_i\right)}{dc_i} = W_j T_j \frac{dm_j\left(c_j\right)}{dc_j}$$

REFERENCES

ALBONESI, D. H. 1999. Selective cache ways: on-demand cache resource allocation. In *Proceedings of Micro, 32*.

AL-ZOUBI, H., MILENKOVIC, A., AND MILENKOVIC, M. 2004. Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite. In *Proceedings of ACMSE*.

CHANG, J. AND SOHI, G. S. 2006. Cooperative caching for chip multiprocessors. In *Proceedigns of ISCA 33*.

CHIOU, D. T. 1999. Extending the reach of microprocessors: column and curious caching. Ph.D. thesis, Massachusetts Institute of Technology.

DHODAPKAR, A. S. AND SMITH, J. E. 2002. Managing multi-configuration hardware via dynamic working set analysis. In *Proceedings of ISCA 29*.

DYBDAHL, H., STENSTROM, P., AND NATVIG, L. 2006. A cache-partitioning aware replacement policy for chip multiprocessors. In *Proceedings of HiPC 13*.

GUO, F. AND SOLIHIN, Y. 2006. An analytical model for cache replacement policy performance. In *Proceedings of SIGMETRICS/Performance*.

GUO, F., SOLIHIN, Y., ZHAO, L., AND IYER, R. 2007. A framework for providing quality of service in chip-multiprocessors. In *Proceedings of MICRO 40*.

GUO, F., KANNAN, H., ZHAO, L., ILLIKKAL, R., IYER, R., NEWELL, D., SOLIHIN, Y., AND KOZYRAKIS, C. 2007. From chaos to QoS: case studies in CMP resource management. *ACM SIGARCH Comput. Architec News 35*, 1.

IYER, R. 2004. CQoS: a framework for enabling QoS in shared caches of CMP platforms. In *Proceedings of ICS 18*.

IYER, R., ZHAO, L., GUO, F., SOLIHIN, Y., MARKINENI, S., NEWELL, D., ILLIKKAL, R., HSU, L., AND REINHARDT, S. 2007. QoS Policy and architecture for cache/memory in CMP platforms. In *Proceedings of ACM SIGMETRICS*.

JALEEL, A., HASENPLAUGH, W., QURESHI, M. K., SEBOT, J., STEELY JR., S., AND EMER, J. 2008. Adaptive insertion policies for managing shared caches on CMPs. In *Proceedings of PACT 17*.

JALEEL, A., THEOBALD, K. B., STEELY JR., S., AND EMER, J. 2010. High performance cache replacement using re-reference interval prediction (RRIP). In *Proceedings of ISCA 37*.

KIM, S., CHANDRA, D., AND SOLIHIN, Y. 2004. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of PACT 13*.

LUO, K., GUMMARAJU, J., AND FRANKLIN, M. 2001. Balancing throughput and fairness in SMT processors. In *Proceedings of ISPASS*.

NESBIT, K. J., LAUDON, J., AND SMITH, J. E. 2007. Virtual private caches. In *Proceedings of ISCA 34*.

PALACHARLA, S. AND KESSLER, R. E. 1994. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of ISCA 21*.

QURESHI, M. K. AND PATT, Y. 2006. Utility based cache partitioning: A low overhead high-performance runtime mechanism to partition shared caches. In *Proceedings of MICRO 39*.

QURESHI, M. K., LYNCH, D. N., MUTLU, O., AND PATT, Y. N. 2006. A case for MLP-aware cache replacement. In *Proceedings of ISCA 33*.

QURESHI, M. K., JALEEL, A., PATT, Y. N., STEELY JR., S. C., AND EMER, J. 2007. Adaptive insertion policies for high-performance caching. In *Proceedings of ISCA 34*.

RAFIQUE, N., LIM, W., AND THOTTETHODI, M. 2006. Architectural support for operating system-driven CMP cache management. In *Proceedings of PACT 15*.

STONE, H. S., TUREK, J., AND WOLF, J. L. 1992. Optimal partitioning of cache memory. *IEEE Trans. Comput. 4*, 9.

SUH, G. E., RUDOLPH, L., AND DEVADAS, S. 2004. Dynamic partitioning of shared cache memory. *J. Supercomput. 28*, 1.

Received July 2011; revised October 2011; accepted November 2011