Enhancing Data Cache Performance via Dynamic Allocation

George Murillo, Scott Noel, Joshua Robinson, Paul Willmann

Rice University
6100 Main Street
Houston, TX 77005, USA
{jmurillo, scnoel, jpr, willmann}@rice.edu

Abstract

As process technologies get smaller, the mounting problem of wire delay is becoming a pervasive challenge in the microprocessor architecture community. In order to not sacrifice the seemingly required benefits of a larger, more associative level 1 (L1) data cache, architects have opted for an L1 data cache that has a multiple cycle latency. Largely overlooked, however, is the data reference patterns of modern applications that exhibit predictive, read-and-throw-away behavior and do not benefit from traditional large, associative caches. We examine a new cache hierarchy that achieves the equivalent or better levels of data locality using lower-delay elements that will scale better with new process technologies. Furthermore, we show that our proposed architecture achieves up to 16.7 % better performance than existing architectures with structures of larger size when real-world latency is applied, and it provides comparable miss rates with respect to much larger, more complex and therefore slower architectures.

1 Introduction

Over the past few years the use of general purpose processors to run computationally intense multimedia applications has increased. In addition to multimedia-only applications, general purpose applications are starting to include multimedia features. This led us to analyze multimedia processors and digital signal processors (DSPs) in order to identify features that could be applied to general purpose processors and improve performance. The inherent nature of media-only applications, however, is strikingly unique compared to general purpose business and scientific programs. With respect to the memory subsystem, media-only applications usually do not make good use of traditional caches due to the limited amount of temporal locality and general reuse that can be extracted from them. This is primarily a result of repetitive read-and-throw-away algorithms in which large amounts of data are processed once and rarely, if ever, revisited. Such data rarely benefits from a large, flexible cache. Media processors and DSPs typically have small simple caches (if they have caches at all) that are often software-managed [1] in order to maximize their efficiency by controlling what is actually cached, and to provide deterministic behavior.

1.1 Hypothesis

In this paper we propose a new cache management policy based on the algorithms and access patterns found in many media processing streams. We propose a special cache sidebuffer where we will place data that we anticipate to have little temporal locality; this data will not be allocated in the main L1 data cache. The architecture will perform dynamic runtime analysis of loads to determine whether a load should be allocated in L1 or instead placed in storage outside L1 (the sidebuffer.) The implementation and careful management of the sidebuffer and L1 data cache will reduce overall (cache and sidebuffer combined) L1 cache miss rates by removing repeating, read-only accesses that would otherwise pollute the L1 cache through by evicting needed data. We believe that our configuration will perform similarly to unmodified configurations that have larger caches, but when compared to configurations equal-sized caches ours will perform better.

In Section 2, we review the background and motivation for our work. In Section 3, we present the Dynamic Cache Allocation architecture. In Section 4, we review our experimental implementation and evaluation process. In Section 5, we present our results and analysis. In Section 6, we discuss related work, and in Section 7 we present our concluding remarks.

2 Background

Before proposing a solution to the hypothetical problem of inefficient cache management given the changing behavior of general purpose applications, we first needed to establish the validity of the problem as posed. If valid, we needed to derive a solution that could not only outperform existing architectures but would scale in the environment of increasing relative wire delay. It would thus serve as a performance boost and a means to deal with the growing scaling problems computer architects currently face. We present the supposition of the problems faced and our evaluation of their scientific basis here.

2.1 Motivation

We believe that the use of general purpose processors to process media applications and media-like workloads will increase during the following years. One of the current major problems that general purpose processors suffer is data cache efficiency as expressed as a miss rate; architects have tried to reduce cache misses by increasing the size of cache and its associativity. However, some workloads (such as media workloads) make very poor use of caches and thus may pollute the cache by introducing large quantities of 'useless' data (lacking significant temporal locality) into the cache, evicting more useful data. Data that is not reused does not benefit from being in the cache, and moreover it is harmful to the rest of the cached data to do so. However, we foresee that future microprocessors will in fact have smaller, less associative caches because large associative caches are slower in absolute terms. Relative to processor clock speed (which continues to increase), wire delay is increasing and thus even *existing* caches are getting slower from one generation of processor to the next. This is anecdotally exhibited in the transition from IBM's single-cycle POWER3-II L1 data cache [11] to the four-cycle store, two-cycle load latency in POWER4 [4]. Thus, increasing sizes and associativity is an infeasible option. Rather, we believe that researching and improving cache management policies will yield designs that are more future-ready.

Analyzing and leveraging the behavior of media applications could increase performance in general purpose processors both in media applications as well as regular applications. Data that is allocated in the cache that will only be read and thrown away can cause cache pollution, because (if repeated across a wide range of data addresses) it may evict data from the cache that could be reused many times in the future. Since a large portion of the data in some media applications tends to fit this classification, data is constantly ejected from general-purpose caches and miss rates increase. This problem will be exacerbated as more workloads with similar behavior are run on a general purpose processor.

2.2 Initial Verification Testing

In order to first determine that modern programs actually do exhibit the type of cache pollution that we are suggesting in this paper we ran several memory traces on benchmark programs. We wanted to determine what percentage of program memory accesses, especially programs in the SPEC benchmark suite and media programs, are a read once type of access with no write back. Our tests determined if a memory access was read once by noting if there was more than one load to the same memory address in both the forward and backwards direction in the trace without a store to that address in between the loads

Figure 1 shows the percentage of memory references expressed as a percentage of the total number of memory references including loads and stores that we would like to target for special allocation. A reference qualifies if, during a given examination window, there are at least two other loads from that cache line, and during that window there are no writes to that line. Our results show that these types of accesses range anywhere from 8 % for the equake benchmark to 50 % for the mcf benchmark assuming a window size of 8 memory accesses in each direction from the memory access that we were analyzing. Note that on average 24 % of the memory accesses over all of the memory traces that we ran exhibited the read once behavior that we were looking for. We felt that this was a high enough percentage to warrant targeting for allocation in our sidebuffer rather than the L1 cache.

We not only wanted to test if we had a large number of memory accesses of this qualifying type, but we also wanted to make sure that these type of accesses were well dispersed throughout the cache. Wide distribution would suggest cache pollution is more likely due to more conflicting addresses and thus our sidebuffer would be more effective in increasing performance (assuming there are memory references available to take advantage of the less-polluted cache.) From a qualitative look at plots (See Figure 2 for an example) of the number of qualifying accesses vs. cache set addresses, it appeared very promising. Figure 2 demonstrates that if we were to target these qualifying accesses and not allocate them in the cache, we would be affecting the full spectrum of cache locations as opposed to only affecting a small corner of the cache. As there are more

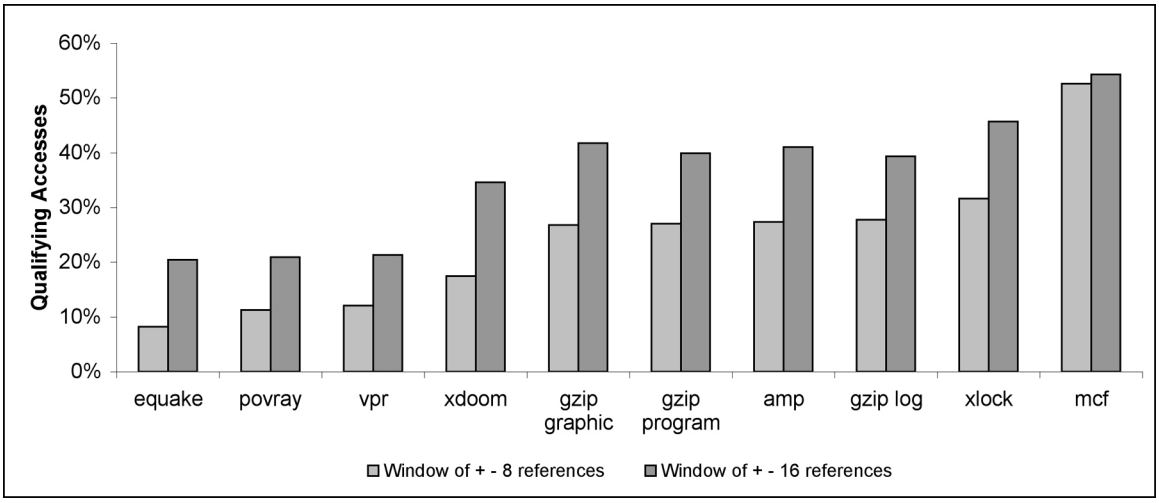

Figure 1: Percentage of Qualifying Accesses

addresses affected, it is more likely that these references may conflict with something else, especially in a direct-mapped cache. Though substantial spikes exist, there

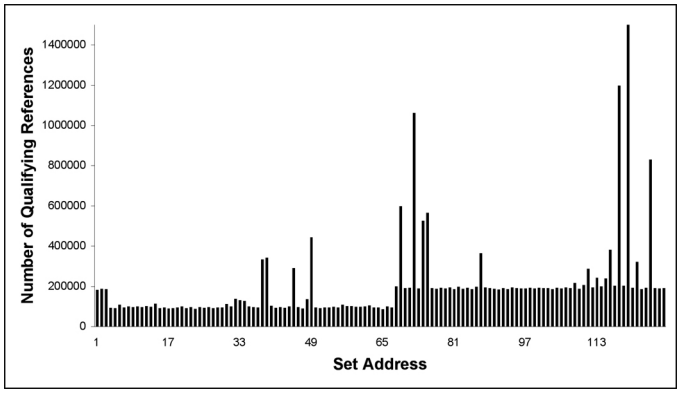

Figure 2: gzip-log Window Size +-8

are well-dispersed accesses throughout the cache, and the absolute numbers of these qualifying accesses (even for the 'small' entries in the graph) are nontrivial. Programs like gzip and MCF showed especially promising results - the qualifying accesses occurred almost equally on almost every set address. Thus the cache is probably becoming widely polluted by allowing these accesses, which do not need to go in the L1, to allocate in L1. (We cannot guarantee a 'polluted' state unless we can guarantee that useful data is being evicted.) Placing these accesses in the sidebuffer rather than the L1 should address the problem.

The last testing that we did was to verify that these qualifying accesses were predictable. It is not beneficial to know that the accesses are polluting the cache if we cannot dynamically predict that they are occurring and can remove them from the cache. To check this we looked at the dispersion of the static instructions by program counter (PC) for all of the memory accesses. We noted from graphing the number of qualifying accesses versus the PC that most of the traces that we ran mapped the memory accesses to very few PCs. In most cases, there were around 700 distinct PCs. Also there were even fewer numbers of PCs that had most of these memory accesses, so we noted that memory access tend to clump around certain PCs. Thus if we know which PCs the qualifying accesses occur at, then we can use the PC to predict when we have a qualifying memory access that should not be allocated in the cache. See Figure 3 for an example of this PC behavior.

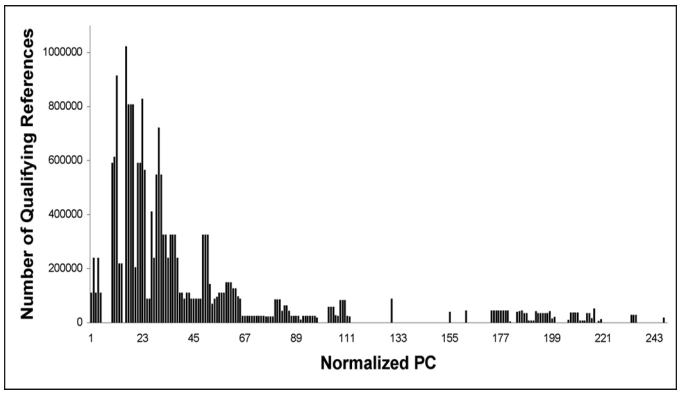

Figure 3: equake PC Analysis, +-128 References

3 Architecture

Through extensive analysis of the runtime memory reference patterns of SPEC2000 benchmarks, we established that there exists significant opportunity to use runtime-available information to classify load instructions into two classes: those that should be allocated a line in the regular cache, and those that should be held aside due to state information that suggests the particular line will not be rewritten. Here, we present an architecture that implements this runtime analysis in such a way as to not lengthen the critical data path.

3.1 Mapping a Solution

Our analysis suggests that the references that classify as 'read-only' (e.g. no writes are encountered within a span of plus or minus 32 memory references) and repetitive (at least three loads from that line) are generated by only a very few static instructions as established in Section 2. This suggests that a simple finite state machine (FSM) can be used to make a prediction about whether a given load instruction is part of a large read-only reference pattern. This FSM is implemented in our *Load History Table* (LHT.) Furthermore, our analysis suggests that increasing the number of distinct cache lines that can be outstanding in a read-only reference pattern reduces early eviction before a line is finished being used. These lines are stored in the *sidebuffer*, which attempts to capture read-only spatial locality. These structures are laid out with the rest of the processor core in Figure 4. Throughout our architecture, we focus on simplicity as defined by wire-delay imposed access time rather than raw transistor count. As technology trends continue to scale smaller, transistor availability will increasingly become a non-issue and performance will be dominated by the wire-delay effects.

3.2 Load History Table

The Load History Table in part works analagously to a traditional branch history table. Upon encountering a load memory reference, the PC of that instruction is used to index a direct-mapped table of history bits. The history bits function as threshhold counters. Since, in this case, the instruction is a load, the count is increased. When there is a store, all that is taken into consideration as input to the LHT is the reference destination address. Stores follow a different path to update state as described below - the result is that the history bits associated with the load to the line now being stored are decremented. Once the threshhold value is met (which we set to eight with four history bits), any new load data for that PC is filled from L2 into the sidebuffer rather than into the L1 cache. However, if the cache line being referenced is already in L1 (that is, we did not cross a cache line boundary since the previous load), the cache line *stays* in L1 and is not moved. Only newly allocated lines are steered by the LHT for simplicity. In a direct-mapped L1, there would be no benefit for evicting a line from L1 to the sidebuffer. However, in an associative L1, it may be beneficial in corner cases to move the line to the sidebuffer in order to prevent the later eviction of 'normal' read-write L1 data.

In order to maintain correct history information, the LHT must also be responsible for maintaining an association between each cache line in the L1 data cache and the PC that 'owns' that cache line. A PC owns a cache line if it was the most recent instruction to load that line. This maintenance is used solely for mapping cache lines to PCs so that, in the case of a store, the LHT can look up the correct history entry to decrement. So, unlike a load, the LHT uses only the store address as input and then calculates the relationship of that address to a PC. For simplicity and presumed speed in physical implementation, we used direct-mapped L1 data caches for our implementation so that the mapping from reference address to PC is a simple table lookup (where the table is the size of the number of L1 data cache lines.) Support for associativity is straightforward to implement but deviates from our goal of reducing latency wherever possible. The LHT layout is pictured in Figure 5.

3.3 Sidebuffer

The sidebuffer serves as storage for read-only lines. It's implemented as a small (8 to 32 lines) fully associative cache with first-in-first-out (FIFO) replacement. Least-recently-used (LRU) would be desirable to evaluate, but the simulation tools we used (Simplescalar 3.0) do not implement a LRU policy. However, due to the nontraditional locality behavior (e.g. spatial locality with very little temporal locality) of the references we are targeting for storage in the sidebuffer, using a simpler replacement policy (such as FIFO) may prove just as effective.

The sidebuffer receives data from the L2 cache upon a fill (when steered by the LHT according to history data.) The sidebuffer is also read-only. Thus, there is no outgoing path to L2 from the sidebuffer. When the sidebuffer encounters a store reference that matches one of the lines inside it, the sidebuffer invalidates that line automatically. The sidebuffer always reports a miss upon store references, and since the data is not in L1, the L2 is triggered as if there was an L1 miss. The LHT then steers the fill data to L1 after the L2 access latency has

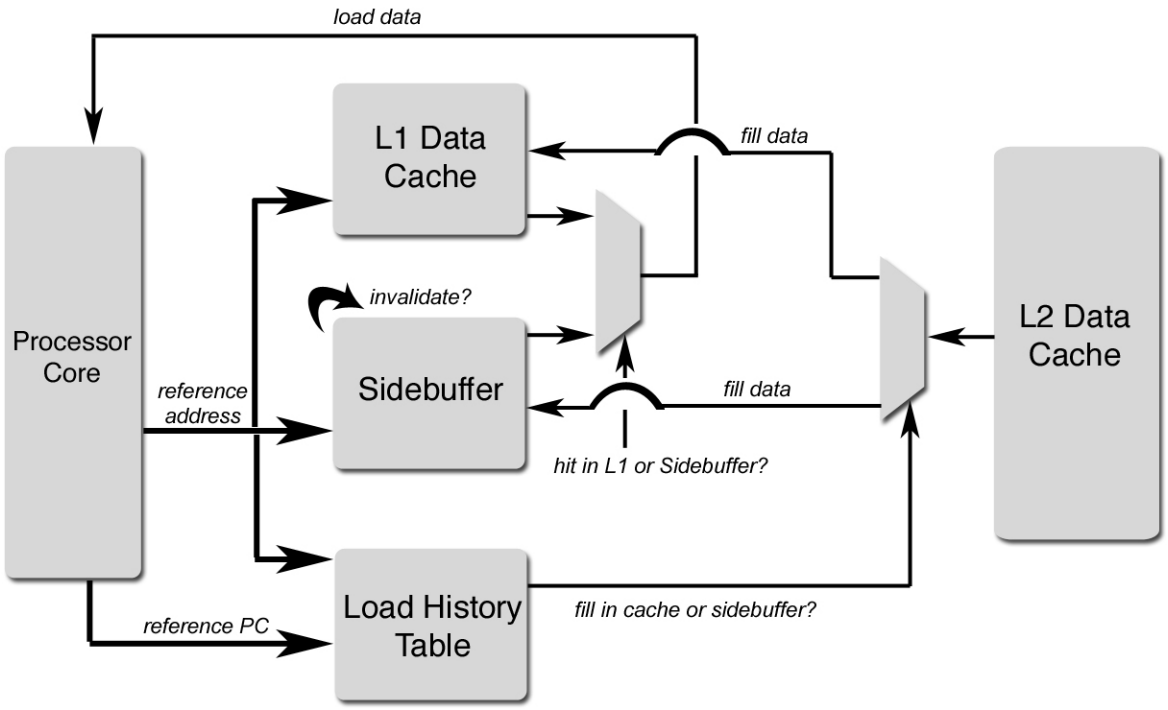

Figure 4: Dynamic Cache Allocation Architecture

expired. We could have optimized the datapath to support transfer of lines from the sidebuffer to L1 to avoid the L2 access penalty, but for simplicity in this initial implementation, we did not. A side effect of this is that the data in L1 and the sidebuffer are mutually exclusive. The only overhead required to maintain this condition is the requirement that the sidebuffer invalidates a line if a write goes to it. If a line in the sidebuffer is invalidated, we say it has been *misallocated*. Due to the mutually exclusive relationship between data in L1 and the sidebuffer, the selection logic for the mux in Figure 4 is extraordinarily simple - after a cache miss and upon completion of the line fill, the data will hit in one and only one of those memories.

4 Implementation

We developed two new architectural structures, the load history table (LHT) and sidebuffer, in order to exploit the memory behavior we identified. In order to determine the effectiveness of these additions, we integrated them into the superscalar architecture simulated by simplescalar. We evaluated cache miss rates and IPC numbers using sim-outorder. By varying the parameters of both these structures using convenient command line options, we explored the design space to pinpoint the most effective use of transistor resources (i.e. the points of diminishing returns) within the realistic bounds of the desired single-cycle delay. Once we determined the optimal LHT and sidebuffer parameters, we compared simulated results using these configurations with results from an unmodified architecture.

4.1 Applications

The memory reference behavior we targeted in our experiments shows up to varying degrees in all programs. In some programs, such as media processing applications, the pattern is very pronounced, while in general purpose programs the patterns appear less predictably. We ran simulations using both general purpose applications (i.e. SPECint and SPECfp) and multimedia applications from the MediaBench suite.

4.2 Exploring the Design Space

Before comparing our modified architecture to an unmodified architecture, we first evaluated how to optimally configure our new structures. We determined the most effective design in terms of transistor count and latency before doing a full performance analysis.

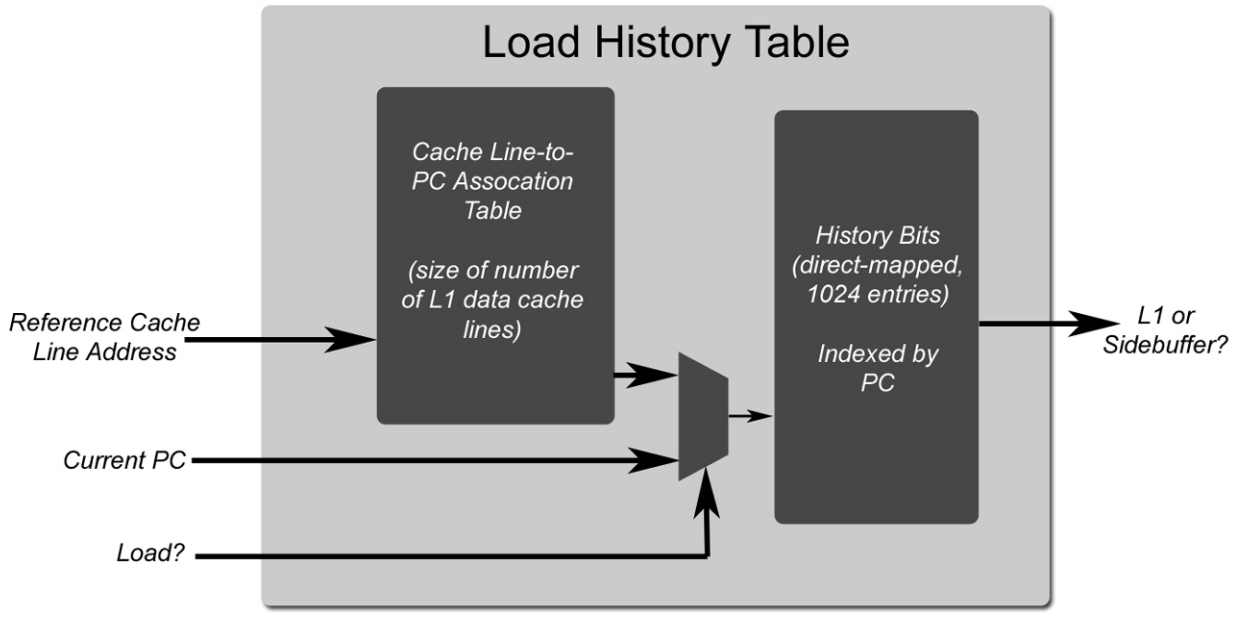

Figure 5: Internal LHT Architecture

Since we intended the inclusion of our changes to affect the hit rate of the level 1 data cache system, we adjusted our parameters with the single goal of minimizing the miss rate. To do this part, we integrated our proposed changes into sim-cache to quickly evaluate L1 data cache miss rates.

Between the two structures (the LHT and sidebuffer), there were four parameters which influence the combined level one miss rates. The LHT has *N* direct-mapped entries, each with *B* bits. Increasing *N* has the effect of reducing the number of conflicts in the table and thereby reducing the number of misallocations to the sidebuffer due to aliasing. The other LHT parameter is *B*, the number of bits per entry, which corresponds to the granularity of the confidence counters. More bits correspond to the ability to capture more history for a particular PC, which eventually relates to better prediction and fewer combined L1 misses due to fewer misallocations in the sidebuffer. Figure 6 shows the effect of different numbers of history bits on the combined L1/Sidebuffer miss rate for the gzip-graphic benchmark. This uses a fixed sidebuffer size of 16 entries.

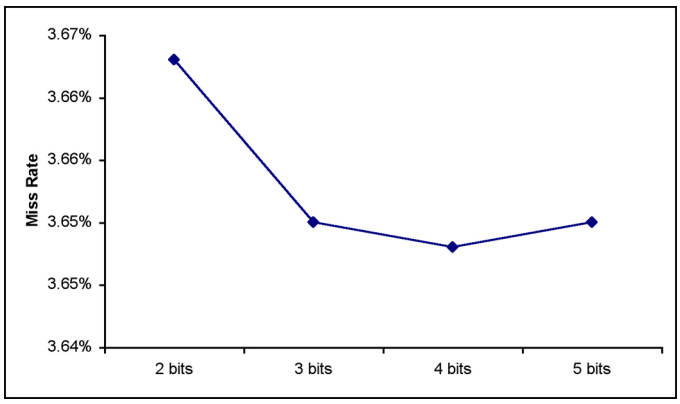

Figure 6: Effect of History Size on Combined L1/Sidebuffer Miss Rate (gzip-graphic)

For each data point, the threshhold is set to be the value at which the most significant bit flips. If the threshhold is too high, our allocation policy into the cache may be too conservative. Therefore, fewer lines will be allocated into the sidebuffer, and we will limit

the benefits of our architecture. This is exhibited in Figure 6 with five history bits. Too few history bits (and thus overly aggressive allocation into the sidebuffer) is detrimental when sidebuffer invalidation happens too often and the L2 load penalty is incurred.

The sidebuffer is implemented as a standard cache structure in simplescalar so it has the standard cache parameters. To maximize the hit rate in the sidebuffer, we decided early on to make it a fully-associative structure. If wire latencies continue to constrain performance, we will eventually want to explore making the sidebuffer set-associative to decrease the access latency. We examined implementations with few (4 to 32) lines in the sidebuffer, so a fully-associative implementation seemed achievable. We assumed a base LRU replacement policy in Simplescalar, though this is currently implemented as FIFO in the Simplescalar 3.0 tools. Evaluating a simpler replacement policy, as discussed above, may be valuable.

Other than the associativity and replacement policy, we are left with two cache parameters for the sidebuffer to examine. First we varied the sidebuffer block size (line size) and found that doubling the L1 block size gave us a small benefit in reducing the miss rate (presumably because of spatial locality). The benefit was small enough so that we decided to avoid the hardware complexity of working with two different L1 line sizes and instead matched the sidebuffer block size with the L1 block size.

The last sidebuffer parameter is the number of lines. Obviously, the greater the number of lines, the better the hit rate of the sidebuffer. But we were not free to make this structure unreasonably large due to our requirement that it be accessible in one cycle. This is crucial since the sidebuffer, like the L1 data cache, is on the critical path of all load instructions (albeit in parallel with the L1 data cache.) Keeping this in mind, we only explored values that we feel would result in a sidebuffer with a 1 cycle access time. Figure 7 shows the effect of varying the number of sidebuffer entries when running the gzip-graphic benchmark with four history bits and a threshhold value of eight.

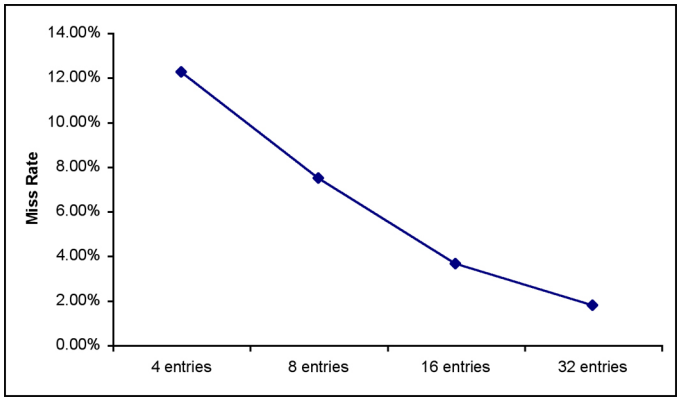

Figure 7: Effect of Sidebuffer Size on Combined L1/Sidebuffer Miss Rate (gzip-graphic)

4.3 Experimental Parameters

We examined a variety of applications from different computing domains on a base architecture simulated by Simplescalar 3.0.

4.3.1 Host Architecture

The final values we chose as optimal for the LHT is a 1024 entry table with four bits per entry, resulting in a 512 byte structure. For the sidebuffer parameters, we determined a 16 line, 32 byte block size cache to be optimal - this corresponds to our estimate at an upper bound for a one-cycle accessible structure. As wire delay effects increasingly dominate in the future and the sidebuffer becomes too slow, our preliminary data suggests we should first reduce associativity and then reduce capacity. The Simplescalar 3.0 sim-outorder defaults we used (when integrated with our changes) are as depicted in Figure 8. It is noteworthy that in following with Simplescalar defaults, all cache latencies are fully pipelined, and cache operations are lockup-free with infinite outstanding references allowed. Unless otherwise specified, the following parameters are constant for all simulations.

Processor Core	**Size/number**
Fetch	4
Decode	4
Issue, inorder	4
Commit	4
RUU	16
Load/Store queue	8
Integer ALU's	4
Integer Multipliers	1
FP ALU's	4
FP Multipliers	1
L1 cache ports	2
Memory Hierarchy	**Latency**
L1	1 cycle
L2	6 cycles
Main Memory	18, 2 cycles
Branch predictor	Binomial using BTB w/ 2-bit counters
Bimod, 2048 entries	

Figure 8: Base Architecture Parameters

4.3.2 Applications Examined

We simulated our modified architecture versus several comparable, non- modified architectures on a wide range of applications. We ran a subset of the SPEC2000 [2] suite including integer applications mcf, vpr, and gzip on a graphic image. We also ran one program from the SPECfp suite: equake. All of these programs were run using the reduced input datasets because of limited computation resources. We also fast-forwarded the simulations past the first million instructions because we are not interested in the performance of the programs while they are initializing. The integer applications were chosen to represent the domain of general purpose applications, while the two floating-point applications were chosen to represent scientific applications.

In addition to SPEC programs, we also ran simulations on a set applications from the MediaBench [6] suite. These programs included a PCM audio compressor and decompressor, EPIC (image compression), a G721 compression and decompression program, and an MPEG2 encoder and decoder. The MediaBench simulations used standard datasets because they were small enough to make the run-time of our simulations reasonable. This time we fast-forwarded through the first hundred thousand instructions. The duration of these benchmarks was considerably shorter than SPEC.

4.4 Effectiveness of LHT and Sidebuffer

With an optimal LHT and sidebuffer size, we then began cycle accurate performance simulations using sim-outorder in the simplescalar suite. We were interested in how well our system performed against the baseline with different sizes of L1 data cache. We ran our experiments over a range of small cache sizes because we believe that as wire delays continue to increase relative to transistor switching time, more and more microprocessors will go to a smaller L1 cache to maintain a one-cycle hit time. As part of our hypothesis, we expect our modified architecture to perform better and better as the L1 data cache is scaled down in size and associativity.

4.5 Comparison Architectures

For baseline comparisons, we compare against a data cache hierarchy with that is identical to Dynamic Allocation in size, latency, and associativity, but lacks a sidebuffer and dynamic management. This should provide some insight into the benefit Dynamic Allocation alone provides, but such a comparison is slightly in our advantage because it ignores the disparity in transistor count. In emerging architectures with larger and larger transistor budgets, we are far less concerned with transistor count than latency, which we have worked to minimize through the use of small structures. Thus, the Dynamic Allocation overhead to latency should be minimal.

The current choice among some architects for use of the increasing transistor budget is to increase the size of the L1 while accepting the two cycle access time and hoping the increased hit rate will offset the loss. State-of-the-art processors currently are forced to deal with this tradeoff. As such, we compared against data cache hierarchies similar to those in current high-performance computers. For our *medium* configuration, we used size, associativity, and latency parameters similar to those in the Intel Itanium [5]. For the *large* configuration, we used parameters similar to those in the IBM POWER4 [4]. We assumed a one-cycle access time for our configuration due to the careful analyzation of our datapath for bottlenecks and the simplification of all serial-access long-delay units in our architecture.

5 Experimental Results

5.1 SPEC2000 Performance

Figure 9 shows the combined L1 miss rate of Dynamic

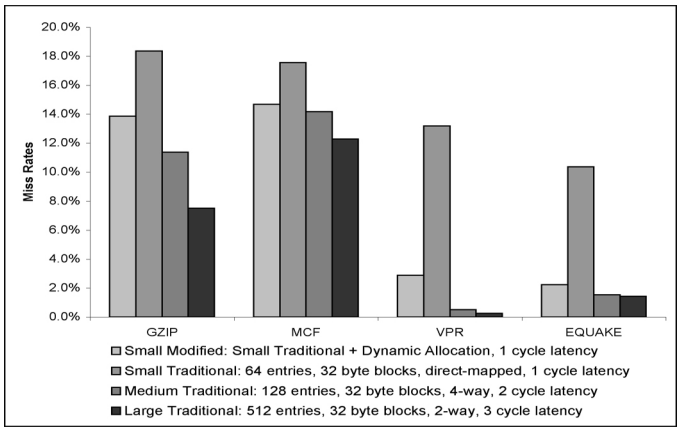

Figure 9: SPEC2000 L1 Data Cache Miss Rates

Allocation versus that of the traditional configurations. Miss rate here refers to the total miss rate between the sidebuffer and the L1 data cache. Dynamic Allocation mostly achieves miss rates comparable to those of far larger (8 and 16 times larger for the *medium* and *large* configurations, respectively) caches with much more associativity, and it drastically outperforms the miss rate of the baseline configuration with the same L1 cache size. While the overall cache size for Dynamic Allocation is larger by 16 cache lines (the size of the side-

buffer), this does not account for such a pronounced difference in miss rate. These benchmarks clearly benefit from the isolation and separation of the types of locality that Dynamic Allocation demarcates. Isolating the references with approximately little temporal locality in the sidebuffer leaves the main L1 free for use by the rest of the application. There are enough *useful* non-sidebuffer references to benefit from the decreased contention in the main L1 data cache such that miss rates with this much smaller yet fast architecture are competitive. Of particular note here is that Dynamic Allocation manages to get "in the ballpark" of the miss rates of significantly more complex and larger memory structures for these general-purpose applications.

Figure 10 shows the performance numbers of Dynamic Allocation versus configurations similar to current designs when latency is taken into consideration. Due to latency-hiding techniques in our baseline configuration such as out-of-order issue, it is not a given that performance will suffer directly according to miss rate [3]. Dynamic Allocation outperforms all other configurations, sometimes by wide margins. Though we did not complete results under all configurations for other SPEC benchmarks, Dynamic Allocation also outperformed all other configurations for ammp, which was the only other SPEC benchmark we ran against the fully-configured modified simulator. Clearly, these benchmarks are sensitive to memory access latency. The smallest speedup we see is 2.4 % for gzip-graphic versus the *medium traditional* configuration, while the largest was 16.7 % for equake versus a cache 16 times as large. The arithmetic mean speedup was 8.66 %. While these numbers are impressive by themselves, the important aspect of these results is future scalability. As wire delay becomes even more of an issue, the disparity between the large, associative structures and Dynamic Allocation will be even more pronounced.

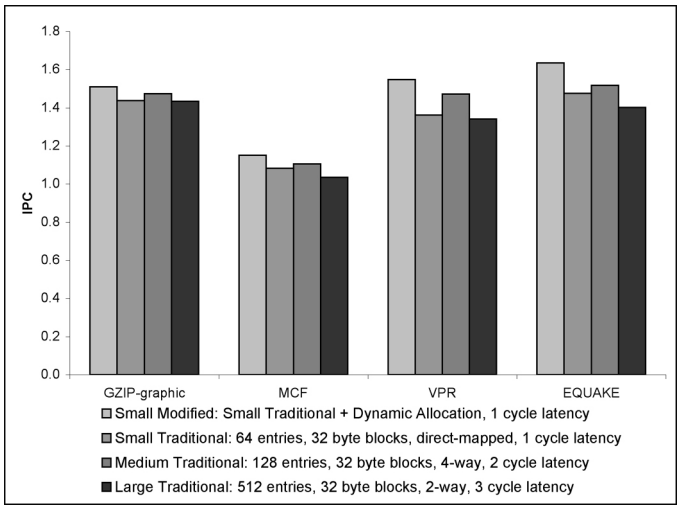

Figure 10: SPEC2000 Performance Results

Figures 11 and 12 plot the miss rate of GZIP-graphic and equake, respectively, as the size of the L1 cache is reduced for varying sidebuffer sizes and configurations. The '32' and '64' bytes refers to the size of the cache lines for those plots. As L1 becomes smaller and smaller, the miss rates for Dynamic Allocation configurations do increase. However, they do not increase as quickly as unmodified configurations, even for Dynamic Allocation with very small sidebuffer sizes. Thus, as L1 data caches get smaller, Dynamic Allocation is even more beneficial.

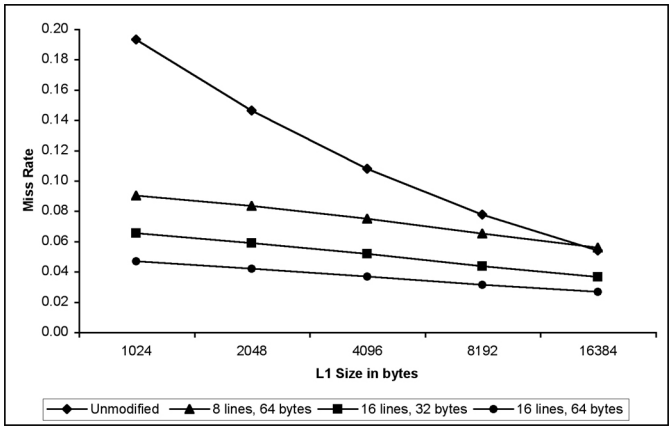

Figure 11: Miss Rate as L1 Decreases - GZIP-graphic

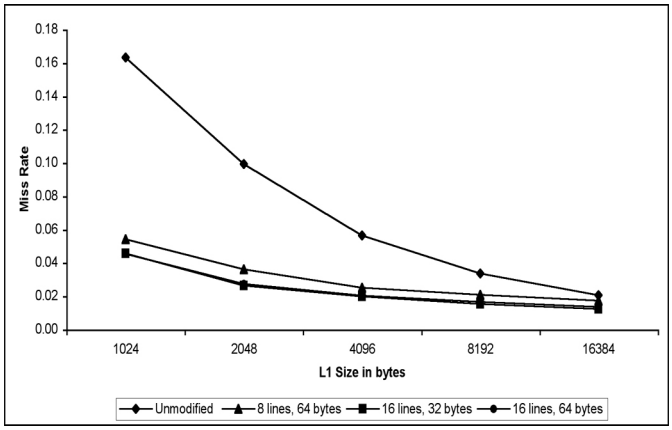

Figure 12: Miss Rate as L1 Decreases - equake

5.2 MediaBench Results

Figure 13 shows the combined (as previously defined) L1 miss rates of Dynamic Allocation versus the traditional configurations described above for the MediaBench suite. Most noteworty is that miss rate does

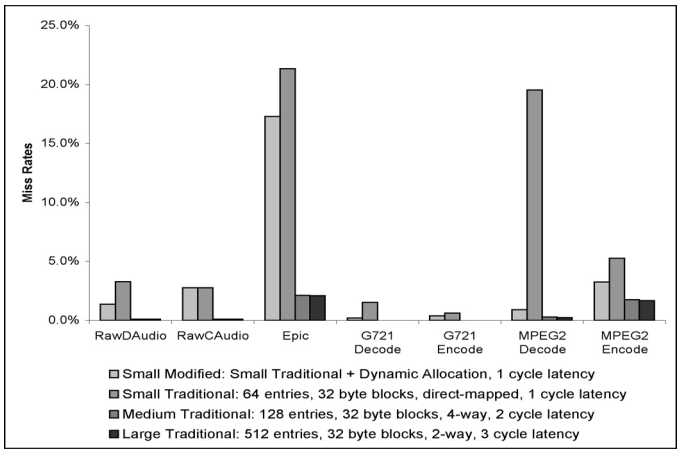

Figure 13: MediaBench L1 Data Cache Miss Rates

not vary significantly with increased cache size past the *small* configuration. This suggests that these types of applications do not benefit greatly from any cache due to their access patterns and typically large working sets, though they have a minimum amount of locality that must be captured. Dynamic Allocation helps meet this minimum and get in the neighborhood of miss rates of the larger, more complex configurations for some applications, such as MPEG2 decode and encode, and G721 encode and decode. However, since most of these applications are in fact small computation kernels that operate only on their incoming and outgoing data, there may be very few (if any) memory references that would not be targeted by the sidebuffer and thus there may be very few available references to take advantage of the less-conflicted main L1 data cache. While sidebuffer access rates for these benchmarks are higher than for SPEC, that does not imply that the regular L1 accesses are well-distributed or that they will benefit from the less-conflicted cache. Furthermore, some applications clearly do not benefit at all from caching (after a certain bound) as continuing to increase cache size and associativity does not show an improvement in miss rate. Epic has some unique behavior and is described below.

Figure 14 presents the performance of the MediaBench suite with latencies applied as before. As suggested by the miss rate numbers, many MediaBench applications do not benefit greatly from Dynamic Allocation. The arithmetic mean speedup was 4.11 %.

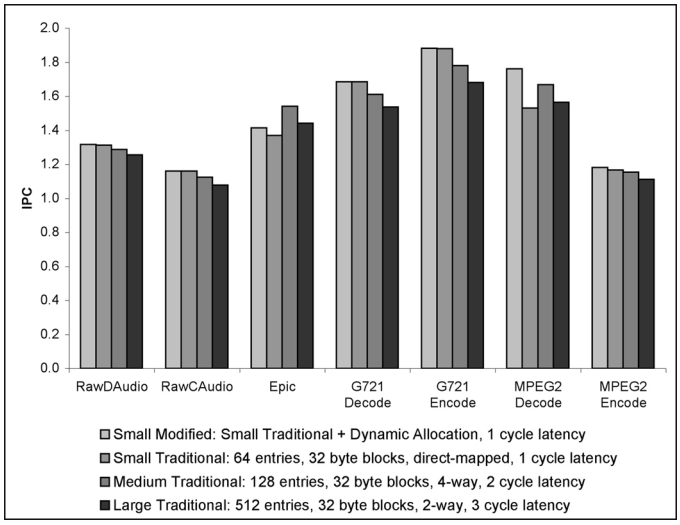

Figure 14: MediaBench Performance Results

Unlike the SPEC results, the baseline *small traditional* configuration with its single-cycle latency performs almost identically to Dynamic Allocation. It is noteworthy to compare MPEG2 decode with MPEG2 encode. MPEG2 decode experiences *cacheable* behavior where adding a larger cache helps decrease misses thus increasing performance. Furthermore, it is quite obvious from the miss rate graph that MPEG2 decode experiences a dramatic improvement in miss rate with Dynamic Allocation. MPEG2 encode, however, has fairly good behavior with respect to miss rate even with the smallest cache, and does not get substantially better as decode does. Thus, for applications where miss rate is nontrivial, Dynamic Allocation provides better performance when latencies are applied to the larger, slower memory hierarchies.

Also of particular note is the Epic benchmark. While Dynamic Allocation did help to reduce the miss rate for equal-size caches, it is still a capacity-bound problem even though Dynamic Allocation attempts to alleviate the capacity-filling effects of many read-only, striding applications. 44.8 % of the memory accesses in Epic are to the sidebuffer, but among the references to regular (including flushes incurred by the sidebuffer), the miss rate is a significant 29.98 %. Epic uses a huffman encoding algorithm that has a large internal symbol tree that is sorted in a read-write manner, and moreover is not in contiguous memory (e.g. it is represented as a node-based structure.) Thus, it is likely that this large read-write structure is organized and referenced in such a way as to make it unwieldy in a very small direct-mapped L1 cache even after conflicts introduced

by read-only data streams are removed. Increasing the associativity of the small cache reduced miss rates to levels comparable with both the larger configurations.

6 Related Work

The field of intelligent cache management for the purposes of improving overall performance and efficiency has been explored in parallel, yet different and sometimes complimentary ways. Jouppi [7] proposed reducing conflicts in direct-mapped caches by adding small associative caches such as victim caches. However, victim caches do not address the problem of large working sets of the kind targeted by Dynamic Allocation that may be larger than the size of the victim cache.

Jouppi also proposed stream buffering on misses, while Baer and Chen [8] extended the idea to use an FSM to predict the references to be filled in the stream buffer. In this vein, the stream buffer can be compared to the functionality provided by the separate storage of the sidebuffer, but there are important distinctions: previous predictive measures were by expected data stride and ran ahead of the program rather than Dynamic Allocation's per-PC analysis of the current instruction, and these were methods of guessing what should be fetched thus increasing L1-to-L2 bandwidth consumed and pressure on branch prediction accuracy. This is often at the expense of wasted bandwidth. Dynamic Allocation does not address main memory latency, and while it may increase bandwidth consumed in rare corner cases when sidebuffer entries must be refetched from L2 if the L2 entry has been evicted (e.g. during an L1 allocation following an attempted write to a sidbuffer line), but this is almost certainly less common than prefetching waste. Analysis of sidebuffer access and L1 reload patterns show this is in fact extremely rare. Compiler algorithms [9] for prefetching work complimentarily to the efforts of Dynamic Allocation since they map to specific instructions in applications.

McFarling [10] proposed an FSM approach to excluding specific lines from allocation in a direct-mapped cache in order to reduce conflicts in direct-mapped caches. His work focuses almost entirely on instruction caching, and the FSM he uses examines state on a per-cache-line basis; it does not consider the static instruction that generated that reference. Furthermore, Dynamic Allocation provides specific management of the 'excluded' (sidebuffer) cache lines and gains performance from the flexibility and the size of the sidebuffer. Most significantly, however, is that dynamic exclusion gives weight to the data pre-existing in the cache by recording the previous hits on given cache lines. While this may have the same effect as Dynamic Allocation in some cases, in others it may be overly pessimistic about the usefulness of incoming data. Dynamic Allocation instead profiles the locality of specific instructions rather than relying on specific cache line state. The Dynamic Allocation FSM is also considerably simpler than McFarling's and does not allow quick thrashing between the sidebuffer and main L1.

González et. al. [12] proposed splitting the L1 data cache into two separate structures, a *dual data cache*, to capture spatial and temporal locality separately. This design is significantly more complex than Dynamic Allocation because it allows a cache line to be valid in both caches at once and allows both to be read-write. Furthermore, the locality prediction table they use is very accurate, but is far more complex than our simple threshhold counting mechanism - it keeps history data about strides, length, last-address-accessed, and state; updating this is nontrivial in hardware. The overall architecture is unlikely to scale well. Dynamic Allocation, in contrast, makes a much looser approximation about the type of locality and achieves reasonable performance with a much simpler design.

7 Concluding Remarks

Dynamic Cache Allocation is a scalable mechanism for enhancing the performance of small, simple data caches in a wide array of applications. Dynamic Cache Allocation improves data cache miss rates substantially over a direct-mapped cache. It also outperforms the data cache hierarchies of modern day processors when real-world latencies are applied. Most importantly, as cache size and associativity are reduced in order to keep up with the latency requirements of increasing clock rates, Dynamic Cache Allocation's cache miss rate diverges from the miss rate of standard cache configurations by wider and wider margins. This suggests the benefit of Dynamic Allocation will increase as wire delay in memories continues to increase latency. Dynamic Allocation offers a lower-latency alternative to achieving miss rates comparable to those of larger, more complex caches.

Acknowledgements

We would like to thank Mike Brogioli for directing us to MediaBench and helping us get started with it.

References

[1] Keith D. Cooper and Timothy J. Harvey, ”Compiler-Controlled Memory”, *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[2] John L. Henning, ”SPEC CPU2000: Measuring CPU Performance in the New Millenium”, *Computer*, July 2000.

[3] John Hennessey and David Patterson, *Computer Architecture: A Quantitative Approach. Third Edition*, Morgan Kaufman Publishers, San Francisco, California, 2002.

[4] J. M. Tendler, J. S. Dodson, J. S. Fields, Jr., H. Le, and B. Sinharoy, ”POWER4 System Architecture”, *IBM Journal of Research and Development*, January 2002.

[5] Harsh Sharangpani and Ken Arora, ”Itanium Processor Microarchitecture”, *IEEE Micro*, September/October 2000.

[6] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith, ”MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems”, *Proceedings of the 30th International Symposium on Microarchitecture*, December 1997.

[7] Noman P. Jouppi, ”Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers”, *Proceedings of the 17th International Symposium on Computer Architecture*, May 1990.

[8] Jean-Loup Baer and Tien-Fu Chen, ”An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty”, *Proceedings of Supercomputing*, November 1991.

[9] Todd C. Mowry, Monica S. Lam, and Anoop Gupta, ”Design and Evaluation of a Compiler Algorithm for Prefetching”, *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.

[10] Scott McFarling, ”Cache Replacement with Dynamic Exclusion”, *Proceedings of the 19th International Symposium on Computer Architecture*, April 1992.

[11] Bob Amos, Sanjay Deshpande, Mike Mayfield, and Frank O’Connell, ”RS/6000 SP 375 MHz POWER3 SMP High Node, *IBM Online Whitepaper Respository*, http://www-1.ibm.com/servers/eserver/pseries/hardware/whitepapers/nighthawk.html, August 2001.

[12] Antonio González, Carlos Aliagas, Mateo Valero, ”A Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality”, *Proceedings of the 9th International Conference on Supercomputing*, July 1995.