Run-time Cache Bypassing

Teresa L. Johnson[†*] Daniel A. Connors[‡] Matthew C. Merten[‡] Wen-mei W. Hwu[‡]

[†]Hewlett-Packard Company
11000 Wolfe Rd, MS42-U5
Cupertino, CA 95014
teresa@cup.hp.com

[‡]Center for Reliable and High-Performance Computing
University of Illinois
Urbana-Champaign, IL 61801
{dconnors,merten,hwu}@crhc.uiuc.edu

January 31, 2000

Abstract

The growing disparity between processor and memory performance has made cache misses increasingly expensive. Additionally, data and instruction caches are not always used efficiently, resulting in large numbers of cache misses. Therefore, the importance of cache performance improvements at each level of the memory hierarchy will continue to grow. In numeric programs there are several known compiler techniques for optimizing data cache performance. However, integer (non-numeric) programs often have irregular access patterns that are more difficult for the compiler to optimize. In the past, cache management techniques such as cache bypassing were implemented manually at the machine-language-programming level. As the available chip area grows, it makes sense to spend more resources to allow intelligent control over the cache management.

In this paper we present an approach to improving cache effectiveness, taking advantage of the growing chip area, utilizing run-time adaptive cache management techniques, optimizing both performance and cost of implementation. Specifically, we are aiming to increase data cache effectiveness for integer programs. We propose a microarchitecture scheme where the hardware determines data placement within the cache hierarchy based on dynamic referencing behavior. This scheme is fully compatible with existing Instruction Set Architectures. This paper examines the theoretical upper bounds on the cache hit ratio that cache bypassing can provide for integer applications, including several Windows applications with OS activity. Then, detailed

*This research was conducted while Teresa Johnson was a graduate student at the University of Illinois.

trace-driven simulations of the integer applications are used to show that the implementation described in this paper can achieve performance close to that of the upper bound.

Keywords: data cache, cache management, cache bypassing, temporal locality, spatial locality

1 Introduction

As processor performance improvements outpace that of main memory [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 improved *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. Even dynamically-scheduled processors, which have a natural ability to better hide the memory latency, have been shown to suffer from memory access penalties [2][3][4].

Methods to improve memory access behavior include *static* (compiler-only) methods, *dynamic* (hardware-only) methods, and *hybrid* (both compiler and hardware) methods. For numeric programs there are several known static compiler techniques for optimizing cache performance and tolerating the remaining latency. However, integer (non-numeric) programs often have irregular access patterns that are more difficult for the compiler to detect and optimize. For example, the temporal and spatial locality of linked list elements and hash table data are often difficult to determine at compile time. Because data cache performance opportunities for numeric programs are fairly well understood, this paper focuses on cache performance optimization for integer programs, in order to better understand what techniques are beneficial. Memory profiling could be used to guide compiler analysis for integer programs, however, no studies have been performed to demonstrate the reliability of mem-

ory profiling results over multiple input sets. Dynamic hardware methods, where caching decisions and latency tolerating prefetches are initiated at run-time, offer greater promise for integer programs. However, many of the previously proposed dynamic techniques have again focused on numeric codes. Other techniques have allowed only limited adaptability to run-time memory access patterns. Hybrid approaches would rely on the compiler to detect some static information, supplying hints to the hardware. For example, the HPL PlayDoh Architecture [5] specifies hints that can be added to load and store instructions, indicating where the corresponding data should be cached at run-time.

In order to increase cache effectiveness for integer programs this paper investigates *run-time adaptive cache hierarchy management*, which seeks to proactively control the movement and placement of data in the hierarchy based on the data usage characteristics. This paper presents a microarchitecture scheme where the hardware guides data caching based on dynamic referencing behavior. This scheme is fully compatible with existing Instruction Set Architectures.

Adaptive cache hierarchy management schemes seek 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, the notion of a *macroblock* is introduced. A macroblock is a contiguous block of memory that is large enough so that the maintenance 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. This extension to the cache microarchitecture significantly improves the overall performance of integer applications. The improvements are due to increased cache hit rates.

This paper extends our earlier work [6] by presenting analysis to determine the theo-

retical upper bounds on the hit ratio that can be achieved with cache bypassing. These upper bounds are independent of the prediction mechanism. Then, the performance of the hardware implementation described in this paper is compared to the upper bounds, and the opportunities missed by the implementation are examined. Additionally, in this paper the effects of bypassing are examined on a uniform (instruction and data) L2 cache, rather than exclusively on data caches. Finally, we include several Windows applications with OS activity in our benchmark suite.

The remainder of this paper is organized as follows: Section 2 discusses related work; Section 3 contains a case study of a particular benchmark, and presents the upper bound analysis; Section 4 discusses the hardware implementation; Section 5 presents simulation results and a comparison to the upper bounds; and Section 6 concludes with future directions.

2 Related Work

Several methods attempt to overlap memory accesses with other computation in the processor, in order 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 [7][8]. 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 [9][10][11][12][13][14][15] and software [16][17][18][19][20][4] prefetching methods for uniprocessor machines have been proposed. However, most of these methods focus on prefetching regular array accesses within well-structured loops[1], which are access pat-

[1]Here *regular* means arrays indexed by the loop iteration variable, or some other induction variable.

terns primarily found in numeric applications. Methods geared towards integer codes have largely been focused on compiler-inserted [2][21] or hardware-generated [22][23] prefetching of pointer targets. Prefetching techniques such as these are orthogonal to the bypassing methods discussed in this paper. There is also a great deal of prior work on prefetching in multiprocessor systems, but since their focus is even more on optimizing numeric applications, they will not be reviewed here.

Another approach to alleviating the memory penalty is to reduce the number of cache misses. Victim caches attempt to reduce conflict miss effects in caches with low associativity [10][24]. Also, both static and dynamic cache bypassing schemes have been investigated. These methods attempt to bypass data with little likelihood of in-cache reuse, and base the bypassing decisions either on the particular load referencing that data [25][26][27][24], or on the address being accessed [28][29][6][24]. The tradeoffs between these two bypassing approaches will be discussed in Section 3.1. The schemes also differ in their specific prediction mechanisms, and in how they handle bypassed data.

3 Concepts

3.1 Case Study

To understand some of the inefficiencies of conventional 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 [30]. 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

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.

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 monitored reuse ratios, which are different than hit 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, the accesses were profiled as explained above and the address distribution plotted for a given execution phase. The profiling results for a 100,000-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

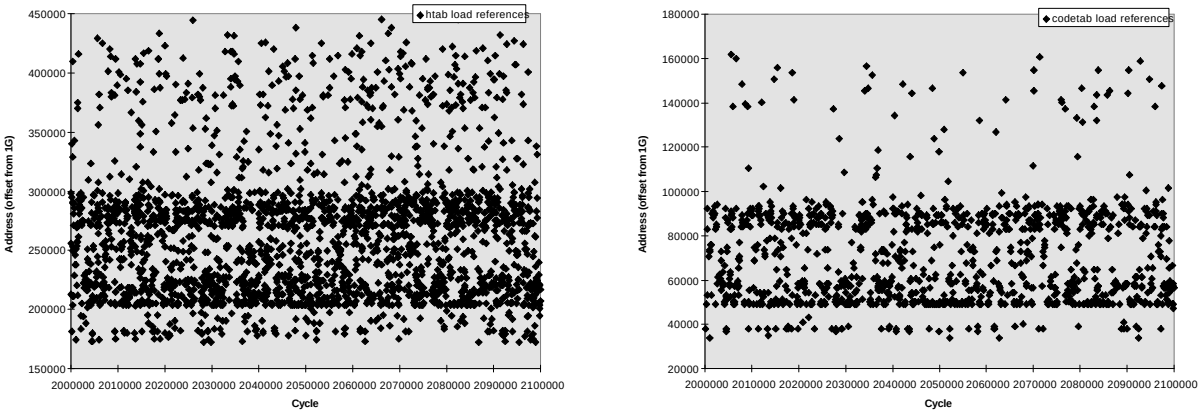
[2]The other load accesses to *htab* in this loop can be eliminated through load elimination optimizations.

[3]This profiler is a simplified version of the detailed simulator described in Section 5. 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.

[4]The reuse ratio is calculated in the following way. Suppose load A accesses a cache block (whether a hit or miss). The reuse counter for load A is incremented once if a load B subsequently hits on that same cache block. 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, and an instruction's hit and reuse ratios will be different.

```
    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



(a) htab (b) codetab

Figure 2: Memory Access Distributions for htab and codetab

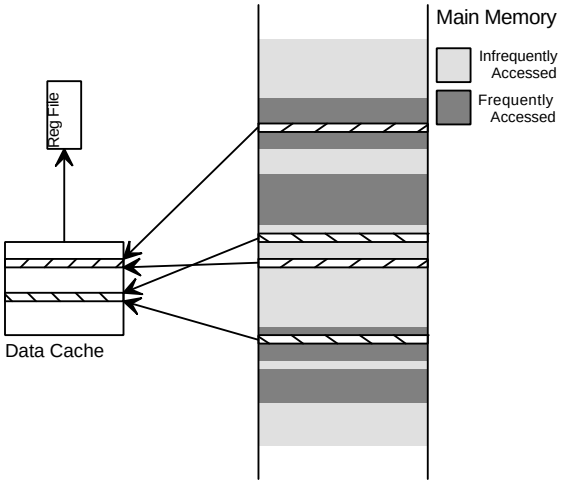
address of 1073741824, or 1GB). As the *htab* distribution shows, much of *htab* is relatively sparsely accessed, except for two bands that are heavily accessed. These bands are located roughly from addresses 200000 to 220000 and 257000 to 300000. Looking at several other execution phases of *compress* shows that this pattern is consistent 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 conditionally.

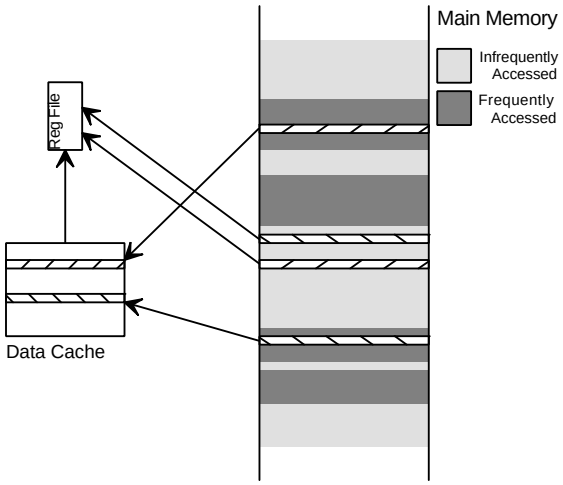
Ideally, the heavily-accessed bands shown in Figure 2 should remain in the cache despite contention for its locations, because the scattered accesses offer little reuse potential and only pollute the cache. More generally, Figure 3 shows how accesses to data with different usage frequencies should be handled. 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. Rather than replacing the heavily accessed block, which has a much greater chance of being reused in the near future, the

missing block would instead 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 can result in less cache pollution, and therefore increased reuse of more frequently accessed data, resulting in an overall increase in the hit ratio.

There are two main aspects on which the bypassing decisions can be based. One possibility is to base the decision on the program counter of the load generating the access. A program counter-based scheme would be preferred if the same memory address had different amounts of locality, depending on the code surrounding each load reference. The other possibility is to base the decision on the memory address of the access. An address-based scheme would be preferred if the same memory address has similar amounts of locality in each phase of the program execution, especially if the code surrounding the load instructions does not dictate the data locality. The memory access distributions of Figure 2 would be better predicted by the latter approach. In *compress* there are only two load instructions in the main loop body that access *htab*. However, the distributions show 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. While other benchmarks may contain patterns better predicted by the program counter, the *compress* behavior motivated our use of memory addresses in the bypass decision.



(a) Conflicts Between Data with Different Usage Patterns



(b) Bypassing Less Frequently Accessed Addresses

Figure 3: Conflict Misses in Compress.

3.2 Upper Bound Analysis

In order to gauge the potential effectiveness of cache bypassing, an upper bound analysis was performed. This section describes the algorithms used to determine an upper bound on the hit ratios that can be obtained with cache bypassing. An optimal replacement algorithm was originally developed by Belady [31]. Optimal replacement algorithms with bypassing have also been developed in the past [32][25], however, these only consider the basic case where bypassed data is not buffered. While our algorithm shares some similarities with these earlier optimal reuse algorithms, we developed our own algorithm with extendability to the case of a *bypass buffer*, described later, in mind.

To find the optimal bypassing decisions for all references mapping to each associative cache set, the trace is processed in reverse chronological order, and each reference is compared to all later references within the potential *reuse window* for the corresponding cache set. A *reuse window* for a cache set at time t is defined as the set of chronologically later references which could reuse the data supplied by reference R_t, given the bypassing decisions that have already been determined. In other words, the data supplied by R_t is in the cache during all

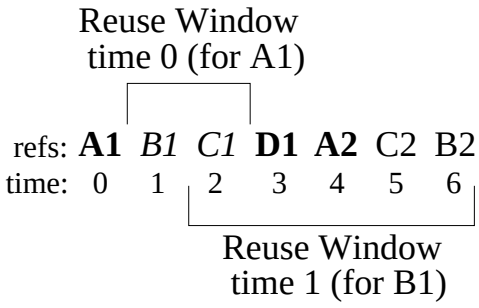
accesses in its reuse window, but is displaced by the last reference in its reuse window. All references to the same cache set occurring between R_t and the last reference in the reuse window must also be in the reuse window.

3.2.1 Simple Bypassing

Figure 4a shows a sample reference stream, where all references are assumed for simplicity to map to the same two-way associative cache set. References to the same cache block in memory are labeled with the same letter, and distinct references to the same cache block are numbered for clarity. Assume that $B2$ is the last reference in the trace, and that it has already been determined that $D1$ and $A2$ should bypass the cache[5]. Consider the reuse window at time 1, corresponding to the references that can reuse reference $B1$. At time 1, reference $B1$ will be cached in the least-recently used (LRU) entry, and will become the most-recently used (MRU) entry. At time 2, reference $C1$ will be cached in the new LRU entry, becoming MRU. Therefore, at the end of time 2, the cache set will contain $\{B,C\}$. Next, $D1$ and $A2$ bypass the cache, so the set contents are unchanged. Then, $C2$ reuses the data cached by $C1$, and the cache contents are again unchanged. Finally, $B2$ reuses the data cached by $B1$. Therefore, the decision is made to cache the data accessed by $B1$, and the reuse window at time 1 contains all later references in the sample reference stream.

Next, moving in reverse-chronological order, we consider reference $A1$ at time 0. Reference $A1$ will be cached in the LRU entry and become MRU. Next, $B1$ is cached in the new LRU entry and becomes MRU. The cache set after time 1 contains $\{A,B\}$. Next, $C1$ will displace the LRU entry, which is now the data cached by $A1$. Therefore, no references after $C1$ (time 2) can possibly reuse $A1$ with the given caching decisions, and $A1$ can also bypass

[5]A close inspection of the trace will illustrate that there are three possible reuses: $A1 \rightarrow A2$, $B1 \rightarrow B2$ and $C1 \rightarrow C2$. But in a two-way set-associative cache, only two of these reuses can be realized. Our algorithm will choose to exploit the reuses $B1 \rightarrow B2$ and $C1 \rightarrow C2$, because they are seen first.



(a) Reference stream and reuse windows.

Alg Iter	Time t	Ref_t	Reuse $Window_t$	Cache Set	Ref Type	Hit/ Miss
1	6	B2			N	H
2	5	C2	B2		N	H
3	4	**A2**	C2 B2		N	M
4	3	**D1**	**A2** C2 B2		N	M
5	2	*C1*	**D1 A2** C2 B2		R	M
6	1	*B1*	*C1* **D1 A2** C2 B2	*C*	R	M
7	0	**A1**	*B1 C1*	*B, C*	N	M

bypass: A1, D1, A2

(b) Illustrated bypassing algorithm.

Figure 4: Example reference stream and corresponding reuse window and LRU cache set at each step in the algorithm. All references in the stream are assumed to map into the same cache set, which is two-way set-associative. Note that the cache set column in (b) is the set maintained internally by the algorithm, not the actual run-time cache contents at that time step.

the cache. The optimality of this approach will be proven later in this section.

The main steps that must be completed by the algorithm for a given reference R_t are the following:

1. Determine if any reference in the corresponding cache set's reuse window has reuse with R_t.
2. Update the reuse window for the next previous reference R_j $(j < t)$ to the same cache set.

During the first step, R_t will be marked as one of two types of references:

$R{-}ref$ A reference that has a corresponding later reuse.

$N{-}ref$ A reference that has no corresponding later reuse.

Therefore, an $N{-}ref$ that misses can bypass the cache without harming the hit ratio. As a result, the reuse window can contain an arbitrary number of $N{-}refs$, possibly greater than the cache associativity, since they do not require cache space. In the example of Figure 4a, references $B1$ and $C1$, shown in italics, are both $R{-}refs$, since they are reused by references $B2$ and $C2$, respectively. All other references are $N{-}refs$. Each $R-ref$ has only one

corresponding reuse, and its corresponding reuse may also be an $R{-}ref$. Therefore, each set of references that access a particular block in the cache before that block is replaced form a *reuse chain* of the form $(R{-}ref \rightarrow)^* N{-}ref$. Note that the number of cache hits in each *reuse chain* is equal to the number of $R{-}ref$s, although the first $R{-}ref$ is a miss, and the $N{-}ref$ is a hit if there was at least one $R{-}ref$. In Figure 4a, the reuse chains are $\{\{A1\}$, $\{B1 \rightarrow B2\}$, $\{C1 \rightarrow C2\}$, $\{D1\}$, $\{A2\}\}$. References $C2$ and $B2$ will be cache hits, even though they are $N{-}refs$. Reuse chains that do not contain any $R{-}refs$ ($\{A1\}$, $\{D1\}$ and $\{A2\}$) will miss, and simply bypass the cache.

The algorithm assumes that all potential cache reuses have equal weight, and is detailed in Figure 5. The reuse windows are maintained in an array of *cache_queues*, one for each set in the cache. For each reference in the trace the corresponding reuse window is traversed once. The references in the reuse window are examined in chronological order, and the addresses are compared to determine if the current reference will result in a reuse within the window. If so, it is marked as an $R{-}ref$. At the same time, a temporary cache set is maintained to determine when the cache set will fill, forcing the displacement of any earlier references. This allows the second step described above to be performed in the same pass through the reuse window. Each reference is added to the set, if the address is not already in the set and if it is marked for reuse ($R{-}ref$). Once the size of the set matches the associativity, none of the subsequent references in the reuse window can obtain reuse with any references prior to the current reference, since any earlier references must necessarily be displaced from the cache under optimal bypassing before the subsequent references occur. Therefore, the subsequent references are simply discarded from the reuse window.

Proof. The proof of the optimality of this algorithm can be shown by induction, where we induct on the references in the trace, which are considered in reverse-chronological order. Assume for simplicity that all references in the trace map to a single set in the cache. *Basis.*

```
/* define */
CA_ASSOC = cache associativity;
CA_ENTRIES = number of sets in cache;
CACHE_INDEX(address) = set index corresponding to address;

algorithm upper_bound_bypass
{
    cache_queues[0..CA_ENTRIES] = NULL;

    for each address (reverse order) do:
    {
        new_entry->flag = N-ref;
        set = NULL;
        ca_entry = CACHE_INDEX(address);
        done = FALSE;

        for each entry in cache_queues[ca_entry] do:
        {
            if (!done)
            {
                if (new_entry->flag == N-ref && address == entry->address)
                    new_entry->flag = R-ref;

                if ((entry->addr not in set) && (entry->flag == R-ref))
                {
                    add entry to set;
                    if (|set| == CA_ASSOC) done = 1;
                }
            }
            else dequeue entry;
        }
        enqueue new_entry in cache_queues[ca_index];
    }
    dequeue all remaining entries in cache_queues[0..CA_ENTRIES];
}
```

Figure 5: Algorithm to determine optimal bypassing decisions.

If the trace is of length T, then the algorithm will process reference R_T at step 0. For step 0, the reuse window is empty, as there are no later references, and R_T has no reuse and is marked an $N{-}ref$. For step 1 the reuse window contains R_T, and if reference R_{T-1} matches R_T, then there is a reuse, and R_{T-1} is an $R{-}ref$. Otherwise it is an $N{-}ref$. It is trivially true that there can be no more reuse than is currently marked. Therefore, the algorithm is optimal for $k = 1$. *Induction.* Now consider step $k > 1$, and assume the first k assignments are optimal, and assume the reuse window contains w references. If R_{T-k} has reuse in the reuse window, it is an $R{-}ref$, and a cache hit has been added. Since every $R{-}ref$ can only create one hit, and any change to the decisions in the first k steps will only reduce the number of hits (since the first k steps are optimal), this is again optimal. If R_{T-k} does not have reuse in the reuse window or anywhere later in the trace, then there is no possibility of reuse, and it is marked an $N{-}ref$ without any loss of optimality. However, consider if R_{T-k} does not have reuse in the reuse window, but does have a match in the trace after the reuse window, say with R_l, where $l - k > w$. Since R_l is not in the reuse window, then the reuse window must contain A unique $R{-}refs$, where A is the cache associativity. In order to force R_{T-k} to have reuse with R_l, one cache location in the set must be taken from an $R{-}ref$, say R_i, in the reuse window (it could also require taking the cache location from later $R{-}refs$ outside the reuse window, but that would clearly be unoptimal and is not considered here). This will add one cache hit for the reuse pair $R_{T-k} \rightarrow R_l$, but remove the hit to R_j corresponding to R_i. Figure 6 shows the example reference stream with reference R_i in the reuse window, and a solid directed line pointing to its reuse with R_j. The potential reuse between R_{T-k} and R_l is denoted by the dashed directed line. The only way a net gain in cache hits can occur is if the removal of the reuse $R_i \rightarrow R_j$ allows another reuse $R_m \rightarrow R_n$ to occur. The $R_m \rightarrow R_n$ must require the same cache location as, and overlap with, $R_i \rightarrow R_j$. Otherwise $R_m \rightarrow R_n$ could have been achieved without removing reuse $R_i \rightarrow R_j$, in which

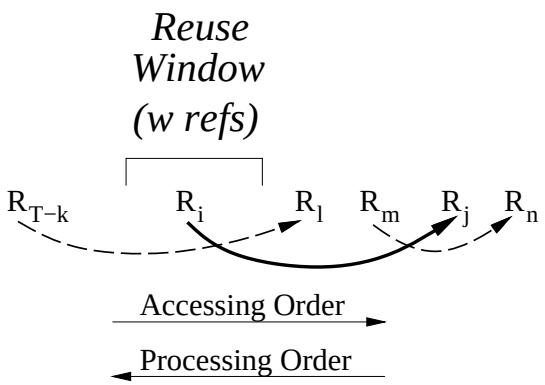


Figure 6: Example reference stream used in proof of the optimality of algorithm `upper_bound_bypass`. Reuses and potential reuses between reference pairs are shown with arrows.

case the first k assignments were not optimal, violating our assumption. Therefore, $m < j$. Additionally, $R_m \rightarrow R_n$ must utilize the same cache location as $R_{T-k} \rightarrow R_l$, in which case $l \leq m$. Therefore, $i < l \leq m < j$. The potential reuse pair R_m and R_n are also shown in Figure 6. Because the algorithm works backwards and assigns reuses greedily, if $i < m < j$, then the algorithm would have seen reuse $R_m \rightarrow R_n$ before $R_i \rightarrow R_j$, as illustrated in Figure 6, and would have assigned the cache location to $R_m \rightarrow R_n$, not $R_i \rightarrow R_j$. Hence, forcing reuse between R_{T-k} and R_l cannot result in a net gain in cache hits, given that the first k assignments were optimal.

Figure 4b shows the reuse window and the algorithm's cache set contents at each iteration of the algorithm. An examination of this stream will show that the hit ratio in a conventional cache would be 0. Reference $B2$ is examined first, as it is the last reference in the stream. At this point the reuse window is empty, so there is no reuse, and $B2$ is marked as an $N{-}ref$. As such, it is not placed in the cache set. Reference $C2$ does not match any address in the reuse window, which simply contains $B2$, and is also marked as an $N{-}ref$. This continues until $C1$ is examined and found to match $C2$ in the reuse window. Therefore, $C1$ is marked as an $R{-}ref$, and $C2$ will be a cache hit. Because $C1$ is an $R{-}ref$, it is placed in the cache

set. However, the cache set does not overflow, so no dequeuing (removing a reference from the reuse window) is necessary. Next, $B1$ is found to be an $R{-}ref$, matching $B2$. Now the cache set contains both B and C, which equals the cache associativity, and would displace from the set any earlier references ($A1$), which can no longer be reused by references after $C1$. Therefore all references following $C1$ are removed from the reuse window. Note that for efficiency the algorithm will actually dequeue the later references from the reuse window during the next iteration; however, for simplicity the example is shown as if the dequeuing occurred at the end of the current iteration. Finally, when $A1$ is examined, the reuse window no longer includes $A2$; therefore, $A1$ is marked as an $N{-}ref$. The table also shows these reference markings and the new cache access results for each corresponding reference. In the ideal case, $N{-}refs$ that miss in the cache would bypass, in this case $A1$, $D1$ and $A2$. The new hit ratio is 2/7.

3.2.2 Bypassing with a Bypass Buffer

While the bypassing algorithm described in Section 3.2.1 will reduce misses by not caching data that has no reuse, it is possible that a reuse may not be exploited due to limited set-associativity of the cache. Therefore, Section 4 will propose adding a *bypass buffer* to the cache hierarchy. Bypassed data is placed in this small set-associative buffer, which is then accessed in parallel with the data cache. In this situation, some data with a few short-term reuses can be bypassed into the buffer, which exploits the reuse, while not displacing more frequently-accessed data from the main cache. With a bypass buffer, the situation becomes significantly more complex. Now each entry can be placed in one of two cache sets, and the set of references with which a given reference will conflict in the different cache sets are not identical. It is also unclear when to assign a reuse chain to the main cache or the bypass buffer.

accessing order →

processing order ←

A1 B1 A2 A3

step	cache1 reuse window	cache2 reuse window
0	A3	A3
1	A2,A3	–
2	B1	B1

(a) reuse assigned to $cache_1$ and removed from $cache_2$

accessing order →

processing order ←

C1 B1 B2 A1 A2 C2

step	cache1 reuse window	cache2 reuse window
0	C2	C2
1	A2	A2,C2
2	A1,A2	A1,A2,C2
3	B2	B2,A1,A2
4	B1,B2	B1,B2,A1,A2

(b) reuses left in both caches

Figure 7: Example reference streams and corresponding $cache_1$ and $cache_2$ reuse windows at each step in the algorithm, for two algorithms. Here, $cache_1$ is direct-mapped, and $cache_2$ is two-way set associative.

Consider the case when a reuse is detected between two references in both caches, meaning that the reuse could be exploited in either cache. One possibility would be to assign the reuse to $cache_1$ and remove the references from the reuse window in $cache_2$. However, it is possible that another earlier (chronologically) reference accesses the same data, but the intermediate references are such that the data would be displaced from $cache_1$ before the reuse, but not from $cache_2$. An example of this situation is shown in Figure 7a, where $cache_1$ is direct-mapped, and $cache_2$ is two-way set-associative. Assume that addresses A and B map to the same set in both caches. Reuse $A2 \rightarrow A3$, found during step 1, can occur in either cache. If $cache_1$ is chosen, and the references are removed from the reuse window of $cache_2$, then no reuse is detected for reference $A1$, since $B1$ displaces the references to A from the reuse window of direct-mapped $cache_1$. Since $cache_2$ is two-way set associative, the reuse $A1 \rightarrow A2$ could have been exploited in $cache_2$.

On the other hand, if the reuse is left in both caches' reuse windows, since the reuse chain must eventually be assigned to a single cache, the reuse window of the other cache will be pessimistically restricted by the data. An example of this situation is shown in Figure 7b, for the same cache associativities described above. Assume that A, B, and C map to the same sets in both caches. The reuses $A1 \rightarrow A2$ and $B1 \rightarrow B2$ will be detected in both caches. If these references are left in both caches' reuse windows, the reuse $C1 \rightarrow C2$ will not be detected, as $C2$ would have been pessimistically replaced from both reuse windows before $C1$ is processed.

Therefore, some simplifications to the algorithm are made, which result in an upper bound that may not always be achievable in practice. These simplifications will be described along with the algorithm, in the remainder of this section. A truly optimal algorithm for this case would most likely require very expensive backtracking procedures during trace analysis. As we will show in Section 5.2, the simplifications utilized here still result in a useful comparison of the bypass mechanism described in Section 4 to an upper bound. Also, we will provide a bound on how much inaccuracy could be introduced for the benchmarks in Table 2.

Now, each reference can be marked as one of five types of references:

R_1-ref A reference that has a corresponding later reuse in $cache_1$.

N_1-ref A reference that has no corresponding later reuse, but itself reuses data in $cache_1$.

R_2-ref A reference that has a corresponding later reuse in $cache_2$.

N_2-ref A reference that has no corresponding later reuse, but itself reuses data in $cache_2$.

$N-ref$ A reference that has no corresponding later reuse, and does not reuse data itself in either cache.

The algorithm is outlined in Figures 8-9.

```
/* define */
CA1_ASSOC = cache 1 associativity;
CA2_ASSOC = cache 2 associativity;
CA1_ENTRIES = number of sets in cache 1;
CA2_ENTRIES = number of sets in cache 2;
CACHE1_INDEX(address) = set index in cache 1 corresponding to address;
CACHE2_INDEX(address) = set index in cache 2 corresponding to address;

algorithm upper_bound_bypassbuf
{
    cache1_queues[0..CA1_ENTRIES] = NULL;
    cache2_queues[0..CA2_ENTRIES] = NULL;

    for each address (reverse order) do:
    {
        new_entry->flag = N-ref;
        ca1_entry = CACHE1_INDEX(address);
        ca2_entry = CACHE2_INDEX(address);

        /* Give priority to bypass buffer (cache 2) reuse */
        check_cache(2,1,1);

        /* If no reuse found yet, check cache1 */
        if (new_entry == N-ref)
            check_cache(1,2,2);

        /* If no reuse found yet, check for C2->C1 reuses */
        if (new_entry == N-ref)
            check_cache(2,1,2);

        /* If no reuse found yet, check for C1->C2 reuses */
        if (new_entry == N-ref)
            check_cache(1,2,1);

        if (new_entry->flag == N-ref)
        {
            enqueue new_entry in cache1_queues[ca1_index];
            enqueue new_entry in cache2_queues[ca2_index];
        }
    }
    dequeue all remaining entries in cache1_queues[0..CA1_ENTRIES];
    dequeue all remaining entries in cache2_queues[0..CA2_ENTRIES];
}
```

Figure 8: Algorithm to determine upper bound bypassing decisions in the presence of a bypass buffer.

```
algorithm check_cache(i,j,j)
{
    set = NULL;
    done = FALSE;

    /* Check cachei */
    for each entry in cache(i)_queues[ca(i)_entry] do:
    {
            if (!done)
            {
                if (new_entry->flag == N-ref && address == entry->address &&
                        (entry->flag != Nk-ref || entry->flag != Rk-ref))
                {
                    new_entry->flag = Ri-ref;
                    if (entry->flag == N-ref) entry->flag = Ni-ref;
                    enqueue new_entry in cache(i)_queues[ca(i)_index];
                    enqueue new_entry in cache(j)_queues[ca(j)_index];
                }

                else if ((entry->addr not in set) &&
                        (entry->flag == Ni-ref || entry->flag == Ri-ref))
                {
                    add entry to set;
                    if (|set| == CAi_ASSOC) done = 1;
                }
            }
            else dequeue entry from cache(i)_queues[ca(i)_entry];
    }
}
```

Figure 9: Routine to support upper bound bypassing decisions.

Figure 8 shows the main body of the upper bound algorithm for bypassing with a bypass buffer. Now two arrays of reuse windows are maintained, one for each cache, the main data cache ($cache_1$) and the bypass buffer ($cache_2$). Many of the steps are the same as those shown in the bypassing algorithm of Figure 5, except that two caches must be checked and maintained. Because the bypass buffer is designed to hold data with shorter-term temporal locality, the algorithm first checks the bypass buffer for reuse. If none is found, the main cache is checked. The checking is performed in the *check_cache* routine, shown in Figure 9. Although it is a heuristic to check the bypass buffer first, this heuristic will rarely result in less than the maximum achievable performance, as will be explained later. The final two calls to *check_cache* will be explained later. The *check_cache* routine in Figure 9 is similar to the checking loop for the bypassing algorithm of Figure 5.

The simplification made to solve the problem described earlier is to place references in both of the caches' reuse windows, however, this reference is only counted for the purposes of updating the reuse window in the cache containing the reuse. Therefore, the other cache's reuse window will contain extra references, allowing more reuses to be detected by the algorithm than can possibly occur. However, this prevents a reuse which could have occurred in both caches from pessimistically shortening both of the windows, and at the same time allows the reference to be "reassigned" to the other cache if it is later found to result in a longer reuse chain. Returning to the example shown in Figure 7b, reuses $A1 \rightarrow A2$ and $B1 \rightarrow B2$ would be left in both caches' reuse windows, but would be assigned to one cache, and not counted in the other cache's reuse window. An examination of the reference stream will show that, regardless of the cache to which the reuses $A1 \rightarrow A2$ and $B1 \rightarrow B2$ are assigned, with this algorithm $C2$ would not be displaced from both reuse windows before $C1$ is processed, and the reuse $C1 \rightarrow C2$ will be detected.

Algorithm *upper_bound_bypassbuf* first checks the two caches for reuse to a reference assigned to the same cache. However, if no reuse is found, the algorithm then checks each cache for reuse to a reference already assigned to the other cache. This is performed by the last two calls to *check_cache*, which are called with slightly different arguments. If one such reuse is found, the tail reference (the reference occurring later in time but processed earlier by the algorithm) of the new reuse is "reassigned" to this cache. There is one aspect of this simplification that can still make the results pessimistic. When references are "reassigned" to another cache as described above, the reuse window originally containing the reference may have been pessimistically shorted before the reassignment. However, it was found that reassignments generally constituted less than 1% of the accesses in each benchmark, as will be illustrated in Table 2. Each reassignment can result in at most one extra (optimistic) hit in the cache to which the reuse chain is reassigned, and at most one extra (pessimistic) miss

in the cache to which the reuse chain was previously assigned. Therefore, if the optimistic and pessimistic effects do not completely cancel each other out, the effect on the hit ratio will be minimal, which the results in Section 3.2.3 will demonstrate. As described later, the bounds will be computed by assuming that each reassignment causes ± 1 hit.

Additionally, if reuse did not occur for a reference, it is inserted into both caches' reuse windows by algorithm *upper_bound_bypassbuf*. It is then not counted in either reuse window until it is assigned to one cache by an earlier reference found to have reuse with this reference in a particular cache. References that are neither reused by a later reference nor found to reuse earlier references remain $N\text{–}refs$, and therefore are treated as references which bypass without being placed in the bypass buffer. This again can result in an unachievable upper bound with this algorithm, since we will always place references in either the main cache or the bypass buffer.

3.2.3 Upper Bound Results

Table 2 presents the hit ratios that are obtained performing bypassing both with and without bypass buffers. Results for a direct-mapped 16K-byte cache with 32-byte lines are shown. The bypass buffer, when used, is 4-way set-associative with 32 32-byte lines. Also shown are the hit ratios obtained with a conventional direct-mapped cache of the same size (*Base Hit Ratio*). The percentage of accesses resulting in reassignments, causing both optimistic and pessimistic effects on the hit ratio as described above, is shown after the $\pm$ when using a bypass buffer.

The results show that bypassing can potentially obtain significant improvements in the hit ratios. However, for many benchmarks, only small improvements are achieved when no bypass buffer is utilized. A bypass buffer is required, even under optimal bypassing, to obtain uniformly significant improvements across all benchmarks. As such, we use a bypass buffer

Benchmark	Base Hit Ratio	Bypassing Hit Ratio	
		No Bypass Buffer	With 1-KB Bypass Buffer
026.compress	69.12%	75.26%	76.38±0.32%
072.sc	93.02%	93.45%	94.78±0.62%
099.go	93.42%	94.95%	98.33±0.32%
147.vortex	95.24%	96.54%	98.80±0.19%
Pcode	80.50%	83.12%	87.48±1.26%
lmdes2_customizer	85.17%	88.25%	91.96±0.53%
085.cc1	93.71%	94.46%	96.37±0.45%
130.li	93.59%	93.96%	95.66±1.18%
134.perl	95.16%	95.57%	97.42±0.10%
124.m88ksim	98.53%	98.81%	99.44±0.02%
word	91.55%	93.65%	97.35±0.43%
excel	93.68%	94.90%	97.78±0.29%
photo	99.11%	99.29%	99.67±0.07%

Table 2: Hit ratios for 16K-byte direct-mapped cache with 32-byte lines both with and without bypassing.

throughout the remainder of this paper.

Our upperbound results show that cache bypassing can achieve significant improvements in the cache hit ratios of integer benchmarks. The following sections describe one implementation of a cache bypassing optimization which attempts to achieve these upper bound hit ratios. After simulation results show the performance of this implementation in Section 5, the performance will be compared to the upper bounds presented in this section, and the opportunities missed by the implementation are examined.

3.3 Macroblocks

Performing selective cache bypassing requires some method of monitoring memory access behavior. As discussed in Section 3.1, the scheme examined in this paper is address-based rather than program counter-based, and therefore must monitor the behavior of different memory regions. Ideally, the usage frequencies of all cache block size data in memory would be monitored. While this would provide the most accurate information, it would result in an unmanageably large amount of information. Instead, groups of adjacent cache block size data are combined 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 the accessing frequency of each macroblock is monitored through some hardware mechanism, then it is possible to determine on a macroblock basis whether or not to cache the contained data.

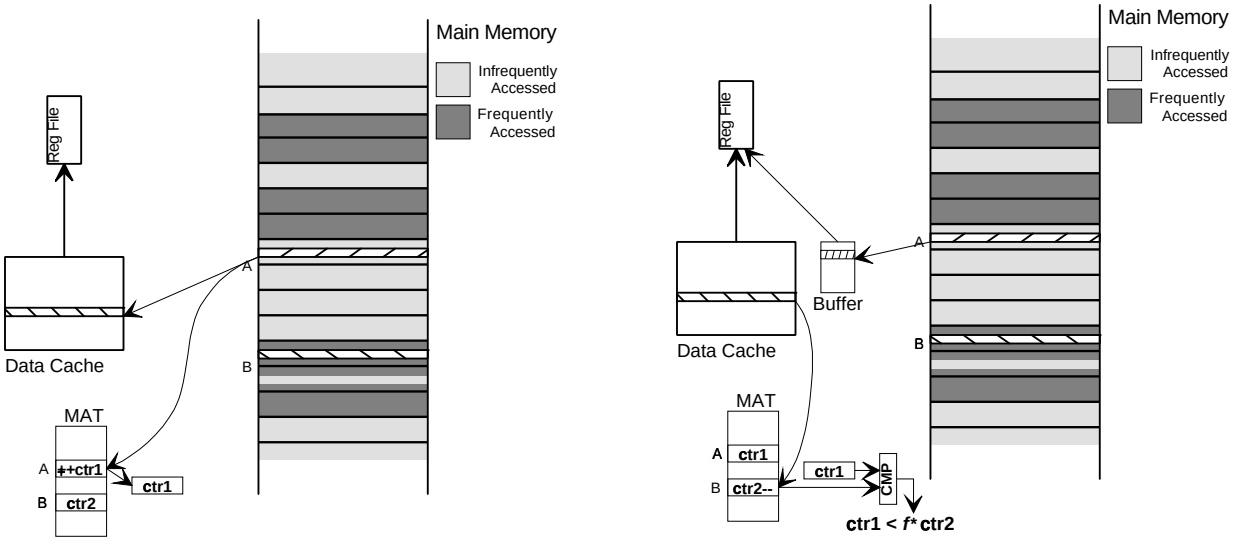
Section 5 examines the effects of varying the granularity of the macroblocks.

4 Hardware

The macroblocks are monitored at run time using a table in hardware called a *Memory Address Table* (*MAT*). The MAT ideally contains an entry for each macroblock, which contains information about the monitored access behavior of the corresponding memory region. In this work, 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 10a, 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 would be replaced to determine which data is more heavily accessed, and therefore predict which is more likely to be reused in cache, as shown in Figure 10b where data in macroblock B would be replaced. This counter value (*ctr2*) is then decremented and compared to the counter value corresponding to the



(a) Accessing *ctr1*. (b) Accessing *ctr2* and bypassing.

Figure 10: MAT Operation.

missing access. The actual comparison performed is:

$$ctr1 < f * ctr2 \tag{1}$$

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 *ctr1* is compared to some

fraction f of *ctr2*. Selecting f less than 1.0 will conservatively prevent bypassing when the counter values are almost the same.

Also, bypassing is avoided 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*), *ctr1* is compared 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 MAT 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. The new counter will start at an equal position, resulting in more selective bypassing. Simulations showed that initialization of *ctr1* with *ctr2* tends to produce better bypassing decisions.

This mechanism uses accessing frequency to estimate the likelihood of data locality. However, there may be temporal or spatial locality within the block, even when the total accessing frequency is relatively low. In 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 prevented by not displacing the much more frequently accessed data.

To optimize performance in this situation, bypassed data is placed in a small set-associative buffer, as shown in Figure 10b. This *bypass buffer* is accessed in parallel with the main cache. As a result, the bypassed data is held close to the processor for a short time, allowing much of the locality of the infrequently accessed data to be exploited. Unlike victim caches, data

Benchmark	Description	Instr Count	Base IPC
026.compress	Performs adaptive Lempel-Ziv coding (*SPEC* reference input)	68M	0.58
072.sc	Spreadsheet operations (*SPEC* reference input)	112M	1.61
099.go	Artificial intelligence in game playing (training input)	533M	0.78
147.vortex	Single-user object-oriented database transaction benchmark (training input)	2414M	1.26
Pcode	*IMPACT* compiler front end (dependence analysis on GNU CC *combine.c* file)	332M	0.55
lmdes2_customizer	*IMPACT* compiler machine description optimizer (SuperSPARC description opti)	15M	0.97
085.cc1	Performs GNU C compilation (*SPEC* reference input)	135M	0.93
130.li	Lisp interpreter (training input)	159M	1.60
134.perl	Perl interpreter (training input)	1100M	1.54
124.m88ksim	Simulator for the 88100 microprocessor (training input)	123M	2.00
Word	Microsoft Word Version 7.0a (search and replace)	5M	0.20
Excel	Microsoft Excel Version 7.0a (spreadsheet cell updates)	5M	0.26
Photo	Adobe PhotoDeluxe Version 1.0 (brightness and contrast adjustment)	8M	0.66

Table 3: Benchmark descriptions.

will never be swapped into the data cache on hits to the bypass buffer, as the low accessing frequency indicates less likelihood of long-term reuse. As illustrated in Section 3.2, adding a bypass buffer significantly increases the potential improvements from cache bypassing, even when the inaccuracies of the implemented prediction mechanism are filtered out.

The cost of the MAT hardware has been analyzed elsewhere [6][33]. While the hardware cost is much smaller than the cost of doubling the caches, the MAT scheme often outperformed the doubled caches.

5 Experimental Evaluation and Analysis

5.1 Experimental Environment

Cycle-accurate trace-driven simulations were performed for a number of non-numeric benchmarks in order to determine the performance of our scheme in comparison to conventional caches. The benchmarks are summarized in Table 3, along with their dynamic instruction counts and the baseline IPCs for the system described in Section 5.1.1.

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 4: Instruction latencies for simulation experiments.

5.1.1 Simulation of SPEC and IMPACT Benchmarks

In order to provide a realistic evaluation of our technique for future high-performance, high-issue rate systems, we first optimized the SPEC and IMPACT benchmark source code using the *IMPACT* compiler [34]. Classical optimizations were applied, then optimizations were performed which increase instruction level parallelism. The code was scheduled, register allocated and optimized for an eight-issue, in-order (statically-scheduled), scoreboarded (using a CRAY-1-style interlocking approach), superscalar processor with register renaming. The ISA is an extension of the HP PA-RISC instruction set that supports compile-time speculation.

We perform cycle-accurate emulation-driven simulation on a Hewlett-Packard *PA-RISC 7100* workstation, modeling the processor and the memory hierarchy (including all related busses). The instruction latencies used are those of a Hewlett-Packard *PA-RISC 7100*, as given in Table 4. The base machine configuration is described in Table 5.

Since simulating the entire applications at this level of detail would be impractical, uniform sampling is used to reduce simulation time [35], however emulation is still performed between samples in order to update the CPU state so that accurate trace information can be generated during later samples. The simulated samples are 500,000 instructions in length and are spaced evenly across program execution, yielding at least a 1% sampling ratio. For smaller applications, the time between samples is reduced to maintain at least 20 samples (10,000,000 instructions). To evaluate the accuracy of this technique, we simulated several configurations both with and without sampling, and found that the improvements reported

L1 Icache	32KB split-block, direct mapped, 64-byte block
L1 Dcache	16KB non-blocking (50 max), direct mapped, 32-byte block, multiported, writeback, no write alloc
L1-to-L2 Bus	8-byte bandwidth, split-transaction, 4-cycle latency, returns critical word first
L2 Cache	same as L1 Dcache except: Unified I&D, 256KB, 64-byte block
System Bus	same as L1-to-L2 Bus except: 100-cycle latency
Issue	8-issue uniform, except 4 memory ops/cycle max
Registers	64 integer, 64 double precision floating-point
Branch Prediction	1K-entry direct-mapped Branch Target Buffer performs dynamic prediction using a 2-bit counter. Branch misprediction penalty is 2 cycles.

Table 5: Base Configuration.

in this paper are very close to those obtained by simulating the entire application [33].

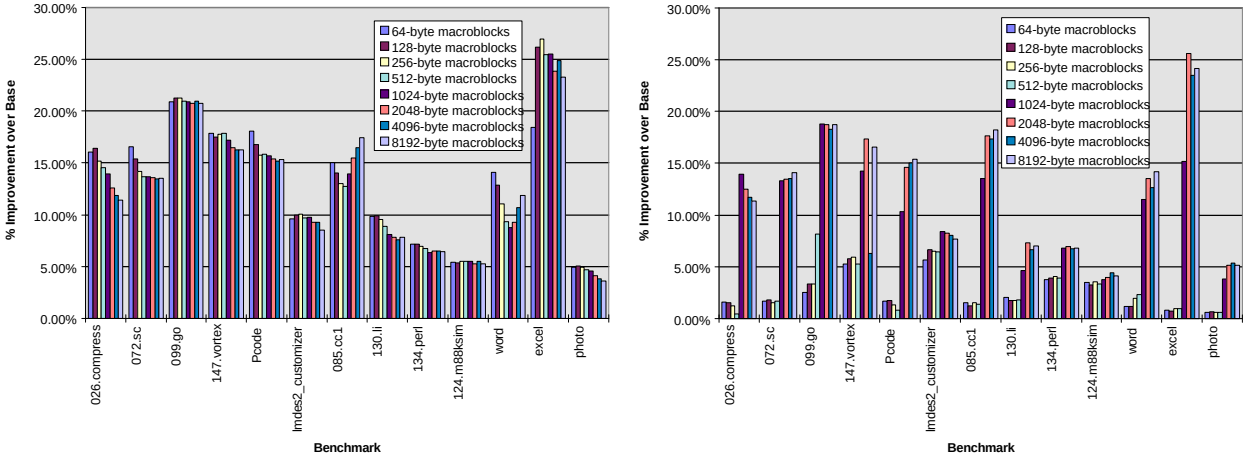
5.1.2 Simulation of Windows NT Benchmarks

Hardware tracing facilities on an AMD K6 [36] system were used in conjunction with special hardware provided by AMD, called SpeedTracer, to take traces of instructions and memory addresses for several Windows NT applications along with their corresponding OS activities. No sampling was performed during simulation of these benchmarks.

5.1.3 Bypassing Simulation

The first set of configurations use the cache configuration described in Table 5, and the MAT scheme presented in Section 4. 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. Results are shown for both infinite-entry and 512-entry MATs. Each MAT is direct-mapped with 8-bit access counters. Both MATs use the same value of f and the same macroblock size. After a large number of simulations $f = 1.0$ was found to have the best performance across all the benchmarks [33]. The bypass buffer supports aggressive bypassing from $f = 1.0$ without much effect from any incorrect bypassing decisions made by the higher f value. Results will be shown for a range of macroblock sizes.

Because the upper bound analysis presented in Section 3.2 showed that bypass buffers are



(a) Infinite-entry MAT

(b) 512-entry MAT

Figure 11: Performance improvements using different macroblock and MAT sizes.

necessary to obtain uniformly significant hit ratio improvements, all configurations examined in this paper utilize bypass buffers. The 4-way set-associative buffers used to hold the bypassed data at the L1 and L2 caches contain 32 and 256 entries, respectively.

The effects of bypassing with greater cache set-associativity and longer memory latencies were investigated elsewhere [6][33], and will not be repeated here. It was shown that most benchmarks can still achieve significant improvements from bypassing with set-associative caches, and that the speedups quickly increase as the memory latency increases.

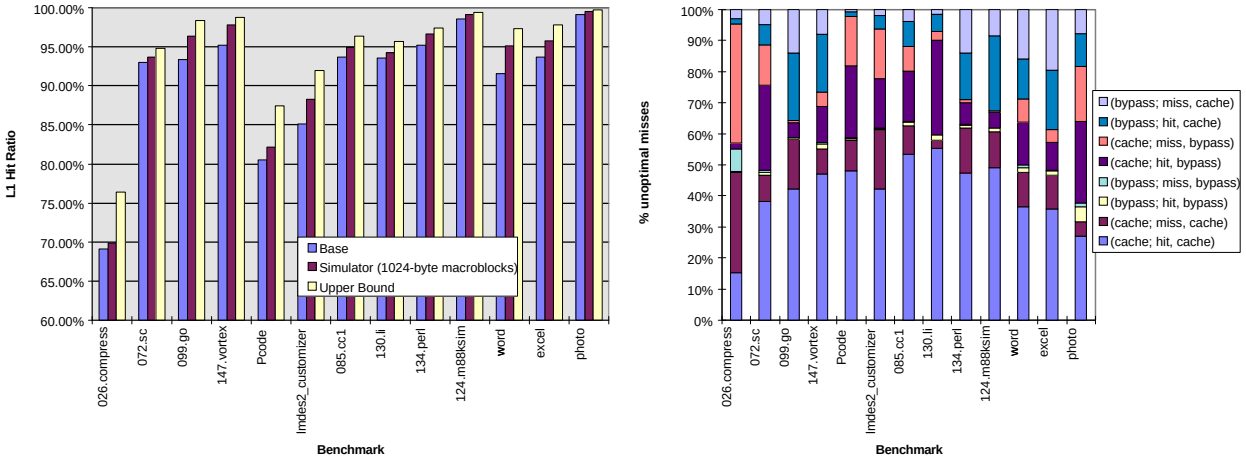
5.2 Results

Figure 11a shows the speedup of each benchmark for the MAT scheme described above, for a range of macroblock sizes using an infinite-entry MAT. Most benchmarks achieve over 10% improvement in execution time, reaching a 25% improvement in the case of *Excel*. The results do vary across the different macroblock sizes, and generally become worse as the macroblock size is doubled. In the case of *085.cc1*, the performance improves as the macroblock size grows, which is an anomaly resulting from the more conservative bypassing decisions

that tend to result with larger macroblock sizes. However, while smaller macroblocks give improved resolution, these results do not take into account the larger number of macroblocks that must be monitored for the smaller macroblock sizes. With 64-byte macroblocks, the L1 cache often monitored several thousand macroblocks, and the unified L2 cache, which contains both data and instructions, monitored up to 30,822 macroblocks. Therefore, Figure 11b shows performance improvements for the same range of macroblock sizes, with a 512-entry direct-mapped MAT. In this case, the MATs were far too small to monitor significant numbers of macroblocks for the smaller sizes. Generally, 2K-byte macroblocks achieve the best results, as the number of macroblocks is manageable, but the accuracy is still high enough to result in good bypassing decisions.

The L1 data cache read hit ratios for the same system configuration as described in Section 5.1.3, with 1K-byte macroblocks and an infinite-entry MAT, are shown in Figure 12a. Also shown for reference are the L1 hit ratios for the base configuration, as well as the upper bound hit ratios reported in Table 2. In most cases the implementation described in Section 4 achieves roughly half of the upper bound hit ratio improvement. A few benchmarks, most notably *compress* and *Pcode*, achieve very little of the upper bound performance. These cases will be examined in greater detail below.

Figure 12b shows more information about the accesses R_2 which had reuse with earlier references R_1 in the upper bound analysis, but which were displaced from the cache without reuse in the simulated implementation. In all cases R_2 missed in the simulator, but R_1 may have hit or missed. The bottom four sections of each bar represent those cases when the reference R_1 that should have been reused was placed in the same cache (main cache or bypass buffer) as in the upper bound analysis. The reason for its displacement would be that either too much data was assigned to the same cache by the implementation, or that the upper bound analysis was too optimistic. These cases make up around 50% of the misses



(a) L1 data cache read hit ratio

(b) Breakdown of unoptimal misses

Figure 12: L1 data cache read hit ratio comparison to upper bound, and breakdown of accesses that had reuse in the upper bound analysis but not in the simulated implementation. The legend for (b) denotes (R_1 upper bound cache/bypass; R_1 simulator hit/miss, R_1 simulator cache/bypass).

examined.

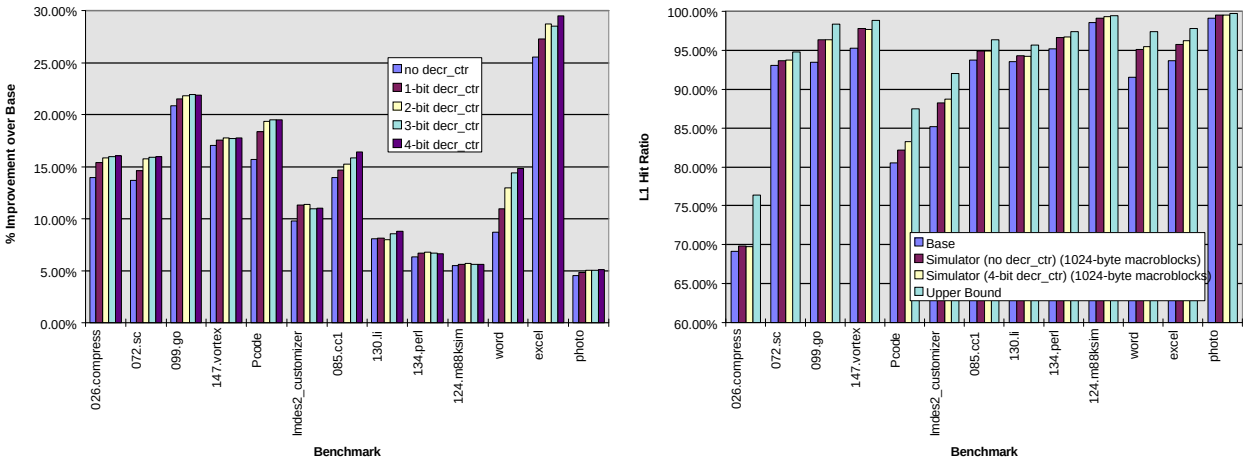
For the remaining misses, the reference R_1 which had reuse in the upper bound analysis was located in a different cache than that assigned by the upper bound analysis. The graph shows that in many of these cases reference R_1 was located in the bypass buffer in the simulator, but not in the upper bound analysis, and was later displaced from the bypass buffer before the reuse by R_2 could occur. In some of these cases R_1 missed and was bypassed, and in other cases R_1 hit, meaning that the decision to bypass the corresponding data was made by an earlier missing reference. In order to understand why the decision was made to bypass R_1 in the cases where it missed, the counter values used in the comparisons for several benchmarks were examined. The values of *ctr1* and *ctr2* from Equation 1 were examined for both the reference R_1 that was incorrectly bypassed, and the reference R_2 which later missed in the simulator (but hit in the upper bound analysis). Table 6 shows the average difference between $ctr2^{R_1}$ and $ctr1^{R_1}$, the average increase in $ctr1$ between R_1 and R_2, and the average increase in $ctr2$ between R_1 and R_2. Note that $ctr1^{R_1}$ and $ctr1^{R_2}$ must be the same physical

Benchmark	Ave($ctr2^{R_1}$-$ctr1^{R_1}$)	Ave($ctr1^{R_2}$-$ctr1^{R_1}$)	Ave($ctr2^{R_2}$-$ctr2^{R_1}$)
026.compress	10.04	2.08	-1.68
072.sc	27.72	18.94	-8.05
099.go	13.19	6.88	-0.70
147.vortex	37.05	3.68	-0.65
Pcode	6.44	2.75	-3.78
lmdes2_customizer	12.41	4.29	-1.71
085.cc1	15.96	8.44	-2.49
130.li	22.16	17.20	-9.12
134.perl	50.48	15.60	-4.95
124.m88ksim	83.64	49.59	1.84
Word	89.56	5.43	-0.02
Excel	83.16	4.60	0.21
Photo	117.89	7.74	1.47

Table 6: Average difference between counter values for references which were not bypassed and had reuse in the upper bound analysis, but which were bypassed and did not have reuse in the simulated implementation. R_1 is the reference which missed and should not have bypassed, and R_2 is the reference which should have reused R_1 but did not.

counter, since R_1 and R_2 reference the same cache block. However $ctr2^{R_1}$ and $ctr2^{R_2}$ may not be the same physical counter, depending on the accesses and cache replacements made between R_1 and R_2.

Table 6 shows that the average difference between the counter values used in the decision to bypass R_1 can be large. Also, because *ctr2* decreases between the two accesses for most benchmarks, the corresponding macroblock is not being accessed as heavily as the macroblock containing the accessed data. However, the average *ctr1* increases and *ctr2* decreases do not come close to covering the gap between the two counters. This suggests that the counter values are taking too long to converge. There are several possible changes that could increase the speed at which the two counters converge. One is to simply reduce the number of bits in the counters. Another is to modify the rate at which $ctr2$ is reduced. It is possible that the heuristic used to decrement the access counters could be improved over simply decrementing by one on every conflict. For example, another small counter could be added to each entry, that detects when cached data from the macroblock is no longer being actively accessed. The saturating counter, called *decr_ctr*, could be incremented by one on every conflict for a cache location held by its macroblock (i.e. whenever its access counter is decremented),



(a) Performance improvement (b) L1 data cache read hit ratio

Figure 13: Performance improvements and cache hit ratios using variable access counter decrements.

and cleared to zero whenever its macroblock is accessed (i.e. whenever its access counter is incremented). In other words, the value of *decr_ctr* is the number of conflicting accesses to other macroblocks that have occurred since the last access to this macroblock. Then, the access counter could be decremented by one plus the value of *decr_ctr* on a conflicting access to another macroblock, rather than by one, effectively enabling access counters of macroblocks no longer being accessed to decrease faster.

Simulations showed that reduced access counter sizes tended to reduce performance [33]. Though smaller access counters produce quicker response times in the counter values, they also reduce the resolution of the comparisons. On the other hand, the variable counter decrement scheme, using a *decr_ctr*, increased performance in almost all cases. Figure 13a shows the performance improvements using 1K-byte macroblocks and infinite-entry MATs for 1 to 4-bit *decr_ctr* sizes in both the L1 and L2 MATs. These results show that accelerating the access counter decrements for less-actively-accessed macroblocks can significantly improve performance. Figure 13b shows the same hit ratios shown in Figure 12a, with the hit ratios using a 4-bit *decr_ctr* added. A comparison of Figure 13a with Figure 13b suggests that even

small improvements in the hit ratios can translate into good performance improvements. The hit ratio for *026.compress* actually decreases, even though the execution cycles are reduced. In this case, the new stream of references sent to the L2 cache have improved locality at the L2 level, and result in fewer L2 misses (even when the L2 cache does not use a *decr_ctr* and therefore does not change).

These results illustrate that the upper-bound analysis can provide useful insights into the behavior of bypassing implementations, and point to ways to improve the mechanism. Additional analysis can be done on the references described in Figure 12b to further improve the performance of the bypassing implementation, so that it converges with the upper-bound performance.

6 Conclusion

In this paper, we examined methods 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. An upper bound analysis was first performed to determine how much improvement in the cache hit ratio could be expected from bypassing. The results of this analysis on a number of integer benchmarks showed that bypassing could achieve significant performance improvements, but that even with optimal bypassing decisions a *bypass buffer* is required to obtain uniformly significant improvements.

Then an implementation of the cache bypassing optimization was presented. The bypassing choices are made by a Memory Address Table (MAT), which performs dynamic reference analysis in a location-sensitive manner. We also introduced the concept of a macroblock, which allows the MAT to feasibly characterize the accessed memory locations.

Cycle-by-cycle simulations of the benchmarks show that significant speedups can be achieved by this technique. The hit ratios achieved by the simulated implementation were compared to the hit ratios achieved by the upper bound analysis, and the discrepancies were examined in further detail. This analysis was then used to improve the hardware monitoring heuristics. The comparisons showed that the bypassing implementation often achieved hit ratios close to the upper bounds.

Acknowledgments

The authors would like all the members of the IMPACT research group, whose comments and suggestions helped to improve the quality of this research. We would also like to thank the anonymous referees for their constructive comments. This research has been supported by the National Science Foundation (under grant CCR-9629948), Hewlett-Packard, Advanced Micro Devices, and Intel.

Biographies

Teresa L. Johnson received B.S., M.S., and Ph.D. degrees in electrical engineering from the University of Illinois at Urbana-Champaign in 1993, 1995 and 1998, respectively. She is currently involved in compiler development and research at Hewlett-Packard in Cupertino, California. Her research interests include cache management, and compiler technology for instruction-level parallelism.

Daniel A. Connors is a doctoral student at the University of Illinois, Urbana-Champaign. His current interests include architecture design and compilation techniques for control specula-

tion and predicated execution in EPIC architectures. He has published on instruction-level parallelism, branch prediction, predicated execution, and computation reuse. He received a B.S. in Electrical Engineering from Purdue University and a M.S. from the University of Illinois.

Matthew C. Merten is a doctoral student at the University of Illinois at Urbana-Champaign, where he received his B.S. in computer engineering in 1996 and M.S. in electrical engineering in 1999. His research interests are focused on microarchitecture and on post-link-time optimization including install-time and run-time optimization. He has published in the areas of automatic profiling and adaptive cache management. He is a member of ACM.

Wen-mei W. Hwu (S'81-M'87-F'98) is a Professor at the Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign. From 1997 to 1999, he served as the chairman of the Computer Engineering Program at the University of Illinois. His research interest is in the area of architecture, implementation, and compilation for high performance computer systems. He is the director of the IMPACT project, which has delivered new compiler and computer architecture technologies to the computer industry since 1987. In recognition of his contributions to the areas of compiler optimization and computer architecture, the Intel Corporation named him the Intel Associate Professor at the College of Engineering, University of Illinois in 1992. He received the 1993 Eta Kappa Nu Outstanding Young Electrical Engineer Award, the 1994 Xerox Award for Faculty Research, the 1994 University Scholar Award of the University of Illinois, the 1997 Eta Kappa Nu Holmes MacDonald Outstanding Teaching Award, and the 1998 ACM SigArch Maurice Wilkes Award. He is an IEEE Fellow. Dr. Hwu received his Ph.D. degree in Computer Science from the University of California, Berkeley.

References

[1] K. Boland and A. Dollas, "Predicting and precluding problems with memory latency," *IEEE Micro*, vol. 14, pp. 59–66, August 1994.

[2] C.-K. Luk and T. C. Mowry, "Compiler-based prefetching for recursive data structures," in *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 222–233, September 1996.

[3] S. T. Srinivasan and A. R. Lebeck, "Load latency tolerance in dynamically scheduled processors," in *Proceedings of the 31st International Symposium on Microarchitecture*, pp. 148–159, December 1998.

[4] V. Santhanam, E. H. Gornish, and W.-C. Hsu, "Data prefetching on the HP-8000," in *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 264–273, June 1997.

[5] V. Kathail, M. S. Schlansker, and B. R. Rau, "HPL PlayDoh architecture specification: Version 1.0," Tech. Rep. HPL-93-80, Hewlett-Packard Laboratories, Palo Alto, CA, February 1994.

[6] T. L. Johnson and W. W. Hwu, "Run-time adaptive cache hierarchy management via reference analysis," in *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 315–326, June 1997.

[7] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," in *Proceedings of the 8th International Symposium on Computer Architecture*, pp. 81–87, May 1981.

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

[9] A. J. Smith, "Cache memories," *Computing Surveys*, vol. 14, pp. 473–530, September 1982.

[10] 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.

[11] 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, November 1991.

[12] 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.

[13] S. Mehrotra and L. Harrison, "Quantifying the performance potential of a data prefetch mechanism for pointer-intensive and numeric programs," Tech. Rep. 1458, Center for Supercomputing Research and Development, University of Illinois, November 1995.

[14] J. W. C. Fu, J. H. Patel, and B. L. Janssens, "Stride directed prefetching in scalar processors," in *Proceedings of the 25th Annual Conference on Microprogramming and Microarchitectures*, pp. 102–110, December 1992.

[15] F. Dahlgren, M. Dubois, and P. Strenstrom, "Fixed and adaptive sequential prefetching in shared memory multiprocessors," in *Proceedings of the 1993 International Conference on Parallel Processing*, pp. 56–63, August 1993.

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

[17] A. C. Klaiber and H. M. Levy, "An architecture for software-controlled data prefetching," in *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pp. 43–53, May 1991.

[18] T. C. Mowry, M. S. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," in *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 62–73, Oct. 1992.

[19] 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.

[20] 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*, pp. 104–113, July 1992.

[21] M. H. Lipasti, W. J. Schmidth, S. R. Kunkel, and R. R. Roediger, "SPAID: Software prefetching in pointer- and call-intensive environments," in *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pp. 231–236, December 1995.

[22] A. Roth, A. Moshovos, and G. S. Sohi, "Dependence based prefetching for linked data structures," in *Proceedings of 8th International Conference on Architectual Support for Programming Languages and Operating Systems*, October 1998.

[23] A. Roth and G. S. Sohi, "Effective jump-pointer prefetching for linked data structures," in *To appear in the Proceedings of the 26th International Symposium on Computer Architecture*, May 1999.

[24] J. A. Rivers, E. S. Tam, G. S. Tyson, E. S. Davidson, and M. Farrens, "Utilizing reuse information in data cache management," in *Proceedings of the 12th ACM International Conference on Supercomputing*, July 1998.

[25] R. A. Sugumar and S. G. Abraham, "Efficient simulation of caches under optimal replacement with applications to miss characterization," in *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 24–35, May 1993.

[26] 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.

[27] A. González, C. Aliagas, and M. Valero, "A data cache with multiple caching strategies tuned to different types of locality," in *Proceedings of the 1995 International Conference on Supercomputing*, pp. 338–347, July 1995.

[28] V. Milutinovic, B. Markovic, M. Tomasevic, and M. Tremblay, "The split temporal/spatial cache: Initial performance analysis," in *Proceedings of the SCIzzL-5*, pp. 63–70, March 1996.

[29] 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.

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

[31] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems Journal*, vol. 5, no. 2, pp. 78–101, 1966.

[32] S. McFarling, *Program Analysis and Optimization for Machines with Instruction Cache*. PhD thesis, Stanford University, 1988.

[33] T. L. Johnson, *Run-time Adaptive Cache Management*. PhD thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1998.

[34] 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.

[35] 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.

[36] AMD Corporation, *AMD-K6 Processor Data Sheet*. AMD Corporation, Sunnyvale, CA, Order Number 20695, 1997.