Run-time Cache Hierarchy Management via Reference Analysis

Teresa L. Johnson* Wen-mei W. Hwu
Center for Reliable and High-Performance Computing
University of Illinois
Urbana-Champaign, IL 61801
{tjohnson,hwu}@crhc.uiuc.edu

Technical Report IMPACT-96-01
August 1996

Abstract

Improvements in memory speeds have not kept pace with increasing processor clock frequency and improved exploitation of instruction-level parallelism. Consequently, the gap between processor and memory speeds is expected to grow. The increased memory latency seen by the processor not only increases the number of execution cycles spent waiting for memory accesses to complete, but can also degrade the compiler-generated instruction schedule. One solution to this growing problem is to reduce the number of cache misses by increasing the effectiveness of the cache hierarchy. In this paper we present a technique for dynamic analysis of program data access behavior, which is then used to proactively guide the placement of data within the cache hierarchy in a location-sensitive manner. We introduce the concept of a *macroblock*, which allows us to feasibly characterize the memory locations accessed by a program, and a *Memory Address Table* (*MAT*), which performs the dynamic reference analysis. Our technique is fully compatible with existing Instruction Set Architectures. Results from detailed simulations of several integer programs show significant speedups.

1 Introduction

As processor performance improvements continue to dwarf improvements in main memory performance [1], the cache miss penalty will begin to dominate the cycle counts of many applications. The large improvements in processor performance are due both to better circuit design and fabrication technology, which reduce the cycle time, and to better Instruction-Level Parallelism (ILP) techniques, which increase the Instructions executed Per Cycle (IPC). This disparity between the processor and memory performance will make cache misses more and more expensive. Not only do the cache misses result in more processor stall cycles, but in processors with dynamic scheduling they can also disrupt the compiler-generated ILP schedule. Also, the data caches are not always used efficiently. In numeric programs there are several known compiler techniques

*Supported by a National Science Foundation Graduate Fellowship.

for optimizing data cache performance. However, integer programs often have irregular access patterns that are more difficult for the compiler to optimize. This paper focuses on data cache performance optimization for integer programs.

In order to increase data cache effectiveness for integer programs we have investigated methods of *cache hierarchy management*, where we proactively control the movement and placement of data in the hierarchy based on the data usage characteristics. In this paper we present a microarchitecture scheme where the hardware determines data placement based on dynamic referencing behavior. This scheme is fully compatible with existing Instruction Set Architectures.

Present cache management methods are location-insensitive in that their policies respect the operation that requested the memory access, rather than the address being accessed [3] [10]. This can result in poor choices, since the same instruction can access many locations with varying locality. Our scheme seeks to overcome this limitation by managing the cache in a manner that is sensitive to the memory locations accessed. Since the number of memory locations is infeasibly large, we introduce the notion of a *macroblock*. A macroblock is a contiguous block of memory that is large enough so that the maintainance overhead is reasonable, but small enough so that the access pattern of the memory addresses within each macroblock is statistically uniform. A hardware mechanism called the *Memory Address Table* (*MAT*) is introduced to maintain and utilize the access patterns of the macroblocks to direct data placement in the cache hierarchy. We show that this extension to the cache microarchitecture significantly improves the overall performance of integer applications. The improvements are due to increased cache hit rates and reduced cache handling latencies.

The remainder of this paper is organized as follows: Section 2 discusses related work; Section 3 contains a case study of a particular benchmark as well as some main concepts used to motivate and develop this work; Section 4 discusses the hardware implementation; Section 5 presents simulation results; Section 5.3 performs a cost analysis of the added hardware; and Section 6 concludes with future directions.

2 Related Work

Several methods exist to overlap memory accesses with other computation in the processor, thereby hiding the memory latency. Write buffers can often successfully hide the latency of write misses by buffering the write data until the bus is idle. Non-blocking caches allow multiple outstanding load misses without stalling the processor in order to overlap load miss latency with other computation that does not consume the result of an outstanding load miss [2]. Prefetching attempts to fetch data from main memory to the cache before it is needed, which also overlaps the load miss latency with other computation. Both hardware [3] [4] [5] [6] and software [7] [8] [9] prefetching methods for uniprocessor machines have been proposed. However, most of these methods focus on prefetching regular array accesses within loops [1], which are access patterns primarily found in numerical applications. There is also much prior work on prefetching in multiprocessors, but since their focus is even more on optimizing numerical applications, we will not review them here.

While the above schemes attempt to hide the latency of load misses, our work focuses on reducing the effective memory latency seen by the processor through the reduction of conflict misses and their effects. Victim caches also attempt to reduce the number of cache misses, in particular by reducing conflict misses in caches with less associativity [4]. While victim caches work well for many programs, as we will show in Section 5.2, they will not work as well for programs that have very large working sets.

Methods for both static and dynamic cache bypassing have also been investigated. In [10], Tyson *et al.* proposed a method where loads are marked for cache bypass either statically by the compiler, or at dynamically at run-time. While we also investigate cache bypassing, our work differs in several key aspects. First, Tyson *et al.* use miss behavior of the load as the main decision metric for determining whether to bypass that load's data. Our work focuses on the reuse behavior, because data that tend to miss may still have large amounts of locality that would result in reuse while in the cache. Secondly, they decide whether to bypass data based on the particular load referencing that data. As we will show in Section 3.1, a single load instruction may reference data with widely varying access patterns. Therefore our work determines whether to bypass based on the address of that data. As a result, we see both increases cache hit ratios and decreases in the bus traffic and total cycle counts, while they achieve a decrease in the bus traffic at the expense of a small drop in the cache hit ratios.

3 Concepts

3.1 Case Study

To understand some of the inefficiencies of current cache hierarchies it is helpful to first examine the accessing behavior of a particular application in detail. Figure 1 shows the main loop body of the *026.compress* program

[1] By this we mean arrays indexed by the loop iteration variable, or some other induction variable.

from the *SPEC'92* benchmark suite [11]. This loop body comprises of over 90% of the execution time of *compress*. The majority of the memory accesses in *compress* are to its hash tables, *htab* and *codetab* (the lines containing the hash table load accesses are numbered). Because of the large size of these hash tables (*htab* is larger than 256K bytes and *codetab* is larger than 128K bytes), and the fact that the hash table accesses have little temporal or spatial locality, there is very little reuse in a first-level data cache.

```
    while ( (c = getchar()) != EOF ) {
        in_count++;
        fcode = (long) (((long) c << maxbits) + ent);
        i = ((c << hshift) ^ ent); /* xor hashing */

1.      if ( htabof (i) == fcode ) {
2.          ent = codetabof (i);
            continue;
        } else if ( (long)htabof (i) < 0 ) /* empty slot */
            goto nomatch;
        disp = hsize_reg - i; /* secondary hash (after G. Knott) */
        if ( i == 0 )
            disp = 1;
probe:
        if ( (i -= disp) < 0 )
            i += hsize_reg;

3.      if ( htabof (i) == fcode ) {
4.          ent = codetabof (i);
            continue;
        }
        if ( (long)htabof (i) > 0 )
            goto probe;
nomatch:
        output ( (code_int) ent );
        out_count++;
        ent = c;
        if ( free_ent < maxmaxcode ) {
            codetabof (i) = free_ent++; /* code -> hashtable */
            htabof (i) = fcode;
        }
        else if ( (count_int)in_count >= checkpoint && block_compress )
            cl_block ();
    }
```

Figure 1: Compress Main Loop Code

Table 1 shows the hash table loads' dynamic execution counts, miss ratios and reuse ratios obtained via memory access profiling. A simple cache simulation was performed to determine whether each of the accesses was a first-level cache hit or miss in a direct-mapped 16K cache with 32-byte lines [2]. Also, the profiler kept track of reuse ratios [3]. The table shows that, indeed, the hash table load accesses have high miss ratios and little reuse of the accessed data.

In order to obtain a clearer picture of how the hash tables are accessed throughout the dynamic execution of the program, we profiled the accesses as explained above and plotted the address distribution for a given

[2] This profiler is a simplified version of the detailed simulator used to generate the results presented in Section 5.2. Unlike the simulator, the profiler assumes a single-issue, in-order machine and zero-cycle load latencies to simplify handling back-to-back accesses to the same cache block. More details on the simulator are given in Section 5.1.3.

[3] The reuse ratio is calculated in the following way. If load A accesses a cache block (whether a hit or miss), on a following hit by load B to that cache block the reuse counter for load A is incremented once. If another access by load C is a hit to the same cache block, the counter for load B is incremented, and so on. The total number of reuses counted for a load, divided by its dynamic execution count, is that load's reuse ratio. Therefore, the hit ratio and reuse ratio are not correlated, because some of the hits will have reuse, as will some of the misses, and some will not.

Line	Hash Table	Dynamic Execution Count	Miss Ratio	Reuse Ratio
1	htab	999999	78.9%	29.2%
2	codetab	566776	70.8%	30.0%
3	htab	1803911	91.4%	15.6%
4	codetab	182336	89.1%	11.5%

Table 1: Profiling Statistics for Hash Table Load Accesses (direct-mapped, 16K-byte data cache with 32-byte lines, single-issue processor).

execution phase.

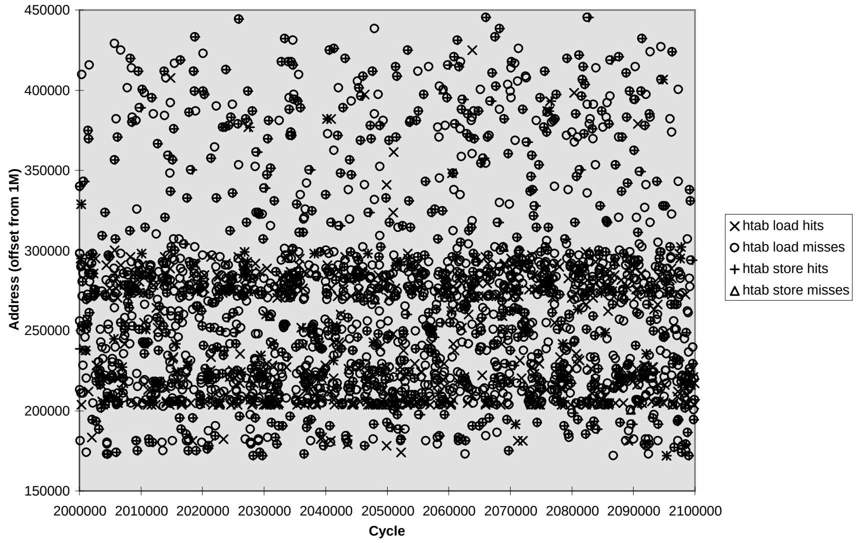
The profiling results for a sample 100000 cycle execution phase of *compress* are shown in Figure 2. The memory access distribution for *htab* is shown in Figure 2a. As the *htab* distribution shows, much of *htab* is relatively sparsely accessed, except for two bands that are heavily accessed. These bands are located roughly from addresses 200000 to 220000 and 257000 to 300000 (all addresses are offsets from a base address of 1048576, or 1M). Most of these accesses are loads resulting in cache misses [4]. Looking at several other execution phases of *compress* shows that this pattern remains the same throughout the execution.

Analogous to Figure 2a, Figure 2b shows the access distribution for codetab. The access patterns of the two figures look similar since *codetab* is accessed with the same index as *htab*. However, Figure 2b is sparser than Figure 2a, since *codetab* is accessed only when the corresponding entry in *htab* has been accessed and matched the current input sequence.

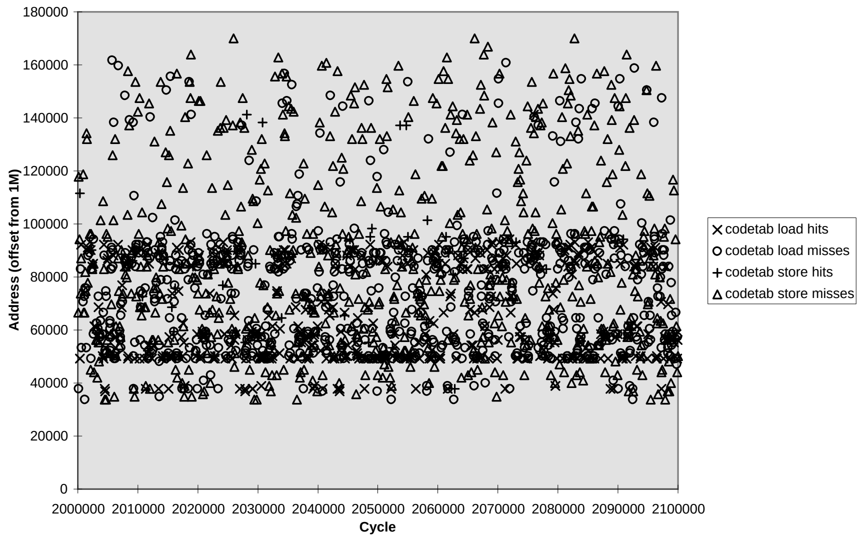
The memory access distributions of Figure 2 illustrate the inherent problem with schemes that determine how to handle data based on the particular load instruction that requested the access. In *compress* there are only two load instructions in the main loop body that access *htab*, however from the distributions we see that these loads can access data with dramatically different usage patterns, even during small time intervals. Schemes that decide where to place the data in the cache hierarchy based on the load instruction accessing that data, whether in a static or a dynamic manner, must treat all data accessed by each load instruction as if it was uniform in behavior. However, as Figure 2a shows, in very small time intervals data with widely varying access patterns is accessed. Therefore, information is lost and some data will be mishandled.

Figure 3 shows how we would like to handle accesses to data with different usage frequencies. In Figure 3a accesses to differently accessed regions of memory will map into the same cache lines, causing conflict misses. Assume that an access to a block in an infrequently accessed region of the memory misses in cache, and the

[4] While the load miss markers do obscure some of the load hit markers, removing the load miss points from the graph shows few additional load hit points.

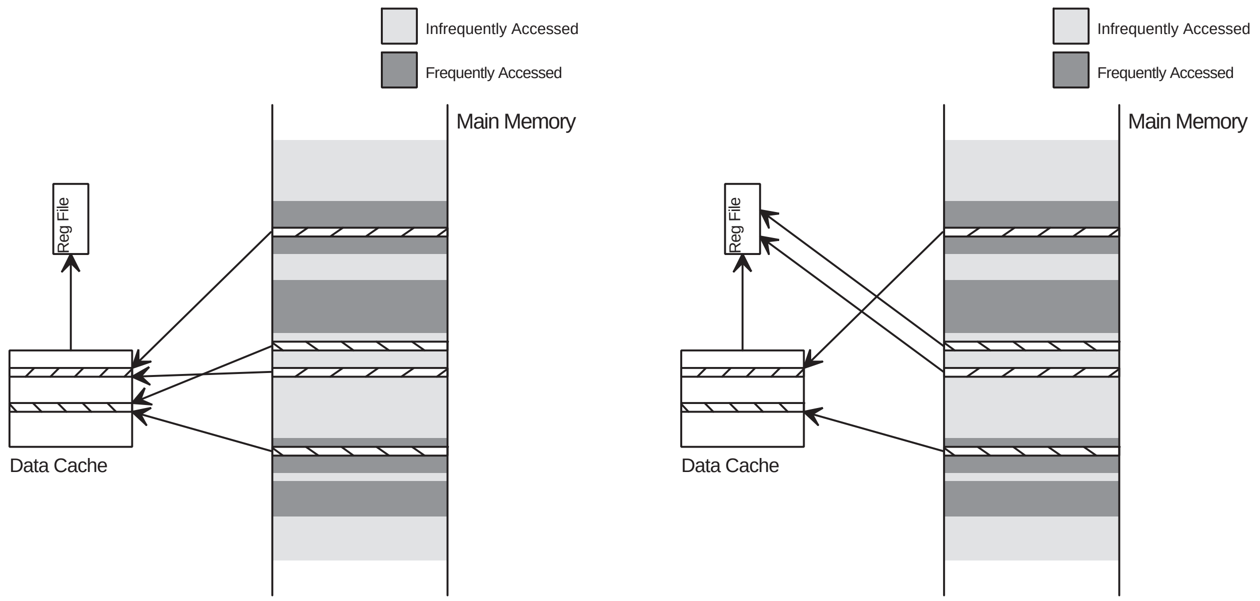


(a) Htab



(a) Codetab

Figure 2: Memory Access Distributions for htab and codetab



(a) Conflicts Between Data with Different Usage Patterns (b) Bypassing Less Frequently Accessed Addresses

Figure 3: Conflict Misses in Compress.

conflicting block that would be replaced from the cache under a normal cache management policy is from a heavily accessed region. Instead of replacing the heavily accessed block, which has a much greater chance of being reused in the near future, we would like the missing block to bypass the cache. In this case the missing block would be sent directly to the register file, and would not be placed in the cache, as illustrated in Figure 3b. Bypassing infrequently accessed data when it conflicts with much more frequently accessed data will result in less cache pollution, and therefore increased reuse of more frequently accessed data, resulting in an overall increase in the hit ratio. Also, when bypassing the cache, only the element size (rather than the cache line size) needs to be fetched, which will further reduce bus traffic. To perform this selective cache bypassing, we need some method of tracking the access behavior of different memory regions.

3.2 Macroblocks

Ideally, we would like to keep track of the usage frequencies of all cache block size data in memory. While this would give us the most accurate information, it would result in an unmanageably large amount of information. Instead, we combine groups of adjacent cache block size data into larger blocks called *macroblocks*. The size of the macroblocks should be large enough so that the total number of macroblocks residing in the accessed portion of memory is not too large, but small enough so that the accessing frequency of the cache blocks

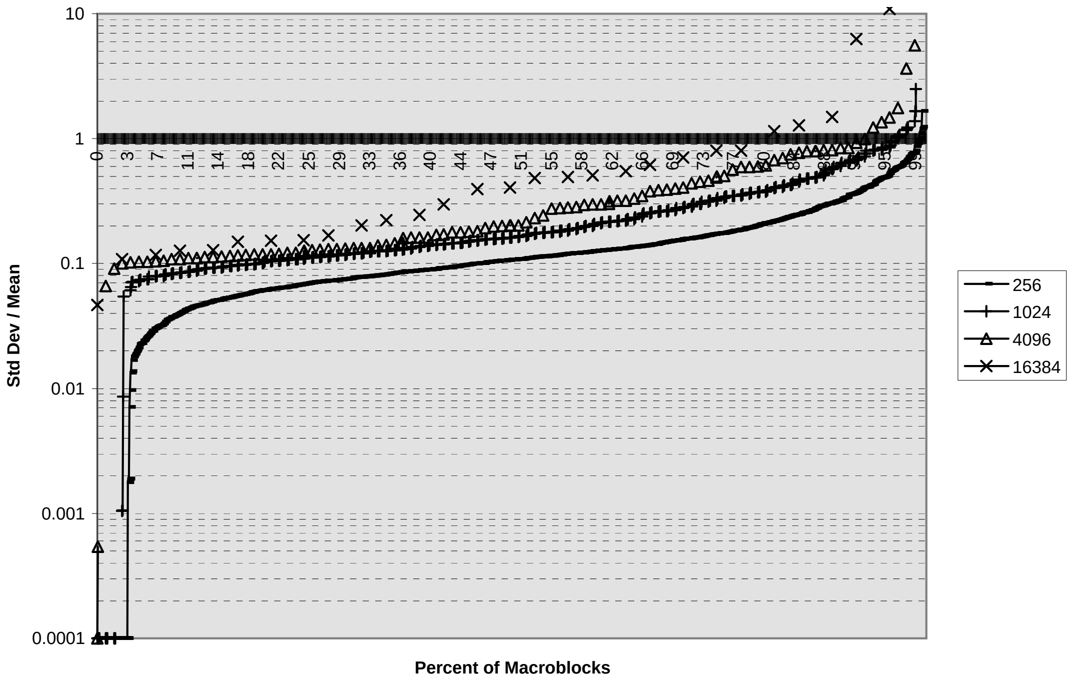


Figure 4: Macroblock Access Uniformity

contained within each macroblock is relatively uniform. If we can keep track of each macroblock's accessing frequency through some hardware mechanism, then it is possible to determine on a macroblock basis whether or not to cache the contained data.

In order to determine the best size of a macroblock in practice, we studied the uniformity of cache block access frequencies within the macroblocks for several macroblock sizes. The number of accesses to each cache block in memory was first profiled. Then for each macroblock size, and for each macroblock in memory, the mean and standard deviation of the number of accesses to the cache blocks contained within that macroblock were computed. Finally, the standard deviation divided by the mean (in order to normalize the results) for all macroblocks were sorted and plotted on a log_{10} scale. For a high intra-macroblock accessing uniformity most of the macroblocks should have relatively small standard deviations.

Figure 4 shows the results for macroblock sizes of 256, 1K, 4K and 16K-bytes. The y-axis is a log_{10} scale of the standard deviation divided by the mean, and the x-axis is the percentage of macroblocks with a standard deviation divided by mean less than or equal to the plotted value. The curve for 256-byte macroblocks has the lowest standard deviations, however using 256-byte macroblocks may result in too many total macroblocks to feasibly track. Using 1K-byte macroblocks, which would include four times as many cache blocks per macroblock, may be much more feasible and still result in most macroblocks having high uniformity. For this size about 60% of the macroblocks lie within 20% of the mean, with almost 90% within 50% of the mean. The 4K-byte curve is only slightly higher, still with most of its macroblocks within 20% of the mean. The 16K-byte curve does not look quite as good. As will be discussed when we present our simulation results in Section 5.2, we chose the 1K and 4K-byte macroblock sizes for our study.

4 Hardware

4.1 Memory Address Table

As discussed in Section 3.2, we would like to determine on a per macroblock basis whether to cache or to bypass the contained data on a miss. In order to keep track of the macroblocks we use a table in hardware called a *Memory Address Table* or *MAT*. The MAT ideally contains an entry for each macroblock. Each entry in the table is a saturating counter, where the counter value represents the frequency of accesses to the corresponding macroblock. Currently we are using a direct-mapped MAT with an 8-bit counter.

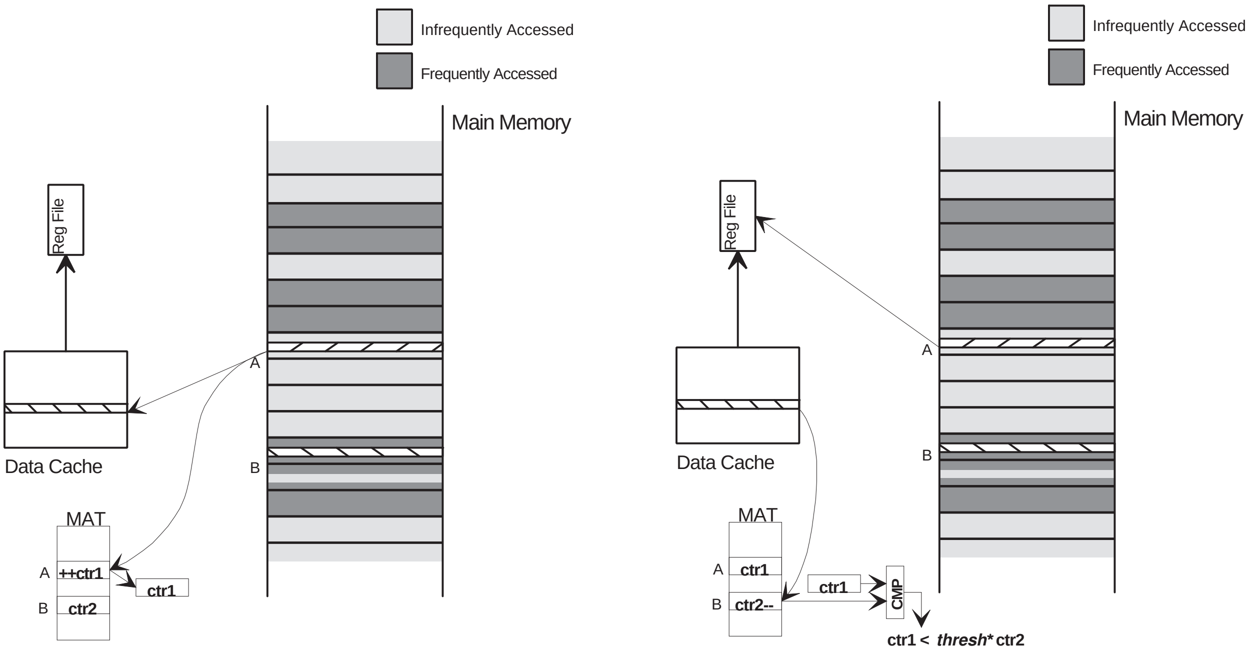
On a memory access, a lookup in the MAT of the corresponding macroblock entry is performed in parallel with the data cache access. If no entry is found, a new entry with a counter value of zero is allocated. If an entry is found, the counter is incremented. Also, the counter value (*ctr1*) must be saved in a register for possible use in the next step. An example of this operation is shown in Figure 5a, where block A is accessed.

If the data cache access resulted in a hit, the access proceeds as normal, and the counter value is ignored. On the other hand, if the access resulted in a cache miss, the cache controller must lookup the MAT counter corresponding to the cache block that would be replaced to determine which data is more heavily accessed, and therefore more likely to be reused in cache, as shown in Figure 5b. This counter value (*ctr2*) is then decremented and compared to the counter value corresponding to the missing access. The actual comparison performed is:

$$ctr1 < thresh * ctr2 \qquad (1)$$

where *thresh* is a fraction from zero to one. If the above inequality is satisfied then the fetched data will bypass the cache. Otherwise it is placed in the cache, replacing the existing cached data as normal.

As mentioned above, the counter corresponding to the currently cached block (*ctr2*) is decremented. This is to ensure that the counter values will eventually decrease, so that after a transition to another phase of the



(a) Accessing *ctr1* (b) Accessing *ctr2*

Figure 5: MAT Operation.

program execution, new data can replace data that was accessed only in the previous phase. The rationale for decrementing counters on missing accesses to conflicting data is that the data currently residing in the cache must justify remaining cached in the face of heavy contention for that cache location. Therefore, the heavier the contention for a particular cache location, the more the cached data must be reused to maintain a counter large enough to satisfy Equation 1.

Rather than compare *ctr1* and *ctr2* exactly, in Equation 1 we compare *ctr1* to some fraction of *ctr2*. This is so we act conservatively when conflicting blocks have almost the same usage. Choosing *thresh* less than 1.0 will prevent bypassing when the counters are almost the same.

Another issue is that we do not want to bypass when the MAT contains no information for one of the macroblocks. In the case where there is no counter for the address being accessed this can be achieved by setting all bits to 1 in the register which holds *ctr1* for the comparison. When there is no counter found for the data residing in cache (*ctr2*), we compare to 0. Both of these are simply a matter of multiplexing in either the counter value read from the MAT or the appropriate constant, using the valid bit as a selector.

The second MAT access and the comparison are needed only during a cache miss to determine the data size requested, since only the element size needs to be fetched on a bypass. It is unlikely that both MAT accesses can be performed in one cycle, so this information will be available the cycle after the miss is detected. However, the size can be sent to the next level of the cache hierarchy the cycle after the address is sent, as the access to the L2 cache or main memory will take at least one cycle before the data can be returned. In some current processors the system bus request takes two cycles, with the address sent the first cycle and the data request size the second [12], which matches the MAT timing.

4.2 Improved MAT Scheme

One factor not yet taken into account is the fact that there can be some temporal locality even when the total accessing frequency is relatively low. In this case the basic MAT scheme presented in Section 4.1 will bypass some data which may have otherwise had a few hits before being displaced from the cache. More than one additional miss will be incurred by not caching that data, whereas only one miss is removed by not displacing the much more frequently accessed data. This will result in an overall increase in the miss ratio, and an overall performance degradation.

To avoid losing performance from the above scenario, we can place bypassing data in a small fully-associative buffer with short lines (corresponding to the element size that is fetched on a bypass), as shown in Figure 6. This buffer will be accessed in the same manner as a victim cache. As a result, the bypassed data is held close to the processor for a short time, allowing much of the temporal locality of the infrequently accessed data to be exploited.

The cost of the MAT hardware will be analyzed in Section 5.3, following the presentation of experimental results.

5 Experimental Evaluation

5.1 Experimental Environment

In this section, the environment used for experimental evaluation of our technique is presented. The applications for this study consist of several integer benchmark programs, that will be discussed below in Section 5.1.5. The experimental environment also includes compiler support, emulation to verify transformation correctness, and the simulation techniques used to generate experimental results.

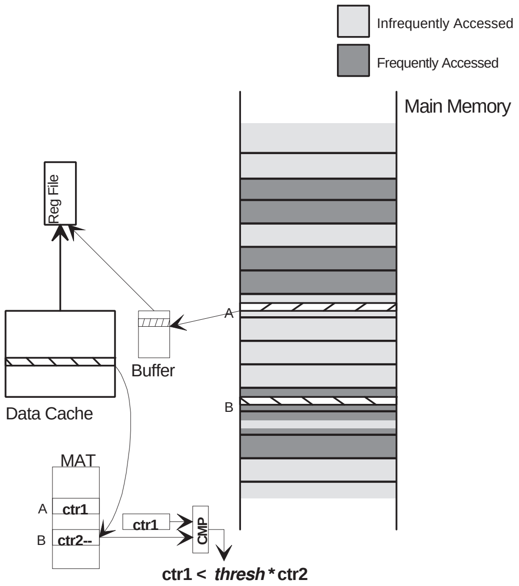


Figure 6: MAT Operation with a Buffer

Function	Latency	Function	Latency
Int ALU	1	FP ALU	2
memory load	2	FP multiply	2
memory store	1	FP divide (single prec.)	8
branch	1 + 1 slot	FP divide (double prec.)	15

Table 2: Instruction latencies for simulation experiments.

5.1.1 Compiler and Architecture

In order to provide a realistic evaluation of our technique, we first optimized the code using the machine-specific phases of the *IMPACT* compiler [13]. Classical optimizations were applied, then optimizations were performed which increase instruction level parallelism such as loop unrolling and superblock formation [14]. The code was scheduled, register allocated, and optimized for an eight-issue, scoreboarded, superscalar processor with register renaming. The Instruction Set Architecture supports compile-time speculation. Up to four memory accesses can be executed per cycle. The register file contains 64 integer registers and 64 double-precision floating-point registers.

5.1.2 Transformation Correctness Verification via Emulation

To verify the correctness of the code transformations, emulation of the target processor architecture was performed for all input programs on a Hewlett-Packard *PA-RISC 7100* workstation.

5.1.3 Simulation Parameters and Techniques

The emulator drives the simulator that models on a cycle-by-cycle basis the processor and the memory hierarchy (including all related busses) to determine application execution time, cache performance and bus utilization. The instruction latencies used are those of a Hewlett-Packard *PA-RISC 7100* microprocessor, as given in Table 2.

The memory hierarchy includes separate L1 instruction and data caches. The L1 instruction cache is a direct-mapped, 32K-byte split-block cache with a 64-byte block size. The L1 data cache is a direct-mapped, 16K-byte non-blocking cache with a 32-byte block size. The data cache is a multiported, write-back, no write-allocate cache that satisfies up to four load or store requests per cycle from the processor and has streaming support. Up to 50 load misses can be outstanding simultaneously on the bus connecting the L1 and L2 data caches. An 8-entry write buffer combines write requests to the same cache block. The instruction cache and data cache share a common, split-transaction L1-L2 bus, with a 4 cycle latency and 8 bytes/cycle data bandwidth. The memory hierarchy also includes a direct-mapped, 256K-byte non-blocking L2 data cache with a 64-byte block size. This cache is also write-back and no write allocate. Up to 50 load misses can be outstanding simultaneously from the L2 data cache on the system bus, which is split-transaction with a 50 cycle latency to memory and 8 bytes/cycle data bandwidth.

A direct-mapped branch target buffer with 1024 entries is used to perform dynamic branch prediction using a 2-bit counter. Hardware speculation is supported, and the branch misprediction penalty is approximately two cycles.

Since simulating the entire applications at this level of detail would be impractical, uniform sampling is used to reduce simulation time [15], however emulation is still performed between samples. The samples are 200,000 instructions in length and are spaced evenly every 20,000,000 instructions, yielding a 1% sampling ratio. For smaller applications, the time between samples is reduced to maintain at least 50 samples (10,000,000 instructions). From experience with the emulation-driven simulator, we have determined that sampling with at least 50 samples introduces typically less than 1% error in generated performance statistics.

5.1.4 Experimental Configurations

Three different configurations (in addition to the base configuration described above) were simulated. First, a traditional victim caching scheme was simulated, in order to compare the new methods presented in this paper with an existing method of reducing conflict misses. A 64-entry L1 victim cache and a 512-entry L2 victim cache are used in this configuration. The sizes we have chosen for the victim caches are rather large to make a fair comparison with our scheme using similar amounts of extra hardware.

The next configuration is the basic MAT scheme presented in Section 4.1. In this case two independent MATs are used, an L1 MAT and an L2 MAT. Each MAT operates independently of the other, so the L2 MAT only reflects the accesses that missed in the L1 cache and therefore accessed the L2 cache. On an L1 bypass eight bytes are fetched, while 32 bytes are fetched on an L2 bypass (we fetch the L1 block size because the L1 access is not guaranteed to bypass as well). Both MATs use the same value of *thresh*, and values of both 0.5 and 1.0 are investigated. Also, several different MAT sizes were simulated.

The third configuration is the improved MAT scheme presented in Section 4.2. The fully-associative buffers used to hold the bypassed data at the L1 and L2 caches contain 32 and 256 entries, respectively (half as many entries as the corresponding victim caches from the first configuration). The line sizes are also smaller than in the victim caches, because the victim cache must hold the entire cache block, while the L1 and L2 bypass buffers only need to hold the amount of data fetched on a bypass, which is the element size and the L1 cache block size, respectively.

We first present results for an infinite-entry MAT, then study the effects of limiting the number of entries in the MAT.

5.1.5 Benchmarks

Four benchmarks were simulated under each of the configurations from Section 5.1.4. The first, *026.compress*, is from the *SPEC'92* benchmark suite, and was discussed in detail in Section 3.1. The second benchmark, *099.go*, is from the *SPEC'95* benchmark suite. The other benchmarks from the *SPEC* benchmark suites had high hit ratios initially [5] and are not likely to benefit as much from data cache optimizations. Instead, the remaining benchmarks consist of modules from the IMPACT compiler. The first of these benchmarks, *lmdes2_customizer*, optimizes a machine description for efficient use by the IMPACT compiler. These optimizations operate over linked list and complex data structures, and utilize hash tables for efficient access to the information. The SuperSPARC machine description file is used as input to this benchmark. *Pcode*, the front end of IMPACT, is the second IMPACT benchmark and is run performing dependence analysis with the internal representation of the *combine.c* file from GNU CC as input.

[5] This has been noted previously for the *SPEC'92* benchmarks in published results [16]. Our own experiments verify this and also show that it is the case for many of the *SPEC'95* benchmarks.

5.2 Results

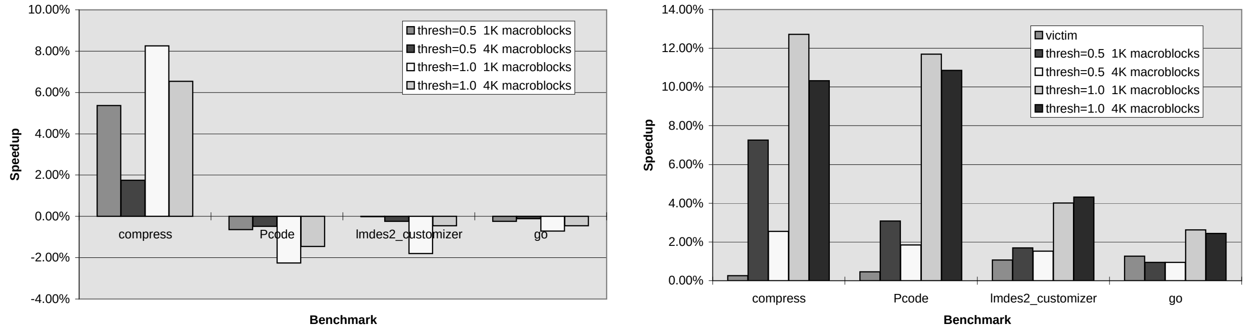
Figure 7a shows the speedup of each benchmark for the basic MAT scheme (no buffers) with an infinite-entry MAT. Compress improves under all *thresh* values and macroblock sizes. It achieves a performance gain of over 8%, with 1K macroblocks and a *thresh* of 1. This suggests that *compress* is amenable to agressive bypassing. However, all of the other benchmarks degrade in performance for all 4 configurations. This is due to the phenomenon noted in Section 4.2, wherein the small amounts of temporal locality that exist in the infrequently accessed macroblocks cannot be exploited after bypassing.

The improved MAT scheme, which places the bypassing data in a small buffer, yields much better results, as shown in Figure 7b. All benchmarks now achieve performance improvements for all of the configurations. Compress improves the most, yielding over 12% improvement for 1K macroblocks and a *thresh* of 1. The same values of macroblock size and *thresh* also give the best performance for *Pcode*, achieving nearly 12% improvement, and *go*, while *lmdes2_customizer* achieves slightly more improvement with 4K macroblocks. For all benchmarks a *thresh* of 1 achieves better results than a *thresh* of 0.5. The bypassing buffers allow this aggressive bypassing to occur, and these results suggest that the buffers hold the data for long enough in most cases to amend any incorrect bypassing decisions made by the higher *thresh* value.

Also shown in Figure 7b are the speedups attained by the victim caches discussed in Section 5.1.4. Although the victim caches are large, they attain smaller speedups than those obtained by the improved MAT scheme, except for *go* with a *thresh* of 0.5. However, all configurations with *thresh* values of 1 greatly outperform the victim caches. We saw in Section 3.1 that the hash table sizes used by *compress* were extremely large, and resulted in many conflict misses. This fact is underlined by the extremely small speedup achieved by the victim caches (0.26%). Thus, the combination of a 64-entry L1 victim cache and a 512-entry L2 victim cache is still too small to hold a significant portion of the conflicting data.

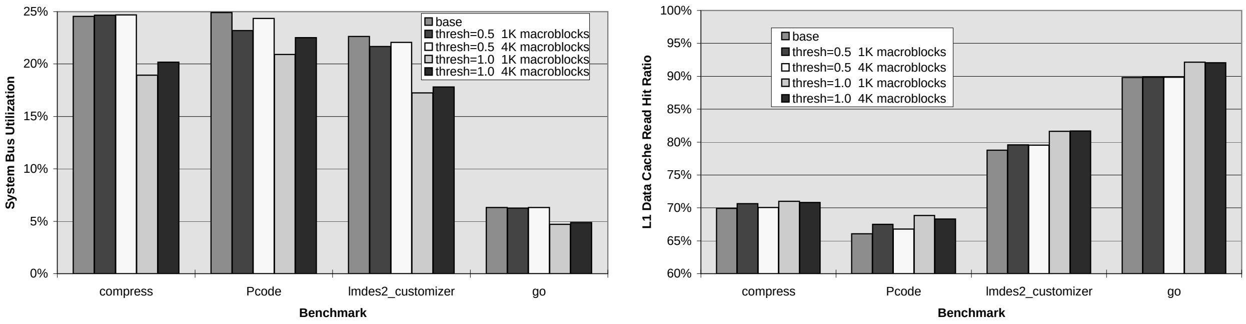
Figure 8a shows the utilization of the system bus (connecting the L2 cache with the main memory) for the improved MAT scheme with an infinite-entry MAT. The reduction in bus traffic is due to both the improvements in hit ratio, and to the reduced data request size for bypassing data.

The L1 data cache read hit ratios for the same configuration are shown in Figure 8b. Surprisingly, the hit ratios for *compress* improve the least, while *compress* achieved the largest speedup. From the memory access distributions of Figure 2 we see that the heavily accessed portions of htab and codetab alone total much more than 16K bytes, so although the MAT allows us to keep only heavily accessed data cached, different blocks from the heavily accessed regions will still replace each other frequently. However, this small



(a) Basic MAT (b) Improved MAT

Figure 7: Speedup with the Basic and Improved MAT Schemes (infinite-entry MAT).



(a) System Bus Utilization (b) L1 Data Cache Read Hit Ratio

Figure 8: Bus Utilization and Read Hit Ratio for the Improved MAT Scheme (infinite-entry MAT).

improvement in hit ratio and the reduced bus traffic from the smaller request size of the bypassing data together achieve a significant speedup. The improvements in the L2 data cache read hit ratio are slightly higher, but are more difficult to interpret, as the number of read requests reaching the L2 cache changes for each of the configurations.

However, the non-blocking caches can result in different amounts of effective memory latency seen by the processor, depending on how many miss requests are outstanding when an access occurs. Therefore, different misses stall the processor for different numbers of cycles, so the hit ratios may not be indicative of the overall performance. A more meaningful metric is the *average miss penalty*, or the average number of cycles the processor is stalled on the data cache per load access [6]. Figure 9 shows these values for the

[6]This is calculated by dividing the number of cycles the processor stalled on a use of outstanding data by the total number of load accesses

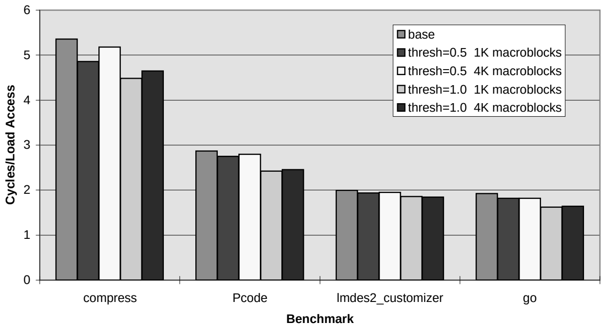
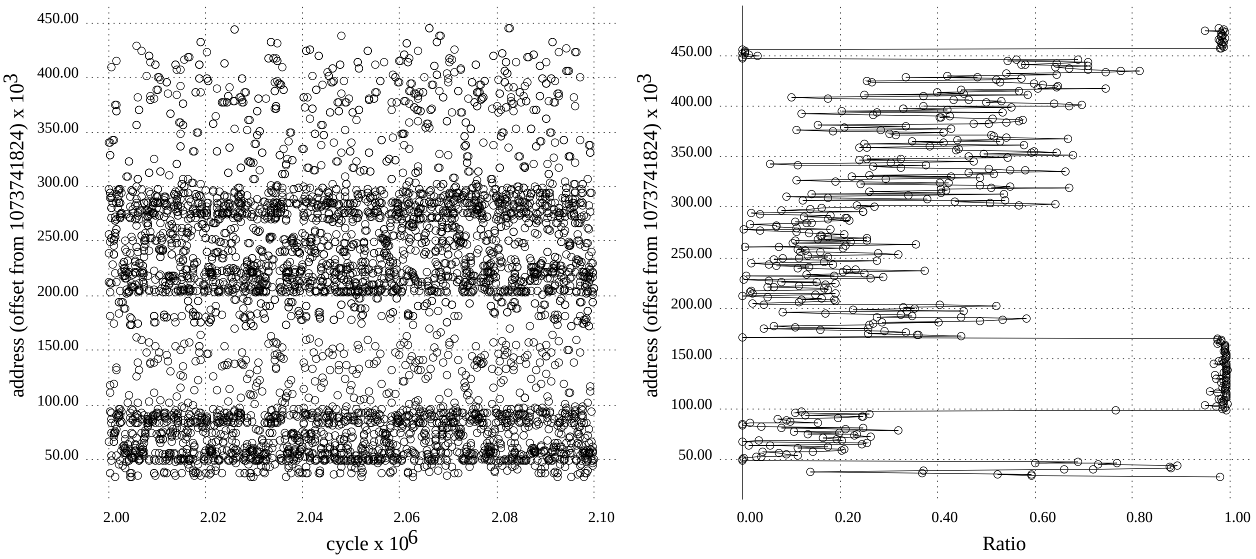


Figure 9: Average Miss Penalty per Load Access for the Improved MAT Scheme (infinite-entry MAT).

improved MAT scheme. The average miss penalties correspond better than the hit ratios with the speedup results shown in Figure 7b.

Figure 10 suggests that the MAT is doing a very good job of deciding which data to bypass for *compress*. Figure 10a shows the memory access distribution for both htab and codetab, for the same execution phase shown in Figure 2. Figure 10b has the same y-axis, memory addresses, and lines up with Figure 10a. The x-axis of Figure 10b is the ratio of times that a missing access was instructed by the MAT to bypass the cache. We can see that the very heavily accessed portions of the hash table bypass very rarely. The sparsely accessed portion of codetab, roughly between address offsets 100000 and 170000, bypasses almost 100% of the time. The sparsely accessed portion of htab bypasses about 50% of the time on average. As noted in Section 3.1, htab is accessed more often than codetab, so it has a less sparse access distribution than codetab and therefore bypasses less often.

To study the effects of a finite-size MAT we chose to simulate the improved MAT scheme with both 512 and 1K-entry direct-mapped MATs. These sizes were chosen because their hardware cost is reasonable, as will be discussed in Section 5.3, but yet they were large enough to hold most of the macroblocks for each of the benchmarks. The bypassing choices should be more conservative as the number of entries in the MAT is decreased, since we do not bypass unless both counters are found in the MAT, as discussed in Section 4.1. Table 3 shows the number of macroblocks accessed by each benchmark for both the 1K and 4K macroblock sizes. Even though *lmdes2_customizer* and *go* access less macroblocks than the number of entries in the MAT sizes we chose, there can still be some effects due to the direct-mapped associativity of the MAT and the fact that the macroblocks are not likely to be contiguous. This effect causes *lmdes2_customizer* to degrade slightly for a *thresh* of 1.0, as well as *Pcode* for the 1K-entry MAT and 4K-byte macroblocks.



(a) Memory Access Distribution (b) Bypassing Ratio

Figure 10: Compress Distribution and Bypassing Ratio Comparison (Improved MAT Scheme, infinite-entry MAT).

	Macroblocks Accessed	
Benchmark	1K Size	4K Size
026.compress	548	139
Pcode	2955	798
lmdes2_customizer	342	96
099.go	238	104

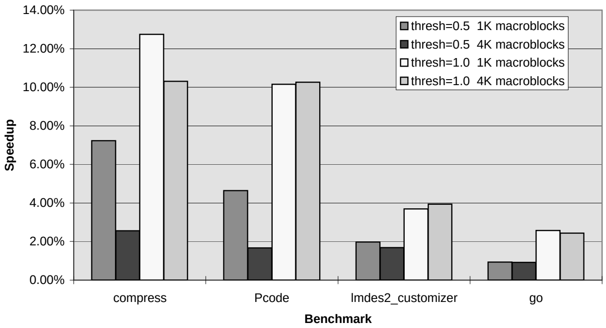
Table 3: Number of Macroblocks Accessed.

Pcode is also affected by the limited entries MAT for the configurations with more macroblocks than MAT entries. One anomaly is that several configurations have slight performance improvements after reducing the MAT entries. For example, *compress* with a *thresh* of 0.5 and 1K-byte macroblocks improves slightly when going from the 1K-entry MAT to a 512-entry MAT. This is probably due to the more conservative bypassing choices avoiding some incorrect bypasses. Several *Pcode* configurations improve similarly.

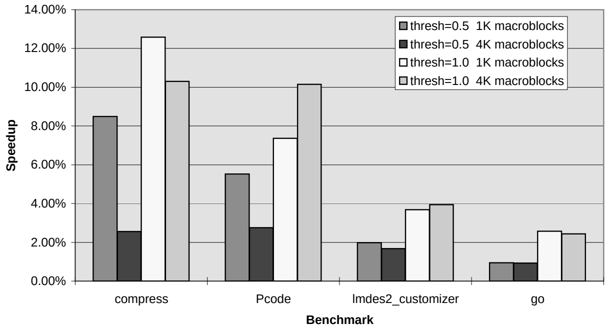
5.3 Design Considerations

The additional hardware cost incurred by the improved MAT scheme (MAT with buffers) is small compared to doubling the cache sizes at each level. This is particularly true for the L2 cache, but is also the case for the L1 cache (which uses the same size MAT).

For the 16K-byte direct-mapped L1 cache used to generate the results of Section 5.2, 18 bits of tag are used per entry (assuming 32-bit addresses). Doubling this cache will result in 17-bit tags. Because the line



(a) 1K-entry MAT



(b) 512-entry MAT

Figure 11: Speedup with the Improved MAT Schemes (1K and 512-entry MAT).

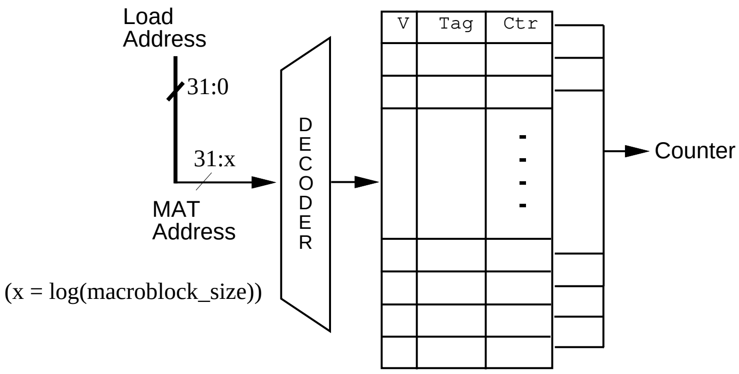


Figure 12: MAT Design.

size is 32 bytes, the total additional cost of the increased tag array will be $17 * 2^{10} - 18 * 2^9$, which is 1K bytes [7]. Besides the larger tag array cost, an extra 16K of data is needed.

For a direct-mapped MAT with 8-bit counters, Table 4 gives hardware cost of the data and tags for the MAT and macroblock sizes discussed in Section 5.2. Since all addresses within a macroblock map to the same MAT counter, a number of lower address bits are thrown away when accessing the MAT, as shown in Figure 12. The size of the resulting *MAT address*, used to access the MAT, is shown in column 4 of Table 4.

The cost for the L1 buffer, which is a fully-associative 32-entry cache with 8 byte lines, is 256 bytes of data and 96 bytes of tag (the tags will each be 24 bits). Although the tag store is fully-associative in design, it is small in size, which makes the total cost of the additional tag and data arrays for the improved MAT

[7] We are ignoring the valid bit and other state, which is conservative since the number of these bits per entry is the same or less in the MAT, and because the number of entries created when doubling the cache is larger than the number of entries in the MAT

MAT Entries	Data Cost (bytes)	Macroblock Size	Size of MAT Address (bits)	Tag Size (bits)	Tag Cost (bytes)
512	512	1K	22	13	832
		4K	20	11	704
1K	1K	1K	22	12	1536
		4K	20	10	1280

Table 4: Hardware Cost of 512 and 1K entry MATs.

scheme much less than that of increasing the data cache size.

A similar calculation will show that a 64-entry victim cache requires 2K bytes of data and 168 bytes of tag, which is slightly more than the total cost of the improved MAT scheme with a 512-entry MAT and a 32-entry bypass buffer, and slightly less than with a 1K-entry MAT and 32-entry bypass buffer. For a similar cost of tags and data, the results presented in Figures 7b and 11 show that the improved MAT scheme almost always performs much better than victim caching, especially for *compress* and *Pcode*.

To make the hardware cost even lower, we could potentially integrate the MAT with the TLB and page tables. For a macroblock size larger than or equal to the page size, each TLB entry will need to hold only one 8-bit counter value. For a macroblock size less than the page size, each TLB entry needs to hold several counters, one for each of the macroblocks within the corresponding page. In this case a small amount of additional hardware is necessary to select between the counter values. However, further study is needed to determine the full effects of TLB integration.

6 Conclusion

In this paper we presented a method to improve the efficiency of the caches in the memory hierarchy, by bypassing data that is expected to have little reuse in cache. This allows more frequently accessed data to remain cached longer, and therefore have a larger chance of reuse. The bypassing choices are made by a Memory Address Table (MAT), which performs dynamic reference analysis in a location-sensitive manner. Both the basic MAT scheme and an improved MAT scheme were investigated, where the latter places bypassing data in a small fully-associative buffer, allowing exploitation of small amounts of temporal locality which may exist in the bypassed data. We also introduced the concept of a macroblock, which allows the MAT to feasibly characterize the accessed memory locations.

Cycle-by-cycle simulations of several benchmarks show that significant speedups can be achieved by this technique. The speedups are due to the improved miss ratios and reduced bus traffic, which also result in a reduction in the average miss penalty per load. The improved MAT scheme was shown to outperform large victim caches, even for a finite size MAT of a similar hardware cost.

For future work we would like to find a larger set of benchmark programs with significant miss ratios so that we can perform further evaluations of our scheme. We will also work on more sophisticated MAT counter algorithms, beyond the simple reference count of this design. TLB integration is another area of future investigation, as mentioned earlier. In general, we believe that the schemes presented in this paper can be extended into a more general framework for runtime management of the cache hierarchy.

Acknowledgements

The authors would like to thank all the members of the IMPACT research group whose comments and suggestions helped to improve the quality of this research. The authors would also like to thank Santosh Abraham and Balas Natarajan at HP Labs for helpful discussions concerning the behavior of the compress benchmark.

This research has been supported by the National Science Foundation (NSF) under grant MIP-9308013, Intel Corporation, Advanced Micro Devices Hewlett-Packard, SUN Microsystems, NCR, and the National Aeronautics and Space Administration (NASA) under Contract NASA NAG 1-613 in cooperation with the Illinois Computer laboratory for Aerospace Systems and Software (ICLASS).

References

[1] K. Boland and A. Dollas, "Predicting and precluding problems with memory latency," *IEEE Micro*, pp. 59–66, August 1994.

[2] G. S. Sohi and M. Franklin, "High-bandwidth data memory systems for superscalar processors," in *Fourth International Conference on Architectural Support for Programm ing Languages and Operating Systems*, pp. 53–62, April 1991.

[3] A. J. Smith, "Cache memories," *Computing Surveys*, vol. 14, no. 3, pp. 473–530, 1982.

[4] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proceedings of the 17th International Symposium on Computer Architecture*, pp. 364–373, June 1990.

[5] J.-L. Baer and T.-F. Chen, "An effective on-chip preloading scheme to reduce data access penalty," in *Proceeding of Supercomputing '91*, pp. 176–186, Nov. 1991.

[6] T.-F. Chen and J.-L. Baer, "Reducing memory latency via non-blocking and prefetching caches," Tech. Rep. 92-06-03, Department of Computer Science and Engineering, University of Washington, Seattle, WA, June 1992.

[7] A. K. Porterfield, *Software Methods for Improvement of Cache Performance on Supercomputer Applications.* PhD thesis, Department of Computer Science, Rice University, Houston, TX, 1989.

[8] A. C. Klaiber and H. M. Levy, "An architecture for software-controlled data prefetching," in *Proc. 18th Ann. Int'l Symp. Computer Architecture*, (Toronto, Canada), pp. 43–53, May 1991.

[9] T. C. Mowry, M. S. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," in *Proc. Fifth Int'l Conf. on Architectural Support for Prog. Lang. and Operating Systems.*, pp. 62–73, Oct. 1992.

[10] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun, "A modified approach to data cache management," in *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pp. 93–103, December 1995.

[11] "SPEC newsletter," 1991. SPEC, Fremont, CA.

[12] Intel, *Pentium Pro Processor at 150 MHz, 166 MHz, 180 MHz and 200 MHz.* Intel Corporation, Santa Clara, CA, 1995.

[13] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "IMPACT: An architectural framework for multiple-instruction-issue processors," in *Proceedings of the 18th International Symposium on Computer Architecture*, pp. 266–275, May 1991.

[14] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The superblock: An effective structure for VLIW and superscalar compilation," tech. rep., Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, February 1992.

[15] J. W. C. Fu and J. H. Patel, "How to simulate 100 billion references cheaply," Tech. Rep. CRHC-91-30, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, 1991.

[16] J. D. Gee, M. D. Hill, D. N. Pnevmatikatos, and A. J. Smith, "Cache performance of the SPEC92 benchmark suite," *IEEE Micro*, pp. 17–26, August 1993.