Run-time Adaptive 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

Abstract

Improvements in main memory speeds have not kept pace with increasing processor clock frequency and improved exploitation of instruction-level parallelism. Consequently, the gap between processor and main memory performance is expected to grow, increasing the number of execution cycles spent waiting for memory accesses to complete. 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, 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 improvements in processor performance outpace that of main memory performance [1], the cache miss penalty will 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. The growing disparity between processor and memory performance will make cache misses increasingly 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. Additionally, data caches are not always used efficiently. In numeric programs there are several known compiler techniques 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.

Permission to make digital/hard copy of part or all this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
ISCA '97 Denver, CO, USA
© 1997 ACM 0-89791-901-7/97/0006...$3.50

In order to increase data cache effectiveness for integer programs we have investigated methods of *adaptive 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.

Our scheme seeks to manage the cache in a manner that is sensitive to the usage patterns of the memory locations accessed. Since the number of memory locations is excessively 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 and 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, attempting to hide 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][7] and software [8][9][10][11][12] prefetching methods for uniprocessor machines have been proposed. However, many of these methods focus on prefetching regular array accesses within well-structured loops[1], which are access patterns primarily found in numerical applications. There is also a great deal of prior work on prefetching in multiprocessors, but since their focus is even more on optimizing numerical applications, we will not review them here.

While these 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 both conflict and capacity misses, and their effects. Victim caches also attempt to reduce conflict miss effects in caches with low associativity [4]. While victim caches work well for some programs, as we will show in Section 5.2, they do not greatly improve programs that have large working sets.

Methods for both static and dynamic cache bypassing have also been investigated. In their pioneer work [13], Tyson *et al.* proposed a method where loads are marked for cache bypass either statically by the compiler, or dynamically at runtime. 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, and reuse behavior as a secondary 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 high 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, we designed a mechanism to determine whether to bypass based on the data address. As a result, we see an increase in cache hit ratios, a decrease in the bus traffic and a decrease in total cycle counts, while they achieve a decrease in the bus traffic at the expense of a small drop in the cache hit ratios, which may result in performance degradation.

Another study [14] presented a hardware bypassing mechanism based on reuse behavior of cached locations, and was proven effective for numeric programs. Their mechanism only performs bypassing of the first-level data cache, whereas we study bypassing of both cache levels. It is important to improve second-level cache performance, since system bus latencies are large. Also, their scheme marks a cache block for bypass permanently until the corresponding block is replaced from the second-level cache, at which time all reuse information is lost. We overcome this limitation by keeping track of reuse behavior in a separate structure, and by allowing bypass decisions to vary based on dynamic accessing behavior.

Some of our techniques, in particular the bypass buffer presented in Section 4.2, could be used in synergy with both of these schemes to obtain improved performance.

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 from the *SPEC92* benchmark suite [15]. Over 90% of *compress'* execution time is in this loop body. Many of the memory accesses in *compress* are to its hash tables, *htab* and *codetab* (the lines containing the hash table load accesses are numbered[2]). Due to the large hash table sizes (*htab* and *codetab* are roughly 270K and 135K bytes, respectively) 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.

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[3]. Also, the profiler kept track of reuse ratios[4]. 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 execution phase. The profiling results for a 100000-cycle sample of *compress* are shown in Figure 2. The memory access distribution for *htab* is shown in Figure 2a, where *htab* starts at address 171680 (all addresses are offsets from a base address of 1073741824, or 1G). 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. 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*.

The memory access distributions of Figure 2 illustrate the

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

[2]The other load accesses to *htab* in this loop can be eliminated through load elimination optimizations

[3]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.

[4]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, some of the hits will have reuse, as will some of the misses.

```
    while ( (c = getchar()) != EOF ) {
        in_count++;
        fcode = (long) (((long) c << maxbits) + ent);
        i = ((c << hshift) ^ ent);
1.      if ( htabof (i) == fcode ) {
2.          ent = codetabof (i);
            continue;
        } else if ( (long)htabof (i) < 0 ) goto nomatch;
        disp = hsize_reg - i;
        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

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). Two-way and four-way cache profiles exhibit similar behavior.

inherent problem with schemes that determine how to handle data based on the particular load instruction that requested the access [13]. 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 dynamic manner, must face the challenge of giving each dynamic instance of a load instruction different treatment. Otherwise, information is lost and some data will be mishandled.

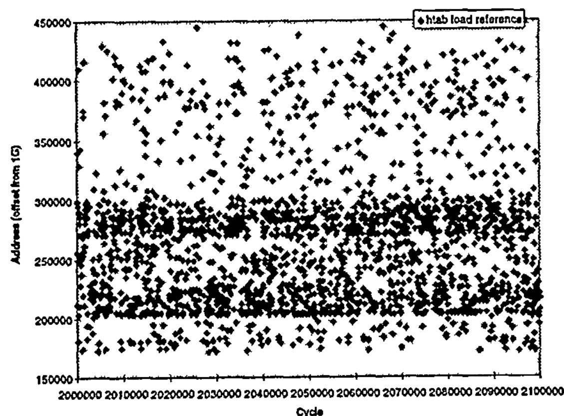
Figure 3 shows how we would like to handle accesses to data with different usage frequencies. In Figure 3a accesses to differently accessed regions of memory will map into the same cache lines, causing conflict or capacity misses. Assume that an access to a block in an infrequently accessed region of the memory misses in cache and that the 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 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. To perform this selective cache bypassing, we need some method of tracking the access behavior of different memory regions.

3.2 Macroblocks

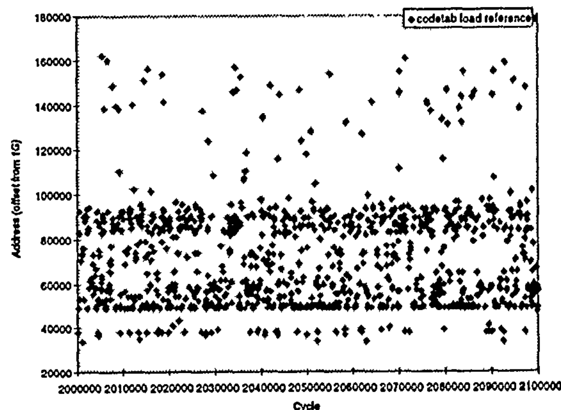
Ideally, we would like to keep track of the usage frequencies of all cache block size data in memory. While this would give us the most accurate information, it would result in an unmanageably large amount of information. Instead, we combine groups of adjacent cache block size data into larger blocks 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 excessively large, but small enough so that the accessing frequency of the cache blocks contained within each macroblock is relatively uniform. If we can keep track of each macroblock's accessing frequency through some hardware mechanism, then we can 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, the mean and standard deviation of the number of accesses to the cache blocks contained within each macroblock in memory were computed. 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 for *compress*. The y-axis is a log_{10} scale of the standard deviation divided by the mean (in order to normalize the results), 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, but the 16K-byte curve does not look quite as good. The curves for other benchmarks have similar characteristics, and will not be presented here due to space constraints. A large number of simulations confirm that 1K-byte macroblocks provide the best cost-performance tradeoff. Therefore, we chose 1K-byte macroblocks for all experiments presented in this paper.

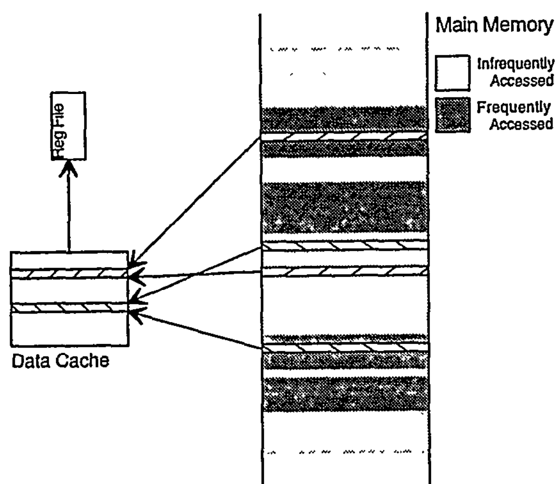


(a) htab

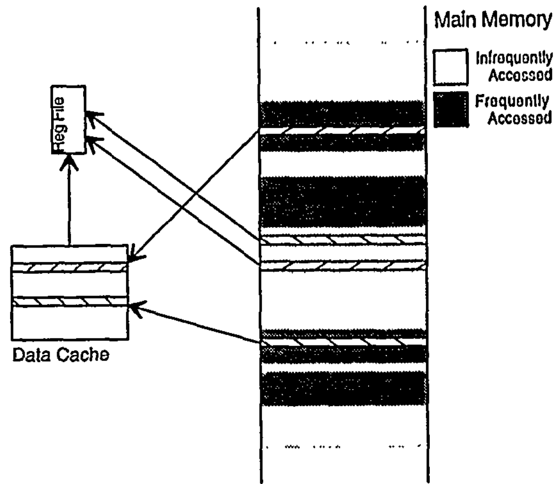


(b) codetab

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.

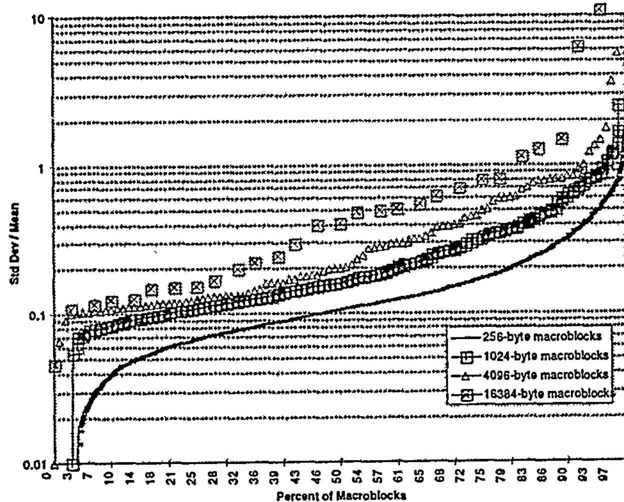


Figure 4: Macroblock Access Uniformity

4 Hardware

4.1 Memory Address Table

In order to keep track of the macroblocks at run time 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 contains a saturating counter, where the counter value represents the frequency of accesses to the corresponding macroblock.

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 is allocated and initialized, as will be discussed later. 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 data in macroblock A is accessed.

If the data cache access resulted in a hit, the access proceeds as normal, and the counter value is ignored. If the access resulted in a cache miss, the cache controller must look up the MAT counter corresponding to the cache block that

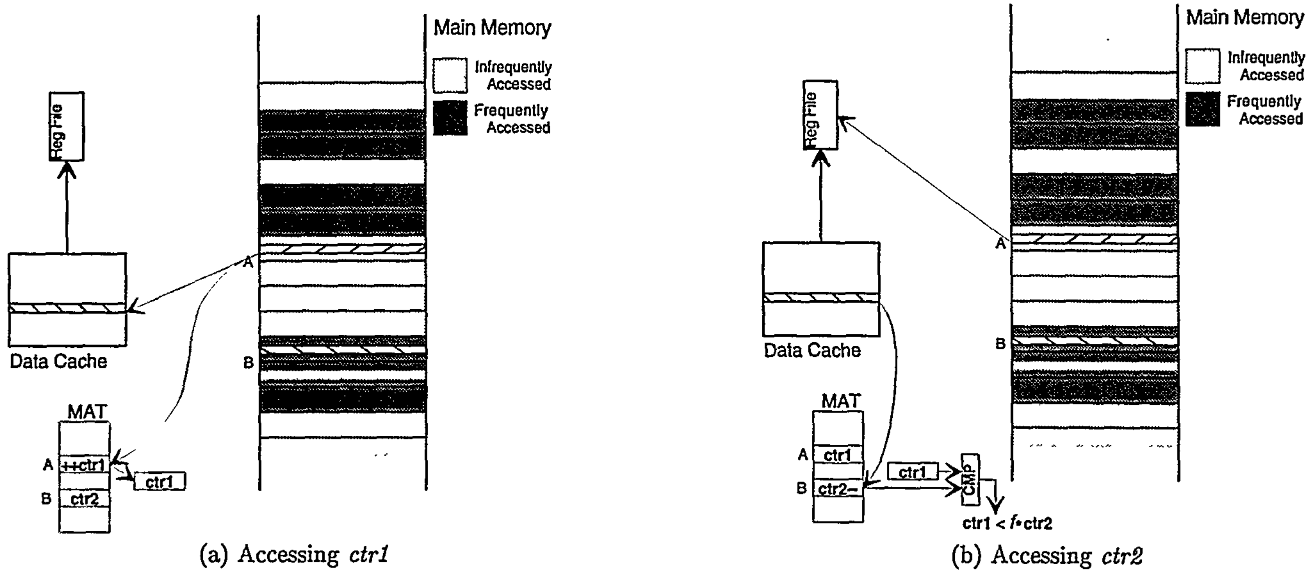


Figure 5: MAT Operation.

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 where data in macroblock B would be replaced. This counter value (*ctr2*) is then decremented and compared to the counter value corresponding to the missing access. The actual comparison performed is:

$$ctr1 < f * ctr2 \quad (1)$$

where *f* is some fraction. If the above inequality is satisfied the fetched data will bypass the cache. Otherwise, it replaces 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 program execution, new data can replace data that is now unused. The rationale for decrementing counters on missing accesses to conflicting data is that the data currently residing in the cache must justify remaining cached when there is 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 f of *ctr2*. Choosing f less than 1.0 will conservatively prevent bypassing when the counter values are almost the same.

Also, we do not want to bypass when the MAT contains no entry for one of the macroblocks, so that the cache will then default to its normal replacement behavior when there is insufficient information. 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.

When a new entry is allocated, the counter value must be initialized to some value. One possibility is to initialize it to 0. This will result in more aggressive bypassing, since it will take longer for the new counter to "catch up" to the older cached data's counter. Another possibility is to initialize to the conflicting data's counter value, as shown in Figure 6a. The new counter will start at an equal position, resulting in more selective bypassing. Simulations verified that initialization to 0 is optimal for an L1 data cache MAT, since the L1-L2 latency is small. The small latency means that the shorter bypass fetch size can be just as important as reducing the number of fetches, and that there will be less penalty when the decision to bypass was incorrect. Simulations also verified that initialization of *ctr1* with *ctr2* is optimal for an L2 data cache MAT, since the system bus latency is long. In this case reducing the length of data transferred is not critical, and although bypassing is still beneficial, it is much more important to make very careful bypassing decisions.

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[5] 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 [16], which matches the MAT timing.

4.2 Temporal Locality in Bypassed Data

Another issue is that there can be some temporal locality even when the total accessing frequency is relatively low. In

[5] In this paper the *element size* refers to the maximum element size allowed by the ISA, which in our system is 8 bytes.

this case the MAT scheme 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.

To optimize performance in this situation, we place bypassing data in a small set-associative buffer, as shown in Figure 6b. This bypass 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.

4.3 Spatial Locality in Bypassed Data

Spatial locality may also exist in the bypassed data. It is only necessary to fetch the element size on a bypass, which can be effective in reducing the bus transfer time. However, simulations show that when simulating the MAT scheme with a small bypass fetch size, around 15% of the L1 data cache misses in several benchmarks are a result of the lost spatial reuse opportunities when fetching only the element on a bypass. These misses are accesses to another element in the same cache block as some element in the bypass buffer. Fetching the entire cache block on a bypass would eliminate these misses.

However, fetching the entire cache block for benchmarks with little spatial locality in bypassed data will unnecessarily increase bus traffic. Therefore, it is beneficial to implement some means of dynamically choosing the optimal bypass fetch size. The choice of bypass fetch size should depend on the spatial locality detected, and can be determined at the macroblock granularity.

This optimization could be implemented using subblock support in the bypass buffer. However, subblocks can result in underutilized lines, when only the element size is fetched. Instead, we use a bypass buffer with small lines, equal to the element size, and optionally fill in consecutive blocks when the larger bypass fetch size is chosen. This approach is similar to that used in some prefetching strategies [17].

To facilitate spatial locality detection, a new counter, *sctr*, will be added to each macroblock entry. In addition, an *sr* (*spatial reuse*) bit and an *fi* (*fetch initiator*) bit are added to each bypass buffer tag. When a new entry is allocated in the bypass buffer, its *sr* bit is reset to 0. On a miss in the bypass buffer, if the upper bits of another tag indicate that data from the same full L1 data cache block resides in the bypass buffer, then the missing data's *sr* bit is set and the corresponding macroblock's *sctr* is incremented. However, in a set-associative bypass buffer we may need to access additional sets, or a separate structure which tracks this information. If the entire cache block was fetched into the bypass buffer, the *sr* bit is set on a hit access to any element other than the element which caused the block to be loaded. This is implemented by setting the *fi* bit to 1 during the cache refill for the bypass buffer block which contains the missing element, otherwise resetting it to 0. Then the *sr* bit is set to 1 on a hit to a block with a *fi* bit of 0. When the bypass buffer block is replaced, the corresponding macroblock's *sctr* is decremented if the *sr* bit is not set for any of the blocks in the same full L1 data cache block.

On a cache bypass, the full cache block is fetched if the *sctr* is saturated, otherwise the element size is fetched. Simulations showed that a 1-bit *sctr* is optimal.

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

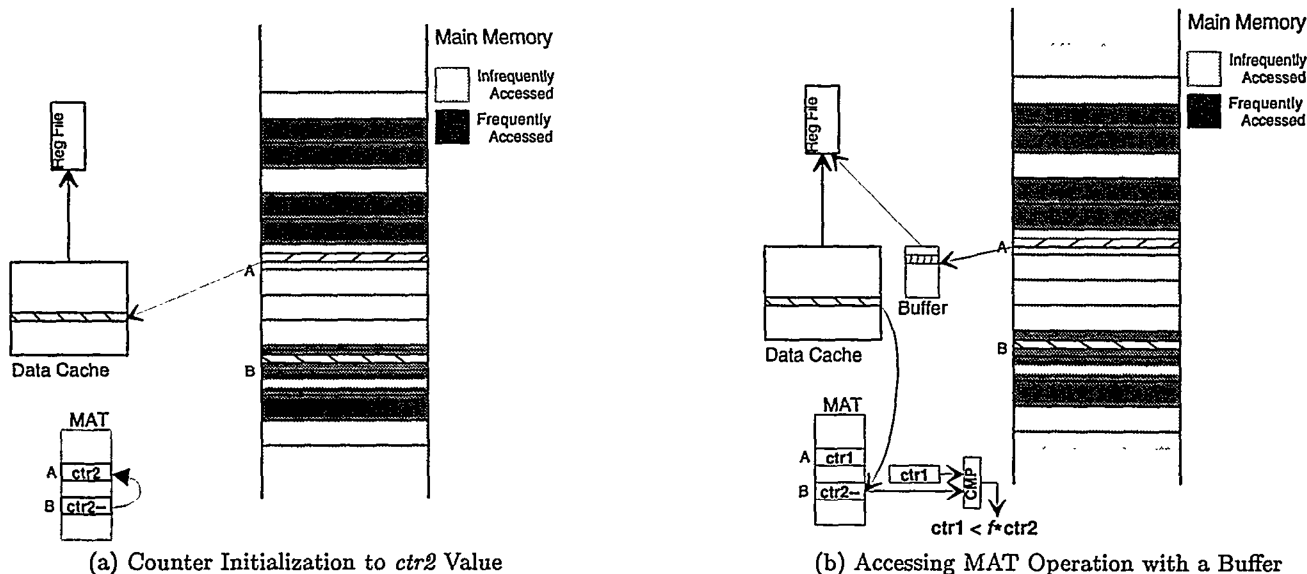
In this section, the environment used for experimental evaluation of our technique is presented. The applications for this study consist of several integer benchmark programs, that will be discussed below in Section 5.1.1. The experimental environment also includes compiler support, emulation to verify transformation correctness, and the simulation techniques used to generate experimental results.

5.1.1 Benchmarks

Ten benchmarks were simulated under each of the configurations from Section 5.1.5. The first, *026.compress*, is from the *SPEC92* benchmark suite, and was discussed in detail in Section 3.1. *072.sc* and *085.cc1*, also from the *SPEC92* benchmark suite, are simulated. *099.go*, *147.vortex*, *130.li* and *134.perl*, from the *SPEC95* benchmark suite, are simulated using the training inputs. The next two benchmarks consist of modules from the *IMPACT* compiler [18] that we felt were representative of many real-world integer applications. *Pcode*, the front end of *IMPACT*, is run performing dependence analysis with the internal representation of the *combine.c* file from GNU CC as input. *lmdes2_customizer*, a machine description optimizer, is run optimizing the SuperSPARC 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 last benchmark is the *java* interpreter. It is run interpreting the compress benchmark which was hand-translated to java code, with the *SPEC92* compress input. The experimental environment for *java* is slightly different from that of the other benchmarks, and will be discussed in Section 5.1.4.

5.1.2 Compiler and Architecture

In order to provide a realistic evaluation of our technique, we first optimized the code using the *IMPACT* compiler [18]. Classical optimizations were applied, then optimizations were performed which increase instruction level parallelism such as loop unrolling and superblock formation [19]. The code was scheduled, register allocated and optimized for an eight-issue, scoreboarded, superscalar processor with register renaming. The Instruction Set Architecture is an extension of the HP PA-RISC instruction set to support 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.



(a) Counter Initialization to *ctr2* Value

(b) Accessing MAT Operation with a Buffer

Figure 6: MAT Operation Details.

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.

5.1.3 Simulation Parameters and Techniques

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.

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[6], 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 [20], however emulation is still performed between samples. The simulated 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.

The current versions of HPUX sometimes allocate the stack frame differently in memory across runs of the same program, which can cause the cache performance to vary. This effect is most noticeable when running the same executable on different machines. To obtain the typical performance benefit of our techniques, we emulated each configuration for every benchmark on three different HP workstations and recorded the median speedup. However, the speedups generally varied less than one percent.

5.1.4 Java

The *java* benchmark was simulated on a Pentium system running Linux, for which we do not yet support virtual machine emulation. Therefore, *java* was optimized for a Pentium machine [16], and x86 machine code was generated and run to ensure correctness, as well as to drive the simulator. The machine model simulated is that of the Pentium, rather than the virtual machine described in Section 5.1.2, except for the memory system which is identical to that explained in Section 5.1.3.

[6] Initially both data caches are direct-mapped, but in a later section we examine the affects of set-associative data caches.

5.1.5 Experimental Configurations

The first set of configurations use the MAT scheme presented in Section 4.1. In this scheme, 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. Both MATs use the same value of f and 1K-byte macroblocks. After a large number of simulations[7] $f = 1.0$ was shown to have the best performance across all the benchmarks, and will be used throughout the rest of this paper. The bypass buffer allows aggressive bypassing from $f = 1.0$ to occur without much effect from any incorrect bypassing decisions made by the higher f value.

The 4-way set-associative buffers used to hold the bypassed data at the L1 and L2 caches contain 32 and 256 entries, respectively, in the first configuration. We fetch the L1 block size on an L2 bypass because the L1 access may not have been a bypass. The next several experimental configurations explore the effects of varying the bypass fetch sizes. Unless noted otherwise, the bypass buffer line sizes will be the maximum bypass fetch size used.

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

5.2 Results

Figure 7a shows the speedup of each benchmark for the MAT scheme, which places the bypassing data in a small buffer, using an infinite MAT with 1K macroblocks and $f = 1$. The leftmost speedup bar for each benchmark, labeled (8,32) to denote an 8-byte L1 bypass fetch size and a 32-byte L2 bypass fetch size, fetches the minimum data necessary on a bypass. Most benchmarks achieve good performance improvement, with *compress* and *Pcode* improving the most, yielding 12% improvement each.

The next three configurations shown in Figure 7a consist of the various combinations of fetching either the minimum size or the cache block size on a bypass. Several observations can be made from this figure. First, all benchmarks perform better using an L2 bypass fetch size of 64, or the entire L2 cache block, compared to the minimum fetch size of 32. This is due to the long system bus latency (50 cycles), which means that the time necessary to transfer the 32 or 64 bytes is small in comparison to the latency of accessing main memory. Therefore, it is better to fetch more data, and reduce the likelihood of needing another fetch for the other data in the cache block.

Comparing L1 bypass fetch sizes of 8 and 32, the trend is not as clear. Some benchmarks (*026.compress*, *072.sc*, *Pcode*) perform better when fetching only the element on an L1 bypass, due to an overall lack of spatial locality in these benchmarks. Other benchmarks, such as *134.perl*, perform much better when fetching the full 32-byte L1 cache block on a bypass. Because the L1-L2 latency is much shorter, the data transfer time is very close to the L2 access latency, and it is not as clear which aspect to optimize. Therefore, the spatial reuse optimization discussed in Section 4.3 was simulated, with results shown by the rightmost speedup bar for each benchmark in Figure 7a. In this case the number of entries in the L1 bypass buffer was increased to 128. However, due to the smaller line size of 8 bytes, the size of the bypass buffer is the same as the 32-entry buffer with a bypass fetch size of 32 bytes. The benchmarks achieve speedups better than, or very close to, the best speedup achieved by the other configurations.

[7]Due to space constraints these results will not be shown here.

Figure 7b shows the utilization of the system bus (connecting the L2 cache with the main memory) for the MAT scheme with an infinite number of entries. Only the 64-byte L2 bypass fetch size configurations are shown. The reduction in system bus traffic is due to the improvements in hit ratio.

The L1 data cache read hit ratios for the same configurations are shown in Figure 7c. 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.

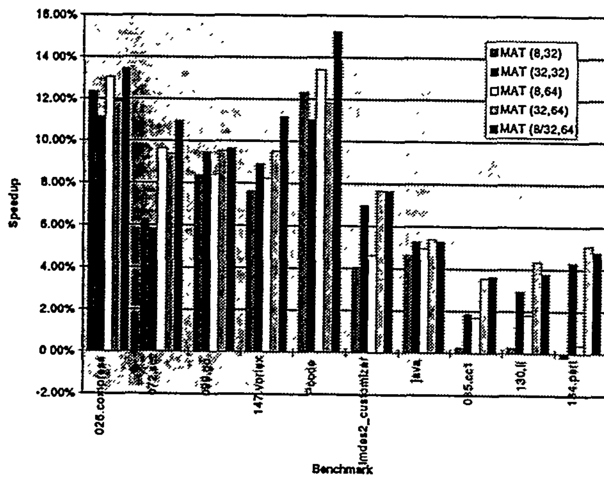
Non-blocking caches can result in different amounts of effective memory latency seen by the processor, depending on how many miss requests are outstanding when an access occurs. Therefore, the hit ratios may not be indicative of the overall performance. A more meaningful metric is the *average miss penalty*, or the average number of cycles the processor is stalled on the data cache per load access. Figure 7d shows that the average miss penalties correspond better than the hit ratios with the speedup results shown in Figure 7a.

Figure 8 suggests that the MAT is correctly deciding which data to bypass for *compress*. Figure 8a shows the memory access distribution for both htab and codetab, for the same execution phase shown in Figure 2. Figure 8b has the same y-axis, and lines up with Figure 8a. The x-axis of Figure 8b is the ratio of times that a missing access was instructed by the MAT to bypass the cache. Figure 8 shows 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.

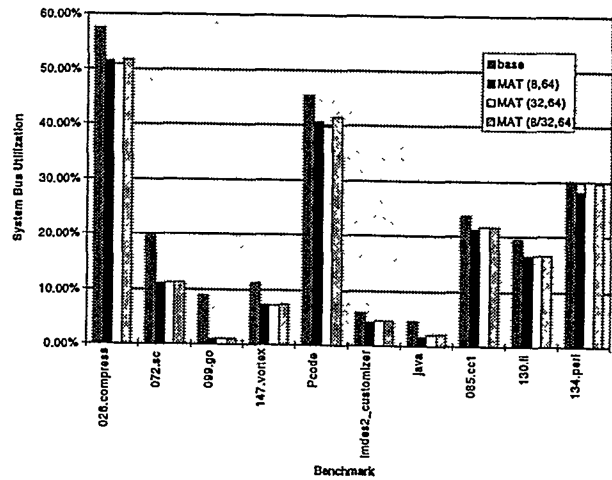
5.2.1 Set-associative Data Caches

Increasing the set-associativity of the data caches can reduce the number of conflict misses, which may in turn reduce the advantage offered by the MAT. To investigate these effects, the data cache configuration discussed in Section 5.1.3 was slightly modified to have a 2-way set-associative L1 data cache and a 4-way set-associative L2 data cache.

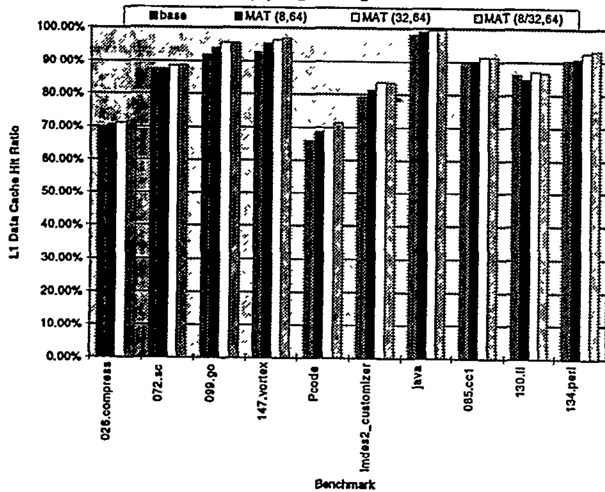
Figure 9 shows the new speedup for two MAT configurations. The first configuration is the same shown in the previous section, with a 64-byte L2 bypass fetch size and the dynamically varying L1 bypass fetch size. Because the higher associativity filters out some of the conflict misses that were bypassed in the case of direct-mapped caches, it may be beneficial to bypass less aggressively. One way in which to bypass less frequently is to have fewer bits in the MAT access



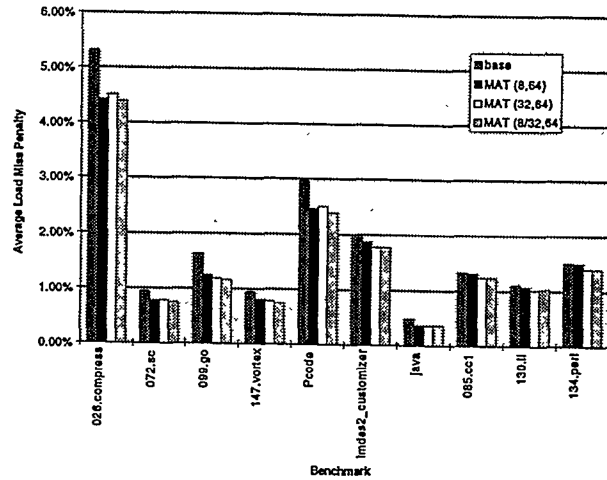
(a) Speedup



(b) System Bus Utilization

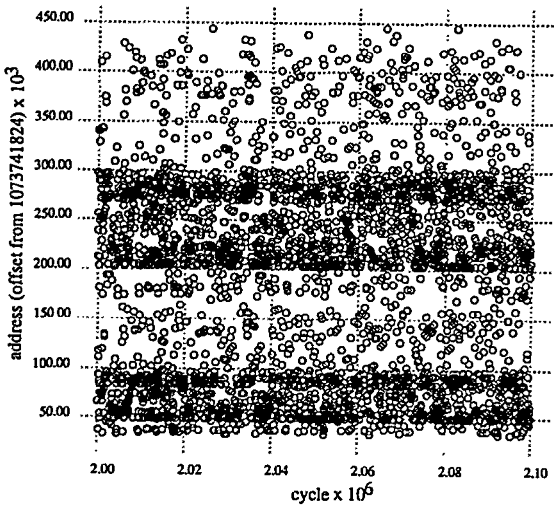


(c) L1 Data Cache Read Hit Ratio

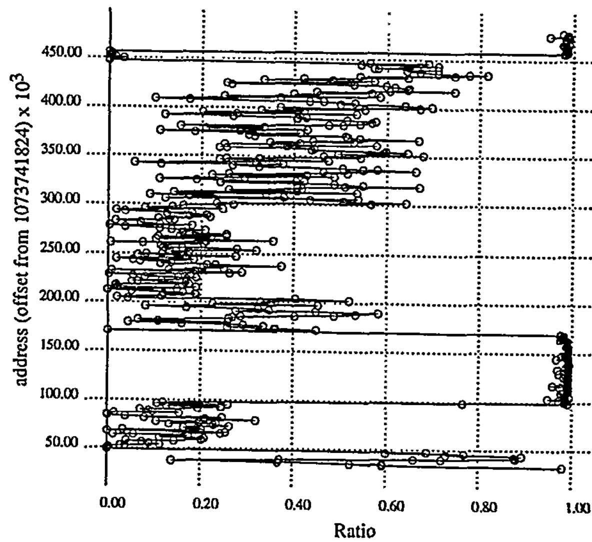


(d) Average Miss Penalty per Load Access

Figure 7: Performance data for the MAT Scheme (infinite-entry MAT). The legend denotes (L1 bypass fetch size, L2 bypass fetch size), where an L1 bypass fetch size of 8/32 denotes the dynamically varying scheme.



(a) Memory Access Distribution



(b) Bypassing Ratio

Figure 8: Compress Distribution and Bypassing Ratio Comparison (MAT Scheme, infinite-entry MAT).

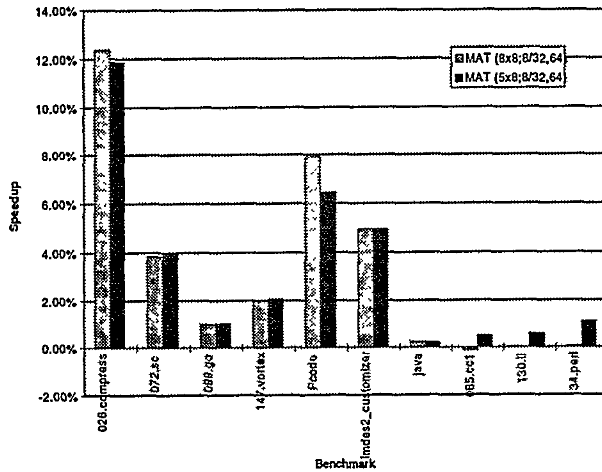


Figure 9: Speedup for 2-way and 4-way set-associative L1 and L2 data caches, respectively, with the MAT Schemes using dynamically-varying L1 bypass fetch sizes (infinite-entry MAT) and various numbers of bits per L1 MAT *ctr*. The legend denotes ({bits/L1 MAT *ctr*}x{bits/L2 MAT *ctr*}; L1 bypass fetch size, L2 bypass fetch size), where an L1 bypass fetch size of 8/32 denotes the dynamically varying scheme.

counter. Because the counters will saturate faster, and at a lower value, it is less likely that one macroblock will have a much larger *ctr* value for very long. As a simple preliminary investigation we modified the previous configuration so the L1 MAT had a 5-bit (rather than 8-bit) *ctr*. This configuration, denoted (5x8;8/32,64) to reflect the 5-bit L1 MAT *ctr* and 8-bit L2 MAT *ctr*, is also shown in Figure 9.

As noted in Section 3.1, *compress* has very large hash tables, resulting in many capacity misses. For this reason, its speedup using a MAT decreases little when increasing the data cache associativity, seen by comparing Figure 9 to Figure 7a. Other benchmarks decrease in varying amounts. However, using a 5-bit L1 MAT *ctr* the MAT still achieves speedup for all benchmarks. With increasing memory latency these speedups are likely to increase.

5.2.2 Finite-size MAT

To study the effects of a finite-size MAT we chose to simulate the 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, 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 a 1K-byte macroblock size. Figure 10 shows speedups achieved by the MAT scheme for direct-mapped data caches using differing numbers of entries per MAT. Some benchmarks degrade slightly when the entries per MAT are decreased. One anomaly is that several benchmarks have slight performance improvements from reducing the MAT entries. This is due to the more conservative bypassing choices avoiding some incorrect bypasses. In general, the performance effects of

	Macroblocks Accessed
026.compress	548
072.sc	321
099.go	238
147.vortex	2275
Pcode	2955
lmdes2_customizer	342
java	527
085.ccl	1864
130.li	670
134.perl	4581

Table 3: Number of Macroblocks Accessed.

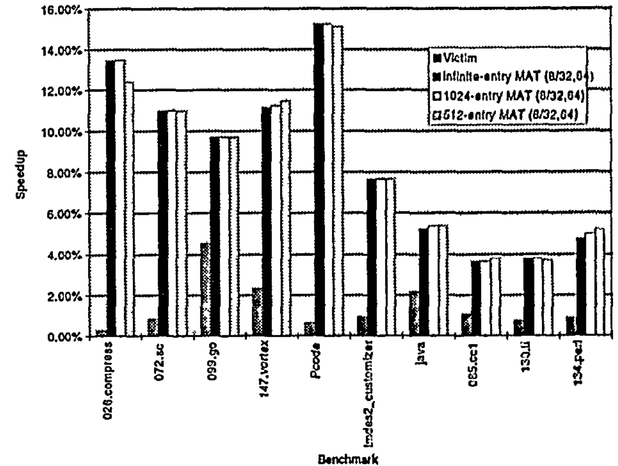


Figure 10: Speedup for the MAT Scheme with infinite, 1024 and 512-entries per MAT. All have 4-way set-associative bypass buffers and dynamically-varying L1 and 64-byte L2 bypass fetch sizes. The L1 and L2 fully-associative victim caches have 64 and 512 entries, respectively.

restricting the number of MAT entries is small.

Also shown in Figure 10 are the speedups attained by the traditional victim caches. A 64-entry fully-associative L1 victim cache and a 512-entry fully-associative L2 victim cache are used in this configuration. Although the victim caches are large and have full associativity, they attain smaller speedups than those obtained by the MAT scheme with 4-way set-associative bypass buffers, using similar amounts of hardware, as will be shown in Section 5.3. The fact that *compress* uses large hash tables, resulting in many capacity misses, 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.

5.2.3 Growing Memory Latency Effects

As discussed in Section 1, memory latencies are increasing, and this trend is expected to continue. Figure 11 shows the speedups of the MAT scheme applied to direct-mapped data caches for three different latencies, the 50-cycle latency used in all previous configurations, as well as 100-cycle and 200-cycle latencies. Most of the benchmarks see much higher performance improvements from the MAT scheme when the memory latencies increase, with speedups close to 25% for *072.sc* and *099.go* for a 200-cycle memory latency. This is to be expected, as the number of cycles spent waiting for out-

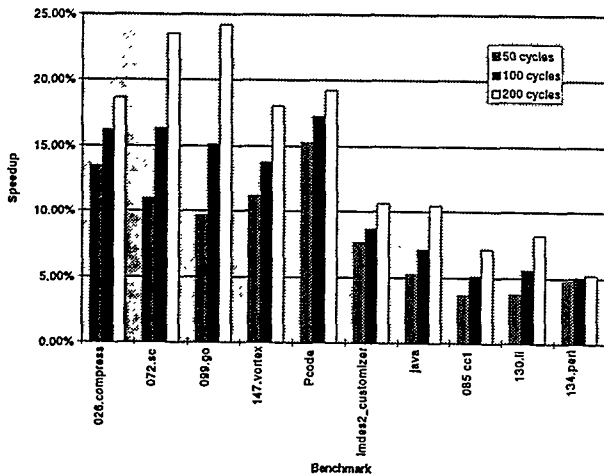


Figure 11: Speedup for the MAT Scheme with varying memory latency. All have dynamically-varying L1 and 64-byte L2 bypass fetch sizes.

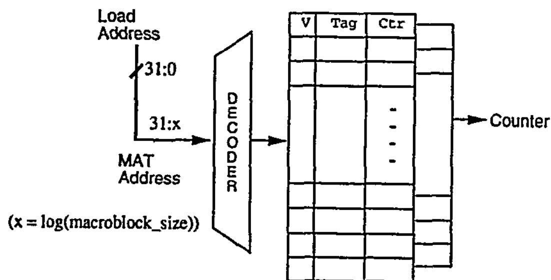


Figure 12: MAT Design.

standing cache misses will increase, and optimizations which decrease the number of cache misses will become increasingly important.

5.3 Design Considerations

The additional hardware cost incurred by the MAT scheme is small compared to doubling the cache sizes at each level, particularly for the L2 cache. 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 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[8]. In addition, an extra 16K of data is needed.

For a direct-mapped MAT with 8-bit access counters and 1-bit spatial counters, Table 4 gives the 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 3 of Table 4.

[8]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)	Size of MAT Address (bits)	Tag Size (bits)	Tag Cost (bytes)
512	576	22	13	832
1K	1152	22	12	1536

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

Entries	Bypass Fetch Size (bytes)	Block Size (bytes)	Data Cost (bytes)	Tag Size (bits)	Tag Cost (bytes)
32	8	8	256	26	104
	32	32	1024	24	96
128	8/32	8	1024	26	416

Table 5: Hardware Cost of L1 Bypass Buffer.

Entries	Block Size (bytes)	Data Cost (bytes)	Tag Size (bits)	Tag Cost (bytes)
64	32	2048	27	216

Table 6: Hardware Cost of L1 Victim Cache.

The cost for the L1 buffer, which is a 4-way set-associative cache with 8 or 32 byte lines, depending on the bypass fetch size chosen, is shown in Table 5. The total cost of the additional tag and data arrays for the MAT scheme is much less than that of increasing the data cache size. If the varying bypass size scheme of Section 4.3 is used, then spatial locality detection support is necessary in the L1 bypass buffer, which will require a small amount of additional hardware. This is reflected by the 8/32 entry in Table 5, where an extra 2 bits of tag are needed per entry (1-bit *spatial reuse bit* and 1-bit *fetch initiator bit*).

Table 6 shows the costs for a 64-entry L1 victim cache. Adding the costs from Tables 4 and 5 and comparing to Table 6 shows that the MAT configurations are only slightly more expensive in tag and data costs. However, the victim caches are fully-associative, which may make the MAT scheme with 4-way set-associative buffers less costly overall. For a similar cost of tags and data, the results presented in Figure 10 show that the MAT scheme always performs better than victim caching with higher associativity.

Similar calculations will show that all L2 MAT configurations are much less expensive than the L2 victim cache configuration used to generate the results of Section 5.2.

To reduce the hardware cost, we could potentially integrate the L1 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. An MAT scheme was investigated, which places bypassing data in a small 4-way set-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.

Variations in the amount of data fetched on a bypass were investigated, including a dynamically-varying bypass fetch size. Additionally, the effects of increasing the data cache associativities and reducing the number of MAT entries were examined.

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 MAT scheme was shown to outperform large victim caches, even for a finite-size MAT of a similar hardware cost and less associativity. In addition, we showed that the speedups achieved by the MAT scheme increase as the memory latency increases.

For future work we will examine more sophisticated MAT counter algorithms, beyond the simple reference count of this design, as well as adaptive cache remapping schemes. 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 intelligent 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, in particular Dave August for proofreading various drafts of this paper. 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. We would also like to thank the anonymous referees for their constructive comments.

This research has been supported by the National Science Foundation (NSF) under grant CCR-9629948, 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] S. Mehrotra and L. Harrison, "Quantifying the performance potential of a data prefetch mechanism for pointer-intensive and numeric programs," Tech. Rep. 1458, CSRD, Univ. of Illinois, November 1995.

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

[9] 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.

[10] 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.

[11] W. Y. Chen, S. A. Mahlke, P. P. Chang, and W. W. Hwu, "Data access microarchitectures for superscalar processors with compiler-assisted data prefetching," in *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pp. 69–73, November 1991.

[12] W. Y. Chen, S. A. Mahlke, W. W. Hwu, T. Kiyohara, and P. P. Chang, "Tolerating data access latency with register preloading," in *Proceedings of the 6th International Conference on Supercomputing*, July 1992.

[13] 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.

[14] J. A. Rivers and E. S. Davidson, "Reducing conflicts in direct-mapped caches with a temporality-based design," in *Proceedings of the 1996 International Conference on Parallel Processing*, pp. 151–162, August 1996.

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

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

[17] J. W. C. Fu and J. H. Patel, "Data prefetching in multiprocessor vector cache memories," in *Proc. 18th Ann. Int'l Symp. Computer Architecture*, (Toronto, Canada), pp. 54–63, June 1991.

[18] 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.

[19] 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.

[20] 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.