Adaptive Caches: Effective Shaping of Cache Behavior to Workloads

Ranjith Subramanian Yannis Smaragdakis[†] Gabriel H. Loh

Georgia Institute of Technology
College of Computing
Atlanta, GA, USA
{ranji,loh}@cc.gatech.edu

[†]University of Oregon
Dept. of Computer and Information Science
Eugene, OR, USA
yannis@cs.uoregon.edu

Abstract

We present and evaluate the idea of adaptive processor cache management. Specifically, we describe a novel and general scheme by which we can combine any two cache management algorithms (e.g., LRU, LFU, FIFO, Random) and adaptively switch between them, closely tracking the locality characteristics of a given program. The scheme is inspired by recent work in virtual memory management at the operating system level, which has shown that it is possible to adapt over two replacement policies to provide an aggregate policy that always performs within a constant factor of the better component policy. A hardware implementation of adaptivity requires very simple logic but duplicate tag structures. To reduce the overhead, we use partial tags, which achieve good performance with a small hardware cost. In particular, adapting between LRU and LFU replacement policies on an 8-way 512KB L2 cache yields a 12.7% improvement in average CPI on applications that exhibit a non-negligible L2 miss ratio. Our approach increases total cache storage by 4.0%, but it still provides slightly better performance than a conventional 10-way set-associative 640KB cache which requires 25% more storage.

1. Introduction

The rapidly increasing gap between the relative speeds of processor and main memory has made the need for advanced caching mechanisms more intense than ever. Processor performance is now crucially determined by the amount of memory accesses served from on-chip caches, as the cost of access to RAM has grown to hundreds of cycles. In response to the need for better caching, computer architects have extensively explored the directions of expanding the cache size or hiding latency through sophisticated prefetching and out-of-order execution. Compared to such techniques, there has been relatively little recent work on improving cache replacement algorithms. Indeed, the implicit assumption seems to be that LRU is good enough, and that there is little one can do to increase cache hit rates for a given size without adding expensive processing to the cache reference handling critical path.

In this paper, we show that more sophisticated cache replacement algorithms exist and can substantially improve performance. Following up on our recent work in virtual memory management at the operating systems level [22], we present and evaluate the idea of *adaptive cache management* for use in microprocessor on-chip caches. Adaptive cache management consists of observing the behavior of two (or more) replacement algorithms, such as LRU and LFU, and then mimicking the behavior of the better performing policy. This work is based on a theoretical foundation that guarantees that our adaptive replacement algorithm never does worse than the *better* of the two component policies by more than a constant factor. In practice, our adaptive cache management accurately tracks the better of the two component policies, and in some cases adaptation can exploit differences in program phases to provide better performance than either policy in isolation.

Our adaptive cache management is a general idea that can be applied to any two cache replacement algorithms. The implementation logic turns out to be remarkably simple and the adaptivity actions are all performed off the critical path of memory reference handling. With an optimization that maintains only partial instead of full tags, the hardware overhead of adaptive caching can be reduced to 4.0% of the cache size for a 64-byte cache line size and 2.1% for a 128-byte cache line size, including the algorithm-specific per-entry overhead of the component replacement policies.

We evaluate the benefit of adaptive caching with a very extensive suite of programs (100 in total). Our goal is to show that over a wide range of program behaviors, our approach yields significant benefit, especially in cases of intense cache activity. Nevertheless, an equally important consideration is to show that adaptive caching never hurts performance by more than a negligible amount, even in the case of applications with less intense cache activity. We demonstrate that adaptive cache management can provide significant benefits in both miss rate reduction and overall performance improvement. Specifically, our results show that an LRU/LFU adaptive cache reduces the average L2 misses over all 100 benchmarks by about 19%. For the 26 programs with non-negligible L2 miss rates, the reduction in cache misses translates in a reduction of the average CPI by 12.9%. Adaptive caching never hurts performance by more than about 1% in the worst case for any of the 100 programs.

In the rest of the paper, we first describe the idea and principles of adaptive caching (Section 2). Then we discuss the hardware implementation (Section 3) and present the results of our evaluation (Section 4). A discussion of related work and our conclusions follow.

2. Adaptive Caching Principles

In this section, we discuss the general idea of adaptive replacement and how it can be applied to a processor's on-chip caches.

2.1. Adaptive Cache Design

The main reason for adaptive caching is that different workloads have better cache behavior under different replacement algorithms. Consider standard replacement policies such as LRU (Least Recently Used), MRU (Most Recently Used), FIFO (First-In, First-Out), or LFU (Least Frequently Used) used to manage each set of a set-associative cache. Traditional code that manipulates scattered data with good temporal locality performs almost optimally with LRU and fairly well with FIFO, yet causes LFU to underperform. In contrast, a linear loop slightly larger than the cache is bad for a set-associative, LRU-managed cache. In fact, the higher the associativity the more likely it becomes for LRU to produce bad behavior for linear loops. Finally, LFU is ideal for separating large regions of blocks that are only used once from commonly accessed data—a common pattern in media-management applications.

2.1.1. Hardware Structures

The goal of adaptive caching is to switch on-the-fly between two policies, in order to imitate the one that has been performing better on recent memory accesses. To do this, the adaptive cache needs to maintain some more information than a traditional cache. Specifically, our adaptive cache design has two extra elements (shown in Figure 1) compared to traditional caches.

The first additional structure is a set of parallel tag arrays that reflect the cache contents of each of the *component* policies A and B being adapted over. That is to say, each of these tag arrays tracks what would have been in the cache if only that one component policy was used. The parallel tag structures for the component policies have the same number of sets and set associativity as the regular tag array of the adaptive cache. Note that, even though the parallel tag structures register which blocks would be in the cache for each of the component policies, there is no record of the *data* contained in these blocks.

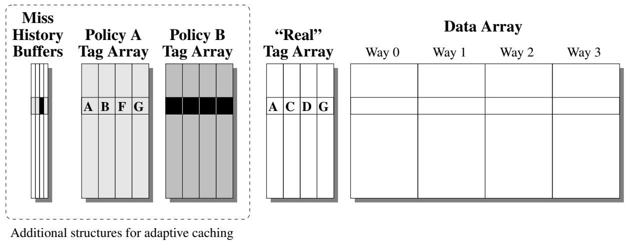


Figure 1. The basic hardware organization of an adaptive cache. The additional components required for the technique are grouped on the left.

The second additional structure is a per-set *miss history buffer*. For each cache set, the miss history buffer represents the past performance of the component replacement algorithms. Whenever a memory access misses in the real adaptive cache, the adaptive algorithm examines the miss history buffer for the appropriate cache set and chooses to imitate the component policy that has suffered fewer misses according to the miss history.

There are many possible implementations of a miss history buffer. The easiest to reason about (i.e., prove theorems) is one that keeps counters of all misses so far for both component policies. Maintaining a count of all misses since the beginning of time would require a large counter, which is neither realistic nor likely to adapt quickly to program behavior. Saturating counters could also be used as an approximation. In our implementation, we use a slightly different approach for quicker adaptation to recent program behavior. We keep a bit-vector of m bits, recording the latest m misses when only one of the two (but not both) component policies misses. (If both component policies would have missed, then there is no need to record this in the history.) For each such miss, the buffer's bit indicates whether it was a miss for the first or the second component policy. We set the parameter m to be equal to the cache associativity (e.g., 8 for a 8-way cache) or a small multiple of it.

The parallel tag arrays and the miss history buffers can cause the overall size of the cache to increase quite considerably. The rest of this section will continue to assume this full overhead, and we will then explain how to reduce this overhead to acceptable levels in Section 3.

2.1.2. The Replacement Algorithm

On every memory block reference, we update the parallel tag structures by emulating the behavior of component caches A and B for that reference. We also update the miss history buffer of the corresponding cache set. If the reference misses the cache, then we use the adaptive replacement logic detailed in Algorithm 1.

Note that on a cache lookup, the original tag and data arrays (in the right side of Figure 1) remain unmodified and therefore the adaptive policy has no impact on the overall cache access latency. The access of the parallel tag arrays, the miss history buffer, and updating these structures can occur in parallel with the original cache lookup and may even have a longer latency (see Section 3.3).

Algorithm 1 Algorithm for adaptive block replacement.

```
if ( misses($A$) > misses($B$) )  // if A missed more than
{                                 // B in history buffer
  // then imitate B
  if($B$ missed AND the block it evicts is in adapt. cache)
      adaptive cache evicts the same block $B$ evicts
  else
      adaptive cache evicts any block not in $B$
      // such a block is guaranteed to exist, since, in this
      // case, B's cache contents are different from those
      // in the adaptive cache and the two caches have
      // the same size.
}
else    // B missed more than A
  Same as above, but with $B$ replaced by $A$
```

2.1.3. An Example

The above definition of adaptivity is independent of the two component policies. Figure 2 illustrates the adaptive cache's behavior in a specific example. The figure shows a single set of the adaptive cache and its 4 entries. We represent the addresses of cache blocks with capital letters for ease of reference. In this example, we also assume for simplicity that the history buffer consists of two miss counters, storing the number of misses ever suffered by each of the component policies. The adaptive cache keeps track of the contents of both of its component policies. After the four initial references, all caches have the same contents and the miss counts of both component policies are equal to 4. The next reference, to block "D", causes evictions. The adaptive cache chooses to imitate policy A, since both component policies have the same number of misses. The subsequent reference to block "B", however, is a miss only for policy A, causing the adaptive cache to start imitating policy B.[1] For this to happen, the adaptive cache needs to find a block that policy B does not currently store—effectively trying to imitate policy B's cache contents. Thus, block "A" is evicted. Subsequently, the reference to block "C" causes an eviction only for policy A, confirming that the adaptive cache correctly picked in the previous step and reinforcing its choice of policy B. Finally, block "G" is referenced. Policy B still has the fewest misses and, furthermore, the contents of the adaptive cache are now identical to that of policy B's cache, therefore the adaptive cache replaces whichever block policy B replaces.

[1]To keep the example brief, we assume that the current miss is counted in the consideration of which component policy performs best. This does not have to be the case in an implementation.

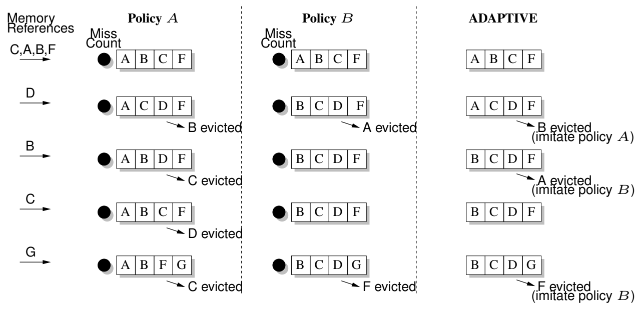


Figure 2. Example adaptive cache behavior.

2.2. Theoretical Guarantees

Interestingly, it is possible to prove that our adaptive algorithm fairly closely imitates the better of its two component algorithms. In the domain of virtual memory, our previously published adaptive algorithm [22] was proven to incur at most three times as many faults as the best of the component algorithms. This algorithm is quite similar to our cache adaptivity logic and all the proofs can be easily adapted to our set associative domain. The complete proof of the bound on the number of misses can be found in the Appendix. The main theoretical result bounds the number of misses of our adaptive policy to twice (or three times, depending on the exact version of adaptivity) the misses of the best of the component policies. In fact, our current theoretical result improves on our earlier ones. Whereas we previously only proved a bound of two for a simplified adaptive algorithm and a bound of three for the realistic algorithm, we now manage to prove a bound of two for a realistic version of adaptivity: one that keeps integer counts of past misses in order to pick the best component algorithm.

Of course, in practice we want our policy to match or outperform the best component policy, rather than just be within a factor of 2 of its misses. (This is indeed what our experiments later show.) Yet the 2x bound has three nice properties. First, it is good to have a worst-case guarantee, to bound how much the adaptivity can be "fooled." Second, the guarantee is often sufficient to ensure a performance benefit, since one component algorithm may be vastly better than the other for some program behaviors and vastly worse for others. Third, the bound is on the misses of the adaptive policy relative to the best component policy *for every set*. Thus, if the best component policy changes from one set of the cache to the other, the adaptive policy will outperform both component policies overall just by selecting the better one for every set.

3. Efficient Hardware Implementation

We next discuss hardware implementation considerations of adaptive caching.

3.1. Partial Tags

As described, our adaptive caching mechanism can potentially increase the implementation overhead of a cache by a significant amount. For example, an 8-way set associative 512KB L2 cache with 64-byte cache lines requires about 32KB of additional storage for tags and other meta-data for a total SRAM storage requirement of 544KB.[2] Implementing two additional tag arrays at 28KB each,[3] plus the miss history buffers at 1KB (8 bits per set), minus 3KB to avoid double-counting the LRU state (we do not need to replicate the LRU meta-data in both the main tag array and the component array) increases the total SRAM storage requirement to $\sim$598KB (+9.9%). An adaptive cache that employed all of this hardware would need to provide a significant performance gain to justify the substantial overhead. Furthermore, the adaptive cache must also provide more benefit than could be had with conventional approaches such as increasing the size and/or associativity of the cache. For example, a 9/10-way set associative cache provides 576KB/640KB of data at a cost of 612KB/680KB total storage for data and tags (+12.5/25.0%) which can reduce both capacity and conflict misses.

The key insight that enables a practical adaptive cache is the observation that a replacement policy is merely a heuristic used to (hopefully) improve performance, but the choice of policy has no bearing on the *correctness* of the processor. In the same way that branch predictors do not always provide a correct branch prediction, a given replacement policy will not always choose the optimal cache line to evict. This implies that any modification to the adaptive replacement scheme is acceptable in that the processor will still perform correctly, and that enables us to explore simplifications and approximations that yield better tradeoffs between performance and overhead.

In our design, we reduce this overhead by employing *partial tags* [12]. Traditional partial tagging serially compares disjoint portions of the tag for fast miss determination, but the entire tag is still maintained. Instead of keeping the entire tag in our parallel tag arrays, we can keep a subset of the tag bits [5], typically the low-order bits of the tag or a combination (e.g., XOR of bit groups). The size of this partial tag needs to be chosen such that the tag contains enough information to make conflicts/aliasing rare. Recall that the parallel tag structures are used to answer the question "what would each component cache contain at this point in the execution?" in order to enable the adaptive cache to emulate either component policy. If the answer is correct most of the time, minor inaccuracies should affect performance very little.

Using partial tags slightly changes the behavior of the overall adaptive replacement policy. Recall that in the definition of adaptive caching, part of the algorithm is stated as "evict a block that is not in cache A/B." (The contents of cache A and B are reflected in the parallel tag structures.) We previously asserted that such blocks are guaranteed to exist. Indeed, in the case of full tags, if no such block exists, the contents of the adaptive cache are identical to those of the corresponding component cache, and one of the other branches of the decision logic would be taken. In the case of partial tags, however, the check for tag equality between blocks in the adaptive cache and in the (partial) parallel tag structures may succeed even though the original block in the component cache is different. For example, the actual cache may contain block "X", and with full tags, the policy would choose to evict "X" because it is not present in a cache with policy A (suppose A contains the cache line "Y" instead). However, if the partial tag for "X" is equal to the partial tag for "Y", then the adaptive policy may not be able to find a "block that is not in cache A." This case is rare, but if it occurs the adaptive cache simply picks an arbitrary block to evict.

3.2. Storage Requirements

By employing partial tags, we can drastically reduce the overhead for the parallel tag arrays. For example, with 8-bit partial tags, the total storage requirements for the same 512KB cache discussed earlier is now reduced from 598KB down to only 566KB (+4.0% over a conventional 512K cache).[4] For a 128-byte cache line size [11], the overhead is reduced to only 2.1%. For the small amount of additional storage needed for our adaptive scheme, it is not even possible to add an extra way to the cache. Using these additional bits to add more sets would not be practical either, because that would make the total number of sets to not be a power-of-two.

Counting bits is only an approximation of the area and transistor overhead. Our parallel tag arrays can be implemented with somewhat less overhead due to reduced port requirements. The main tag array needs an extra port for snooping to maintain cache coherence in a multi-processor/multi-core environment. However, our parallel tag arrays can be implemented without support for snooping, which reduces the area, latency and power require-

[2] 8K cache lines, with about 32 bits each (24 for tags assuming a 40-bit physical address + 8 for LRU, valid, dirty and coherence bits, etc.) = 32KB.

[3] The additional tag arrays do not require as many meta-data bits (e.g., no valid, dirty, or coherence bits), which leaves 24 bits for tags + 4$\pm$ bits for policy-specific meta-data, e.g., LRU ordering or LFU counts.

[4] Each of the parallel tag arrays has been reduced from 28KB down to 12KB (1K sets $\times$ 8 ways $\times$ 12 bits for an 8-bit partial tag and four more bits for policy-specific meta-data).

ments of our technique. As a result, the parallel tag may report that a given cache line is present when it has been invalidated, but this only causes the replacement policy to deviate slightly from an implementation that accounted for coherence invalidations. This is likely to have very little impact on performance and provides a much greater benefit in reducing the hardware overhead of the extra tag arrays.

The small area overhead for the extra logic and state of our adaptive architecture is also a first-order approximation of its power overhead. Nevertheless, our approach can provide a substantial reduction in off-chip memory accesses that consume a substantial amount of power. Other mitigating factors include the fact that the adaptivity logic is only activated on accesses to the L2 cache, and that the timing of the adaptivity logic is off the critical path of the L2 cache and therefore can be implemented with lower-power and/or less leaky transistors. Thus, overall, we consider it unlikely that power will be a concern, or even a net overhead.

3.3. Timing

When accessing the adaptive cache, the processor accesses the primary components (right side of Figure 1) in the same fashion as a conventional non-adaptive cache. The adaptive cache must also access the parallel tag structures, but this can occur in parallel and off of the critical path of the main cache lookup. In the case of a cache hit, the adaptive cache returns the data with the same latency as the conventional cache, and the tag array (replacement policy) updates can complete without slowing down the corresponding memory instruction. The bookkeeping for the adaptive policy does not prevent the miss from initiating a fill request from the next level of the memory hierarchy either.[5]

The cache applies the adaptivity logic only in the case of a miss and entails choosing the component algorithm to imitate and searching in the set for the block to replace. Even though the decision and search will incur some overhead, the actions take place in parallel with fetching the missed data and should not be a bottleneck. The most involved part of adaptivity is the search for a block that is in the adaptive cache but not in one of the component caches. For reasonably low associativity caches (4-way or 8-way) this search can be implemented in parallel efficiently with only a small amount of hardware. In practice, this search can be made easier by taking advantage of the properties of the specific component policies: for instance, when adapting over LRU, the adaptive cache can keep a recency order and evict the least recent block when it wants to imitate LRU, instead of checking which block is not in the LRU tag structure.

[5]Depending on the implementation of the eviction/writeback buffers, an entry can be pre-reserved before the actual evictee has been determined to prevent deadlocking the buffers and queues of the hierarchy.

Instruction Cache	16KB, 64B line-size, 4-way LRU, 2 cycles
Data Cache	16KB , 64B line-size, 4-way LRU, 2 cycles
Branch Predictor	16KB gshare/16KB bimodal/16KB meta; 4K-entry, 4-way BTB
Decode/ Issue	8-wide; 32 RS entries, 64 ROB entries
Execution units	4 Integer ALUs, 4 Integer Mult/Div, 4 FP ALUs, 4 FP Mult/Div, 2 Memory ports
Execution unit latencies	IALU (1), IMULT/IDIV (8), FPADD (4), FPDIV (16)
Unified L2 Cache	512KB, 64B - line size, 8-way with adaptive LRU/LFU replacement, history buffer size m = 8, 5-bit LFU counters, 15 cycle hit latency, 4-entry store buffer
Memory	120 cycle latency
Bus	8B-wide split-transaction bus; processor to bus frequency ratio 8:1

Table 1. Simulated processor configuration.

4. Evaluation

We evaluated adaptive caching primarily using an LRU/LFU combination. This yielded significant performance improvement over a wide range of benchmarks.

4.1. Workloads and Experimental Settings

We use cycle-level simulation to evaluate the performance of adaptive caching. Specifically, we use the MASE simulator [13] from the SimpleScalar toolset [2] for the Alpha instruction set architecture. We made modifications to more accurately model the cost of memory stores, as the original simulator effectively assumed an infinite number of store buffers. Table 1 shows the configuration of the simulated processor—the settings are quite analogous to other recent caching studies [18, 19]. We later vary cache configurations to analyze the sensitivity of our technique to different processor configurations. Note that our L2 hit latency is pessimistic, which is conservative since it understates the potential performance speedup of our approach.

We simulated 100 applications[6] (our *extended set*) from many popular benchmark suites: SPECcpu2000 (INT and FP), MediaBench [14], MiBench [9], BioBench [1], pointer intensive applications [3], and graphics programs including 3D games and ray-tracing. All SPEC applications use the reference inputs and applications with multiple reference inputs are listed with different suffixes. We used the SimPoint 2.0 toolset to choose representative samples of 100 million instructions from these applications [17]. Using SimPoint avoids simulating non-representative portions of an application such as start-up and initialization code, which can sometimes take up the first several billion instructions of a program's execution.

Of these 100 programs, we focus our attention on those for which improvements to the L2 cache are likely to have a performance impact and use them as the *primary set* for our

[6]Some benchmarks have multiple inputs. The number 100 counts each benchmark$\times$input pair as a separate "application."

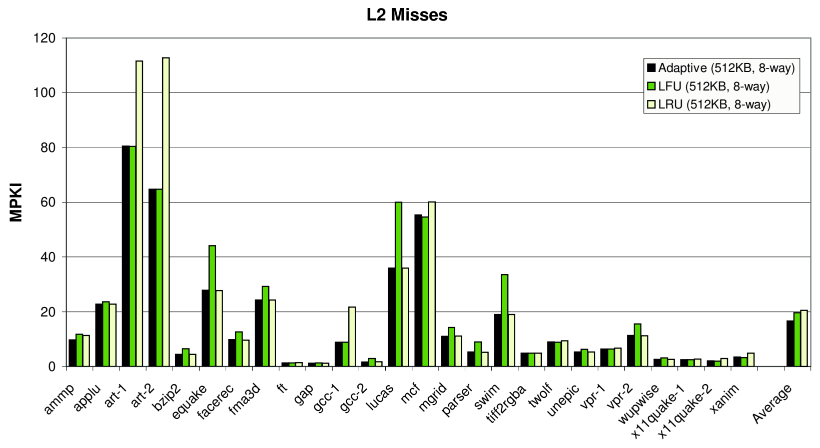


Figure 3. L2 Misses-per-thousand-instructions (MPKI) for each benchmark in our primary set for the adaptive policy and its component policies. (Lower is better.)

evaluation. The 26 selected programs are those whose execution suffers more than one miss per thousand instructions (MPKI) for a 512KB L2 cache managed with plain LRU. In the following, when we do not explicitly mention the set of tested applications, our primary set of 26 applications should be assumed.

4.2. Main Results

Adaptive caching has the potential to reduce cache misses and improve overall processor performance. Figure 3 shows the MPKI rates for each of the 26 applications and the overall average in our primary set.[7] Note that these results are for an implementation of adaptive caching with full tags—we later show that 6-or-more-bit partial tags achieve practically identical miss rates and performance.

We show the miss rates for our adaptive cache, and the LRU and LFU component policies. Of particular interest is the ability of the adaptive cache to accurately track the better performing component policy. For example, the lucas benchmark shows much better miss rates with an LRU policy, and the adaptive cache achieves a nearly identical rate. However for art, an LFU policy is superior and our adaptive scheme accurately adopts this behavior. Overall, adaptive caching reduces the average MPKI rate of our primary set by 19.0%.

In a modern out-of-order processor, the impact of miss rates on overall performance depends on many factors including the amount of independent work available to hide the memory latency (ILP), the degree of overlap of misses (MLP [8]), the criticality [7] or vitalness [20] of the cache miss, and many other factors. Figure 4 shows the CPI rates for each program in our primary set and the average CPI rates. Overall, adaptive caching reduces the average CPI of these applications by 12.9%. Ten of the executions benefit substantially from using adaptive caching: ammp, art (2 runs), gcc, mcf, mgrid, twolf, x11quake (2 runs) and xanim see a CPI improvement of 4% to 60%. The rest of the programs are never significantly harmed by employing adaptivity. The maximum CPI deterioration due to adaptive caching is 1.2% for unepic. This is exactly the desired behavior for an adaptive caching mechanism: the mechanism should be neutral relative to a reasonable baseline, unless it can derive benefit.

Over all 100 programs (primary set and otherwise), the reduction in average misses is 18.6%, and the improvement in average CPI is 8.4%. The reduction in benefit is primarily due to dilution from many of the traces whose working sets fit comfortably in a 512KB cache. A program that rarely misses in our 512KB L2 cache will not derive much benefit from adaptive caching. Yet, our 100-program extended evaluation set is important for demonstrating the stability of adaptive caching: adaptivity never increases misses by more than 2.7% (for the BioBench program tigr) and never hurts CPI by more than 1.2% (for unepic, as mentioned earlier).

4.3. Partial Tags

Partial tags are necessary to reduce the hardware cost of implementing adaptive caching. At the same time, partial tags introduce inaccuracies due to false positive matches. Excessive false positives could make the adaptive cache performance suffer by not accurately tracking the policy that truly has fewer misses.

We expect such errors to be quite rare with reasonable length partial tags. For our 8-way cache, we simulated partial tags of 4, 6, 8, 10, and 12 bits, using only the low-order

[7]Throughout the paper, we present “linear” metrics of performance cost, such as MPKI and CPI, so that they can be meaningfully averaged with a simple arithmetic average. For instance, our arithmetic mean of CPI rates is equivalent to the harmonic mean of IPC, and provides a metric proportional to overall execution time.

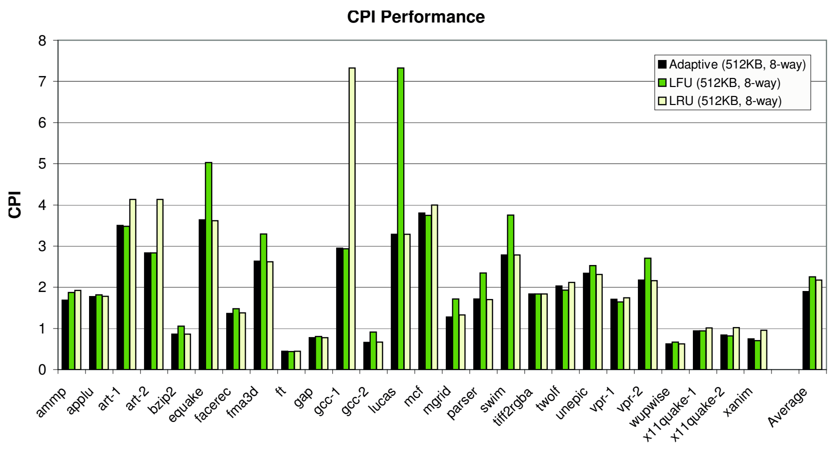


Figure 4. Cycles-per-instruction (CPI) rates for each benchmark in our primary set for the adaptive policy and its component policies. (Lower is better.)

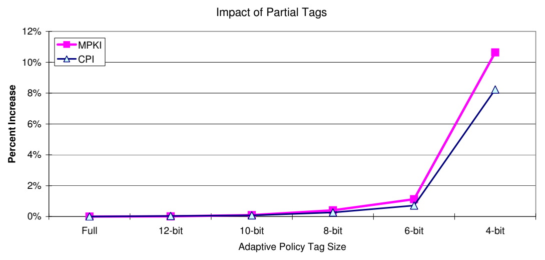


Figure 5. Effect of using partial tags on average CPI and MPKI rates.

bits of the tag (no XOR'ing of tag bits). Figure 5 compares the miss rate and CPI performance of full tags compared to partial tags for the 26 programs of our primary set. As Figure 5 shows, the difference is minimal: under 1% for partial tags of 6 bits or more. This difference does not affect in a significant way the benefit of adaptivity we reported earlier. With 8-bit partial tags we obtain about a 12.7% improvement in average CPI, compared to the 12.9% of full tags. Inspecting the individual programs' performance confirms that partial tags of 8 bits or higher do not distort the performance of adaptivity. 6-bit partial tags also achieve very good overall performance, however they introduce more per-benchmark variation. Whereas the maximum CPI deterioration for adaptive caching was around 1% with 8-bit or higher partial tags, with 6-bit partial tags this number climbs to 4% (for the lucas benchmark).

As discussed earlier in Section 3, using 8-bit partial tags results in an overall SRAM storage requirement of 566KB for our adaptive cache. An alternative to dedicating more resources to building an adaptive cache is to implement larger caches. Figure 6 shows the CPI performance for our adaptive cache with full tags, 8-bit partial tags, and conventional LRU caches with 8-, 9- and 10-way set associativities. Note that our adaptive cache is only 4.0% larger than the 512KB 8-way cache, whereas the 9-way and 10-way caches are 12.5% and 25% larger, respectively. We can see that for these applications, it is much more effective to use the existing resources more intelligently (i.e. use adaptation) than to blindly increase the size of the cache. With less than one sixth of the overhead (4.0% vs. 25.0%), our adaptive cache still performs slightly better (2.8%) than the 10-way 640KB cache.

4.4. Adaptivity Behavior and Comparison to Other Replacement Policies

We have already shown how our adaptive caching is able to accurately mimic the better of two replacement policies (LRU and LFU). However, on occasion, as for the ammp benchmark, the adaptive cache outperforms both of the component LRU and LFU policies, suggesting that each of them is more appropriate for different cache sets and/or phases of program execution. That is, the adaptive cache can track different behaviors temporally across different phases of a program as well as spatially across the different cache sets.

Indeed, ammp's behavior switches between LFU-friendly and LRU-friendly during the course of its execution. Figure 7(a) shows the behavior of all 1024 sets in our cache for ammp, sampled every one million cycles. A dark point in the figure indicates that the majority of replacement decisions during that time quantum were LRU, while a white point corresponds to LFU. As can be seen, there is a distinct phase at the beginning of execution where both LFU and LRU are the best replacement policies depending on the cache set, then from about 34M to 46M cycles, LFU is dominant, and then finally LRU takes over for the vast majority of the sets.

The pattern of switching behaviors during execution is

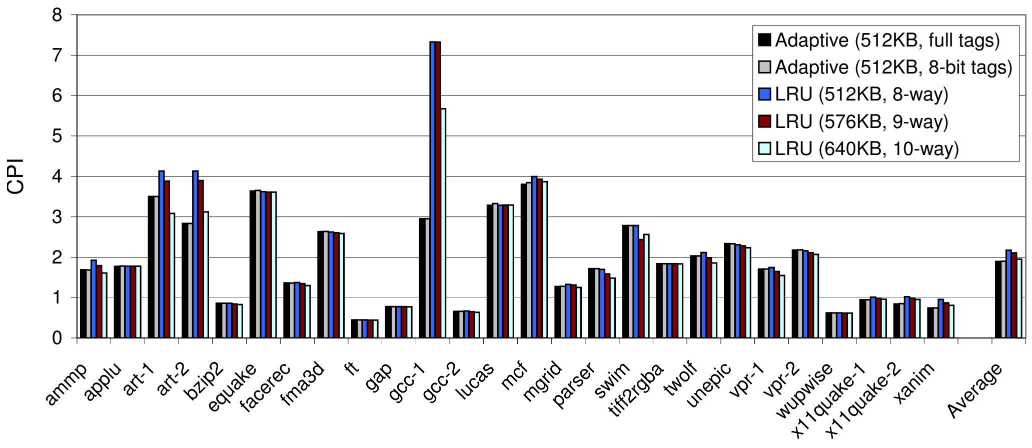
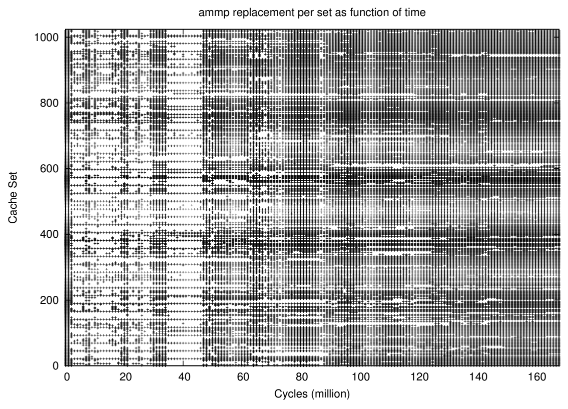
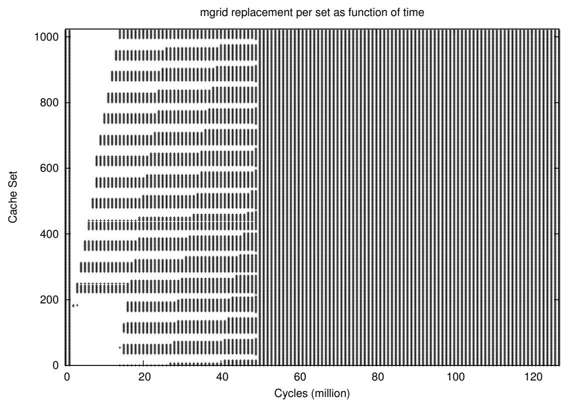


Figure 6. CPI comparison of partially-tagged adaptive replacement to increasing the size and set-associativity of a conventional cache.



(a)



(b)

Figure 7. Time- and space-varying behavior of (a) ammp and (b) mgrid: white dots correspond to LFU-favorable regions, black to LRU-favorable.

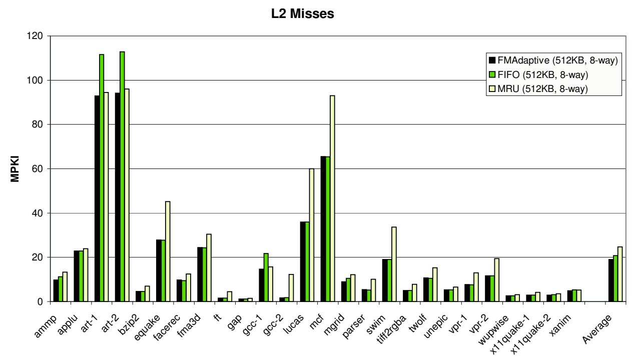


Figure 8. L2 Misses-per-thousand-instructions (MPKI) for a policy adapting between FIFO and MRU. (Lower is better.)

not rare among the tested applications. Similar to ammp, consider the mgrid benchmark whose behavior is shown in Figure 7(b). There is a clear pattern of behavior that favors LFU at first. The extent of the pattern gradually disappears and eventually the program's behavior is LRU-friendly. However, the rate of transition to LFU also varies spatially per set. The mgrid benchmark handles large 3-dimensional arrays, but the refernce patterns to the arrays vary depending on the exact subroutine. The subroutines such as `ZERO3` and `NORM2U3` traverse the arrays in a linear fashion, whereas the subroutine `RPRJ3` skips elements but references all of the neighboring array entries.

We have also experimented with combinations of a few other standard algorithms, such as FIFO, MRU, and Random. These expriments are important for demonstrating the generality of our adaptivity approach, as well as for exploring the design space. For instance, consider Figure 8, which shows the misses of an L2 cache adapting between FIFO and MRU. This is an interesting combination in that MRU

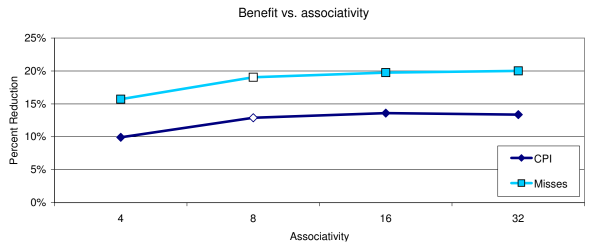


Figure 9. Overall benefit in terms of improvement of average CPI and reduction of average misses vs. associativity.

on its own is typically a very bad replacement algorithm. Yet for programs with large linear loops, MRU will outperform more reasonable policies such as LRU and FIFO. As can be seen in Figure 8, the adaptive policy tightly tracks the better of the two component algorithms. MRU is only beneficial for one of the gcc inputs, as well as for the art benchmark. For these, the adaptive policy tracks the MRU behavior.

Across the board, no combination of policies outperformed the LRU+LFU adaptivity. Furthermore, we experimented with a generalized version of the adaptive policy which combined five policies (LRU, LFU, FIFO, MRU, and Random). Although this is perhaps not a realistic configuration due to its high implementation overhead for five sets of extra parallel tag arrays (even with partial tags), it is interesting to see the achievable benefit in practice. The combination of all five policies was not clearly superior to just combining LRU and LFU, however. Although some benchmarks exhibited improvement, of up to 10% in CPI, others experienced an equal loss. The cumulative CPI over our primary evaluation set was virtually identical to that of LRU/LFU adaptivity.

4.5. Sensitivity Analysis

We next examine the sensitivity of our adaptive cache to various parameters and show that it is robust across a range of design points. In the rest of this section, our sensitivity results are for configurations using full tags. We do this to isolate the effects of the parameter under consideration from effects due to partial tagging. We have also repeated all of these experiments using 8-bit partial tags and have verified that the results still hold, but we do not include them here in the interest of space.

4.5.1. Associativity

In Figure 9 we show how the benefit metrics discussed earlier (improvement in average CPI and reduction of average misses) vary relative to associativity, over the 26 applications of our primary evaluation set. All configurations are for a 512KB cache, and so, for example, the 16-way cache (indicated by white points in the figure) has only half as many sets as the baseline 8-way cache. For highly set-associative caches (16- and 32-way), the relative benefit of the adaptive cache policy increases slightly, indicating that our technique would be effective for future highly-associative last-level caches.

4.5.2. Store Buffer Capacity

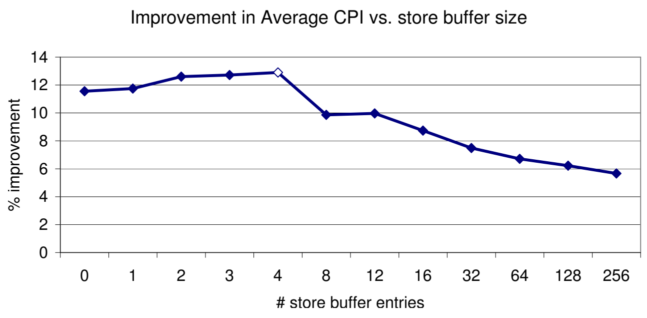
Out-of-order processors often use store buffers to queue up cache writeback requests after store instructions have retired from the main processor buffers (e.g., the store queue and ROB). The store buffer may also perform other functions such as write combining to further enhance performance. Figure 10 shows the effect of varying the number of entries in the store buffer on overall performance (average CPI improvement over the 26 benchmarks of our primary set). The benefit of adaptive caching is not only due to read misses but also due to store buffer stalls. As the number of store buffer entries increases, processor stalls due to store buffer contention decreases which reduces the overall number of opportunities for adaptive caching to provide a benefit. However, more than half of the benefit remains even for an unrealistically large 256-entry store buffer. The reduction in overall performance benefit degrades gracefully as the store buffer size increases (note that, after 16, the X axis becomes logarithmically spaced).

4.6. Adaptivity at Other Levels

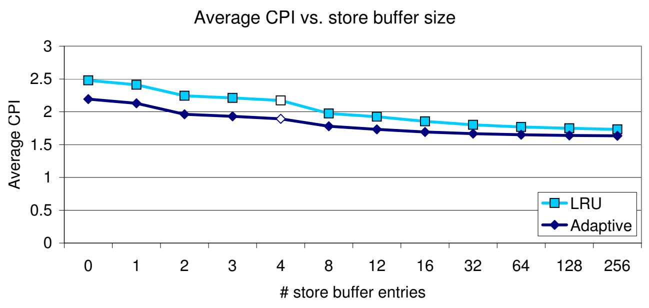
The idea of adaptive caching can be applied at other levels of the cache hierarchy. We simulated our standard configuration with LRU/LFU adaptive L1 instruction and data caches. In a 16KB instruction cache, the adaptive approach reduces the average MPKI rate by about 12%, whereas in the data cache the miss rate reduction was less than 1%. This did not result in any meaningful performance improvement ($<$0.1%) because our out-of-order microarchitecture can buffer enough instructions to tolerate the occasional instruction cache miss, and the L1 data cache is so dominated by capacity misses where there is not much opportunity for better replacement policies to help.

4.7. Eliminating the Overheads with Set Sampling (SBAR)

Although the hardware overheads of our adaptive cache are low (4.0% for the evaluated version), they can be further reduced by eliminating the overhead for all but a few sample sets in the cache. This is the recently proposed Sampling Based Adaptive Replacement (SBAR) technique of Qureshi, Lynch, Mutlu and Patt [18]. Under SBAR, a small number of "leader" sets are used to approximate how well a cache using an alternative policy would have performed. In fact, Qureshi et al. mentioned the potential of SBAR as a general technique capable of adapting between any two policies, and proposed the evaluation of the general case as



(a)



(b)

Figure 10. Effect of store buffer size on adaptive performance. Note that the x axis is irregular.

future work. Our evaluation shows that indeed SBAR is very promising as a general adaptivity technique.

We implemented an SBAR-like adaptive cache in our simulator, using the same component policies as described earlier, namely LRU and LFU. Policy-specific meta-data (e.g., frequency counts or recency information) are kept at all times for the blocks in the cache. For the leader sets (the few sample sets used to determine which policy performs best) the SBAR-like policy is very similar to our regular adaptive policy. Nevertheless, the SBAR-like cache does not have duplicate tag structures for the rest of the sets. Thus, when a decision is made to switch from, e.g., LRU to LFU, the cache does not have a record of what the contents of the LFU cache would have been at this point in the execution. Instead, the LFU algorithm begins executing on the blocks that are currently in the cache, and replaces the one with the lowest frequency. This means that the SBAR-like cache does not enjoy the theoretical guarantees of our adaptivity scheme, and may suffer a few more misses when switching policies often.

Yet in practice the SBAR-like cache's performance turns out to be quite competitive. For the programs in our primary set, the SBAR-like cache results in a 12.5% improvement in average CPI while our regular adaptive cache is only slightly better at 12.9%. As expected, the SBAR-like cache is a little less robust. We observed two benchmarks where the regular adaptive cache performed much better than the SBAR-like cache (9% on ammp and 40% on xanim) and the SBAR-like cache never performed much better (at most 4.4% on twolf). However, the SBAR-like cache's hardware overhead is just 0.16% compared to 11.8% for an adaptive cache with full tags, and 4.0% for an adaptive cache with 8-bit partial tags.

For even further benefits, the set sampling and partial tags techniques can be combined. When the leader sets of the SBAR-like policy make use of 8-bit partial tags, the performance is nearly identical to the original SBAR-like configuration and the overhead has been reduced to a miniscule 0.09%. Our evaluation over a larger set of benchmarks for both the basic adaptive cache and the SBAR-variant provides very strong evidence for the value of adaptive replacement policies for on-chip caching.

5. Related Work

There is clearly much work on improving cache performance for modern microarchitectures. We next selectively discuss only representative recent work or work particularly related to ours, and compare to our adaptive caching when possible.

The closest relatives of our work are our past research on adaptivity in virtual memory systems [22] and the recent work of Qureshi, Lynch, Mutlu and Patt [18].

Adaptive replacement algorithms in virtual memory systems [22] are algorithmically quite similar to our hardware scheme and offer the inspiration for stating our algorithm in its general form and proving theoretical bounds about it. The adaptive virtual memory management scheme operates by simulating two competing replacement policies, and then mimicking the best one. Just like in our approach, the OS needs to maintain three sets of "tags" (i.e., VM subsystem data structures with an entry for each page) and simulate the component policies in addition to managing real memory. However, the tag data structures are small (the meta-data for a 4K OS page is typically around a 40-byte linked-list node) and the extra processing time (perhaps a few microseconds) is a negligible price to pay for the corresponding reduction in disk accesses due to page misses (many milliseconds). Yet the adaptive replacement work in virtual memory did not examine the problem at all from the perspective of processor microarchitectures (e.g., set-associativity) and the physical constraints of a hardware implementation (e.g., timing, hardware cost of tag structures).

On the other hand, Qureshi et al. [18] recently proposed an adaptive caching scheme for processor caches. Their scheme is similar to ours, in that duplicate tag structures are maintained and adaptivity is affected by choosing the algorithm to imitate based on past statistics. Qureshi et al.'s work focused on adding awareness of memory-level parallelism to the cache subsystem. Their adaptivity considers switching between LRU and their LIN algorithm, which combines recency information with the expected cost to fetch a block. Qureshi et al. proposed the evaluation of their scheme for general adaptivity as future work. Indeed, our experiment of Section 4.7 shows that their set sampling (SBAR) approach is very powerful for general adaptivity.

Much of the recent work in caching deals with the elimination of context misses using techniques to increase the effective associativity, either uniformly for the entire cache, or on a dynamic, per-set basis. Good representatives of such work are Hallnor and Reinhardt's fully associative cache [10] and Qureshi et al.'s V-Way cache [19]. We next discuss the V-Way cache as an example, but our comments also hold for most other work in increasing associativity. The V-Way cache is a technique for dynamically varying the associativity of a cache on a per-set basis, in response to program behavior. Thus, the V-Way cache has some similarities to our approach, due to its adaptive and per-set character. Nevertheless, our adaptive caching scheme is strictly employed within the framework of a standard set-associative cache and incurs no overhead in the critical path of cache access. Our adaptive caching technique is sufficiently general in that it can simulate adapting between two different set associativities where policy A uses all n ways, and policy B effectively manages its cache lines as two separate sets of $n/2$ ways. Peir, Lee and Hsu also employ cache adaptivity for determining globally "least recently used" blocks for replacement [16]. Their work is closest to the V-Way cache, rather than to our adaptive replacement—the Peir, Lee and Hsu cache's adaptive behavior can be seen as a way to dynamically vary its associativity.

The above is also a more general observation: any advanced caching algorithm can be used as a component algorithm in an adaptive cache implementation. Examples include Seznec and Bodin's skewed associativity [4, 21], Hallnor and Reinhardt's fully associative cache [10], etc. Given the robust performance of an adaptive cache and its ability to faithfully imitate the better one of its component mechanisms, we believe that our work is orthogonal to other recent caching advances.

Other work uses adaptivity in processor caching, yet without applying it in relation to replacement algorithms. Speight et al. described "adaptive mechanisms" for managing cache hierarchies in multiprocessors [23]. This is a very different use of adaptivity, however. Their cache adapts its behavior (e.g., the way to perform write-backs) based on hints regarding the residence of blocks at different cache levels. Yet the replacement policy is never varied.

Although not entirely directly related to caching, there has also been much work in predictor design that attempts to make use of adaptivity between multiple policies (i.e. predictors). For example, there have been a variety of hybrid branch predictors that attempt to combine or adapt across multiple component algorithms [6, 15], and similar techniques have been attempted for value prediction [24], and load hit-miss prediction [25]. It may be possible to generalize the theoretical underpinnings of our work to prove worst-case bounds on these other types of adaptivity.

6. Conclusions

While there has been much work on improving processor caches, our work makes several new contributions:

- We introduce a general adaptivity scheme based on a strong theoretical foundation which provides a guaranteed worst-case bound on the number of misses relative to the component replacement policies.
- We propose a practical hardware embodiment of the adaptive replacement algorithm employing a partial-tagging scheme to reduce implementation overhead.
- We evaluate our approach thoroughly over a wide variety of benchmark programs, demonstrating the effectiveness and robustness of the technique across different behaviors as well as cache configurations.

There are several possible future directions for this work. We plan on evaluating adaptive caching policies for shared last-level caches in a multi-core environment. We believe that the combination of memory traffic from dissimilar threads or applications will provide even more opportunities for the adaptive mechanism to help performance. Our adaptation technique could possibly be modified to improve hybrid hardware prefetchers as well (hit/miss is replaced with useful/not-useful prefetch). As discussed earlier, the theoretical analysis of worst-case adaptation behavior can be extended to other hybrid/adaptive microarchitecture structures to better understand how and why these techniques improve processor performance. The strong theoretical guarantees of our technique increase the likelihood that it will be effective across a wide range of microarchitecture applications.

Acknowledgments

This research was sponsored in part by equipment and funding donations from Intel® Corporation. This research was performed while Yannis Smaragdakis was at the Georgia Institute of Technology and supported by NSF grants No. CCR-0220248 and CCR-0238289. We would like to thank Joshua Fryman (Intel) for his comments on earlier drafts of our paper. We are also grateful for the comments from the anonymous reviewers. Moinuddin Qureshi (UT-Austin) provided useful feedback and discussion about the SBAR technique.

References

[1] K. Albayraktaroglu, A. Jalell, X. Wu, M. Franklin, B. Jacob, C.-W. Tseng, and D. Yeung. BioBench: A Benchmark Suite of Bioinformatics Applications. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, pages 2–9, Austin, TX, USA, March 2005.

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

[3] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient Detection of All Pointer and Array Access Errors. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 290–301, Orlando, FL, USA, June 1994.

[4] F. Bodin and A. Seznec. Skewed Associativity Enhances Performance Predictability. In *Proceedings of the 22nd International Symposium on Computer Architecture*, Santa Margheruta Liguire, Italy, June 1995.

[5] A. N. Eden and T. N. Mudge. The YAGS Branch Prediction Scheme. In *Proceedings of the 31st International Symposium on Microarchitecture*, pages 69–77, Dallas, TX, USA, November 1998.

[6] M. Evers, P.-Y. Chang, and Y. N. Patt. Using Hybrid Branch Predictors to Improve Branch Prediction Accuracy in the Presence of Context Switches. In *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 3–11, Philadelphia, PA, USA, May 1996.

[7] B. Fields, S. Rubin, and R. Bodík. Focusing Processor Policies via Critical-Path Prediction. In *Proceedings of the 28th International Symposium on Computer Architecture*, pages 74–85, Göteborg, Sweden, June 2001.

[8] A. Glew. MLP Yes! ILP No! Memory Level Parallelism, or, Why I No Longer Worry About IPC. In *Proceedings of the ASPLOS Wild and Crazy Ideas Session*, San Jose, CA, USA, October 1997.

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

[10] E. G. Hallnor and S. K. Reinhardt. A Fully Associative Software-Managed Cache Design. In *Proceedings of the 27th International Symposium on Computer Architecture*, Vancouver, Canada, June 2000.

[11] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyler, and P. Roussel. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*, Q1 2001.

[12] R. E. Kessler, R. Jooss, A. Lebeck, and M. D. Hill. Inexpensive Implementations of Set-Associativity. In *Proceedings of the 16th International Symposium on Computer Architecture*, pages 131–139, Jerusalem, Israel, May 1989.

[13] E. Larson, S. Chatterjee, and T. Austin. MASE: A Novel Infrastructure for Detailed Microarchitectural Modeling. In *Proceedings of the 2001 International Symposium on Performance Analysis of Systems and Software*, pages 1–9, Tucson, AZ, USA, November 2001.

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

[15] S. McFarling. Combining Branch Predictors. TN 36, Compaq Computer Corporation Western Research Laboratory, June 1993.

[16] J.-K. Peir, Y. Lee, and W. W. Hsu. Capturing Dynamic Memory Reference Behavior with Adaptive Cache Technology. In *Proceedings of the 8th Symposium on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, USA, October 1998.

[17] E. Perelman, G. Hamerly, and B. Calder. Picking Statistically Valid and Early Simulation Points. In *Proceedings of the 2003 International Conference on Parallel Architectures and Compilation Techniques*, pages 244–255, New Orleans, LA, USA, September 2004.

[18] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A Case for MLP-Aware Cache Replacement. In *Proceedings of the 33rd International Symposium on Computer Architecture*, Boston, MA, June 2006.

[19] M. K. Qureshi, D. Thompson, and Y. N. Patt. The V-Way Cache: Demand-Based Associativity via Global Replacement. In *Proceedings of the 32nd International Symposium on Computer Architecture*, Madison, Wisconsin, June 2005.

[20] R. Rakvic, B. Black, D. Limaye, and J. P. Shen. Non-Vital Loads. In *Proceedings of the 8th International Symposium on High Performance Computer Architecture*, pages 165–174, May 2002.

[21] A. Seznec. A Case for Two-way Skewed-Associative Caches. In *Proceedings of the 20th International Symposium on Computer Architecture*, San Diego, CA, USA, May 1993.

[22] Y. Smaragdakis. General Adaptive Replacement Policies. In *International Symposium on Memory Management*, pages 108–119, Oct. 2004.

[23] E. Speight, H. Shafi, L. Zhang, and R. Rajamony. Adaptive Mechanisms and Policies for Managing Cache Hierarchies in Chip Multiprocessors. In *Proceedings of the 32nd International Symposium on Computer Architecture*, Madison, Wisconsin, June 2005.

[24] K. Wang and M. Franklin. Highly Accurate Data Value Prediction Using Hybrid Predictors. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 281–290, 1997.

[25] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan. Speculation Techniques for Improving Load Related Instruction Scheduling. In *Proceedings of the 26th International Symposium on Computer Architecture*, pages 42–53, Atlanta, GA, USA, June 1999.

Appendix

Below we prove that the adaptive caching mechanism, as defined in Section 2, never incurs more than a small factor (plus a constant) more misses than either of its component algorithms. We examine two versions of adaptivity: one where the miss history buffer records the counts of all misses since the beginning of execution for the component policies, and one that models our exact implementation, which maintains an m-bit vector of recent misses. In the former case we prove that the adaptive caching technique is always within a factor of 2, plus a constant, of the best component policy. In the latter case, we prove a factor of 3. Our approach is similar to that of our earlier work [22], but our first version of adaptivity improves on the earlier result by showing a two-fold bound for a non-contrived algorithm.

Consider first an adaptive cache that keeps two counters C_A and C_B of all misses suffered by component policies A and B. Then, at every miss, the adaptive policy will imitate the component policy with the lowest miss count. We will prove that such an adaptive policy will never suffer more than twice as many misses as either of its component policies. (The counters can be periodically reset, but our theorem will hold, adjusted by a constant, between successive resets.)

First we show an intermediate lemma.

Lemma 1 *If a tag appears in the parallel tag structures for both component algorithms A and B then it is also in the adaptive cache.*

Proof: The above property holds initially (empty caches) and if it holds up to a point, then consider the next miss for either algorithm A or B. If the access is not a miss for the adaptive cache, then the property will hold after the eviction (because the new block is already in the adaptive cache). If the access is also a miss for the adaptive cache, then there are 4 cases in the adaptive cache's eviction logic:

- the adaptive cache evicts a block not in B's tag structure.
- the adaptive cache evicts a block not in A's tag structure.
- the adaptive cache evicts evicts the same block as A
- the adaptive cache evicts evicts the same block as B

In each of the above cases, the block evicted by the adaptive cache either could not have been in both A's and B's tag structures before the eviction or will be evicted from one of them at the same time. Thus, if the property held before the current miss, it will hold after the eviction.

Now we can show our theorem.

Theorem 1 *The adaptive cache will never suffer more than $2x + w$ misses, where x is the minimum of the numbers of misses of component algorithms A and B, and w is the size of the cache in blocks.*

Proof:

We will treat each cache set independently and will prove that the total misses suffered by the adaptive cache for a single set alone are at most twice the misses of algorithms A or B for that same set plus a constant equal to the cache's associativity, a. If we then sum over all sets, we get the desired theorem.

We define "potential" quantities and examine how their values change on every miss for either algorithm A or B.

Let d_A be the number of blocks currently in the adaptive cache that are not in A's tag structure. Let d_B be the number of blocks currently in the adaptive cache that are not in B's tag structure.

The values of C_A, C_B, d_A, and d_B change as follows on every miss (we denote the new values C'_A, C'_B, d'_A, and d'_B):

(Note that according to Lemma 1 a hit for both A and B implies a hit for the adaptive cache and none of the potentials changes.)

1. if $C_A > C_B$ (the adaptive cache imitates policy A)
   (a) miss for B, hit for A, hit for adaptive: $C'_A = C_A$, $C'_B = C_B + 1$, $d'_A = d_A$, $d'_B \le d_B$
   (b) miss for B, hit for A, miss for adaptive: $C'_A = C_A$, $C'_B = C_B + 1$, $d'_A \le d_A$, $d'_B \le d_B$
   (c) miss for B, miss for A, miss for adaptive: $C'_A = C_A + 1$, $C'_B = C_B + 1$, $d'_A \le d_A + 1$, $d'_B \le d_B$
   (d) miss for B, miss for A, hit for adaptive: $C'_A = C_A + 1$, $C'_B = C_B + 1$, $d'_A \le d_A$, $d'_B \le d_B$
   (e) hit for B, miss for A, hit for adaptive: $C'_A = C_A + 1$, $C'_B = C_B$, $d'_A \le d_A$, $d'_B = d_B$
   (f) hit for B, miss for A, miss for adaptive: $C'_A = C_A + 1$, $C'_B = C_B$, $d'_A \le d_A + 1$, $d'_B < d_B$
2. if $C_A \le C_B$
   (a) miss for B, hit for A, hit for adaptive: $C'_A = C_A$, $C'_B = C_B + 1$, $d'_A = d_A$, $d'_B \le d_B$
   (b) miss for B, hit for A, miss for adaptive: $C'_A = C_A$, $C'_B = C_B + 1$, $d'_A < d_A$, $d'_B \le d_B + 1$
   (c) miss for B, miss for A, miss for adaptive: $C'_A = C_A + 1$, $C'_B = C_B + 1$, $d'_A \le d_A$, $d'_B \le d_B + 1$
   (d) miss for B, miss for A, hit for adaptive: $C'_A = C_A + 1$, $C'_B = C_B + 1$, $d'_A \le d_A$, $d'_B \le d_B$
   (e) hit for B, miss for A, hit for adaptive: $C'_A = C_A + 1$, $C'_B = C_B$, $d'_A \le d_A$, $d'_B = d_B$
   (f) hit for B, miss for A, miss for adaptive: $C'_A = C_A + 1$, $C'_B = C_B$, $d'_A \le d_A$, $d'_B \le d_B$

We do not show all the case-by-case reasoning needed to derive the above values. As a single example, consider, for instance, case 1.e. In this case, there is a miss for A but none for B, which is reflected in the update of miss counts. Since the reference is a hit for the adaptive cache, one extra common block will exist in both tag structures (for A and for adaptive) after this reference is processed. At the same time, however, A can evict a block that is in the adaptive cache, so d_A (the number of blocks in the adaptive cache's tag structure that are not in A's tag structure) may decrease or stay the same (hence the $d'_A \le d_A$). d_B stays the same since the reference was a hit both for the adaptive cache and for algorithm B.

Now we are ready to show our result. Assume, without loss of generality, that the algorithm with the fewest total misses is B. (The case where A is the better algorithm for the current execution is handled practically identically.) Consider the point in the execution when C_A was last equal to C_B—we will call this the "turning point". This was the last time the adaptive cache ever emulated policy A. (This point, of course, could be the very beginning of the execution.) The main theorem will be broken up to two parts: first we show that the adaptive policy cannot have suffered more than twice as many misses as B until the turning point, and then we show that the adaptive policy cannot have suffered more than a more misses than B after the turning point. The two bounds have a total of $2x + a$.

It is easy to see that the adaptive cache never suffered more than twice as many misses as policy B until the turning point: The adaptive cache suffers a miss that is not a miss for B only in cases 1.f and 2.f, above. But each of these cases increases the metric $C_A - C_B$. Since $C_A - C_B$ is zero initially and also zero at the turning point (by definition), the number of times cases 1.f and 2.f could have occurred is at most as many as the number of times the difference $C_A - C_B$ has decreased. But this only occurs in cases 1.a, 1.b, 2.a, and 2.b, and in all of those policy B suffers a miss. In other words, the number of misses for the adaptive cache that are not misses for policy B are at most as many as the misses of B. That is, the adaptive cache can only suffer up to twice the misses of cache B up until the turning point.

At the turning point (and, in fact, at any point) the value of quantity d_B is at most equal to the associativity, a—since d_B reflects how many blocks in the set are different in the adaptive cache and in cache B. After the turning point, the adaptive cache imitates policy B. Thus, the only case where the adaptive cache suffers a miss that is not a miss for B is case 1.f. But in this case, d_B gets decremented and it can never fall below zero. Thus, the adaptive cache suffers at most d_B misses over those of B after the turning point, which is at most equal to a.

To summarize, for any given set, the number of misses of the adaptive cache are at most $2x + a$, where x is the number of misses of the better of the two policies for that set. If we sum over all sets in the cache, we see that the adaptive policy never suffers more than twice plus w the misses of the better of the two component policies. $\square$

Consider now the version of adaptivity where the miss history buffer consists of m bits recording the last m misses for either policy A or policy B but not for both. We have not been able to prove a 2x bound in this case, but proving a 3x bound is simple by appealing directly to a theorem in our earlier work for fully-associative memory [22]. If we treat each set as a separate fully associative memory, our m-bit miss history buffer is identical to the data structure kept in that work, for which we proved a 3x bound of the adaptive policy [22].