University of Maryland Inistitute for Advanced Computer Studies Technical Report

UMIACS-TR-2001-81

Exploiting Application-Level Information to Reduce Memory Bandwidth Consumption

Abstract

As processors continue to deliver ever higher levels of performance and as memory latency tolerance techniques become widespread to address the increasing cost of accessing memory, memory bandwidth will emerge as a major limitation to continued increases in application performance. In this paper, we propose a hybrid hardware/software technique for addressing the memory bandwidth bottleneck by more intelligently transferring data between the memory system and cache. Our approach uses off-line analysis of the source code and special annotated memory instructions to convey spatial locality information to the hardware at runtime. The memory system uses this information to fetch only the data that will be accessed by the program–data that is unlikely to be referenced is not fetched, hence reducing the application's memory traffic. Our technique uses modified sectored caches to fetch and cache the variable-sized fine-grained data accessed through annotated memory instructions. Our results show that annotated memory instructions remove between 20% and 59% of the cache traffic for 7 applications. Furthermore, annotated memory instructions achieve a 13% performance gain on a cycle-accurate simulator when used alone, and a 26.4% performance gain when combined with software prefetching, compared to a 2.3% performance degradation when prefetching with normal memory instructions.

1 Introduction

The primary focus of memory system research has traditionally been to overcome *memory latency.* However, another important determiner of application performance is *memory bandwidth.* In the past, several researchers have observed that memory bandwidth limitations are responsible for large performance degradations in many important applications. For example, Burger *et al* [2] report between 11% and 31% of the total memory stalls observed in several SPEC benchmarks are due to insufficient memory bandwidth. Perl and Sites [12] find the memory bandwidth required by a commercial SQL Server exceeds the pin bandwidth provided by the Alpha 21164 by a factor of 2. Ding and Kennedy [5] observe several scientific kernels require between 3.4 and 10.5 times the L2-memory bus bandwidth provided by the SGI Origin 2000.

While these testimonials highlight the memory bandwidth bottleneck on existing machines, unfortunately, memory bandwidth limitations are likely to become worse on future high-performance systems due to two factors. First, increased clock rates driven by technology improvements and greater exploitation of ILP will produce processors that consume data at a higher rate. Second, as the gap between processor and memory speeds increases, architects will employ memory latency tolerance techniques, such as prefetching [9, 4], streaming [11], multithreading [16], and speculative loads [14], to address the high cost of accessing memory. These techniques hide memory latency underneath useful work, but they do not reduce memory traffic. Consequently, performance gains achieved through the toleration of memory latency come directly at the expense of increased memory bandwidth consumption. In the absence of sufficient memory bandwidth, memory latency tolerance techniques become ineffective.

These trends will pressure future memory systems to provide increased levels of memory bandwidth in order to realize the potential performance gains of faster processors and aggressive memory latency tolerance techniques. Rather than rely solely on wider and faster memory systems to deliver the required bandwidths, we propose to address the memory bandwidth problem by improving the efficiency of memory bandwidth utilization.

In this paper, we investigate a hybrid hardware/software technique that reduces the memory bandwidth requirement of applications by more intelligently transferring data between the memory system and cache. Our approach relies on the programmer or compiler to extract spatial locality information from the program source code via static analysis. Through special size-annotated memory instructions, the software conveys this information to the hardware, allowing the memory system to fetch only the data that will be accessed by the program on each cache miss. Data that is unlikely to be referenced is not fetched, hence reducing the memory traffic of the application. Finally, hardware support is provided to fetch and cache variable-sized fine-grained data accessed through the annotated memory instructions. We use a sectored cache for this purpose. On a cache miss, our sectored cache fetches a variable number of small cache blocks determined dynamically by each cache-missing memory instruction.

By reducing an application's memory bandwidth requirement, our technique improves performance by reducing memory contention. We also find our technique increases the throughput of serialized memory references since the latency of back-to-back memory accesses is reduced due to the smaller memory transfers. This benefits pointer-intensive codes. In addition to providing performance gains, our technique also offers memory system designers with an alternative to building wider and faster memory systems. Our technique utilizes existing memory resources more effectively, making the memory system appear to deliver more memory bandwidth than it actually does. Although we require additional hardware, this hardware support is modest. Consequently, our technique provides a cost-effective way to address the memory bandwidth bottleneck.

This paper makes the following contributions. First, we present an off-line algorithm for inserting annotated memory instructions to convey spatial locality information to the hardware. Second, we propose using sectored caches to support variable-sized fine-grained data fetched through our annotated memory instructions. Finally, we conduct an experimental evaluation of our technique. Our results show that annotated memory instructions remove between 20% and 59% of the cache traffic for 7 applications. Furthermore, annotated memory instructions achieve a 13% performance gain on a cycle-accurate simulator when used alone, and a 26.4% performance gain when combined with software prefetching, compared to a 2.3% performance degradation when prefetching with normal memory instructions.

The rest of this paper is organized as follows. Sections 2 and 3 present our technique, describing how to insert the annotated memory instructions and the hardware support necessary. Then, Section 4 characterizes the bandwidth reductions our technique achieves, and Section 5 evaluates its performance gains. Finally, Section 6 discusses related work, and Section 7 concludes the paper.

2 Controlling Memory Bandwidth Consumption in Software

Conventional caches exploit spatial locality by fetching long cache blocks on each cache miss. However, this can fetch data wastefully. If the application's memory reference pattern exhibits low spatial locality, portions of a cache block may be unused. Even if spatial locality exists, cache blocks may not remain in cache long enough to capitalize on the spatial reuse. Such wastefully fetched data consume precious memory bandwidth without increasing the number of cache hits. This presents an opportunity: memory bandwidth can be conserved if the memory system avoids fetching data that is unlikely to be referenced by the processor.

2.1 Approach

Our approach enables the application to convey information about its memory access patterns to the memory system so that the transfer size on each cache miss can be customized to match the degree of spatial locality associated with the missing reference. We target memory references that are likely to exhibit *sparse memory access patterns.* For these memory references, the memory system selects a cache miss transfer size smaller than a cache block to avoid fetching useless data, hence reducing the application's memory bandwidth consumption. For all other memory references, the memory system uses the default cache block transfer size to exploit spatial locality.

To provide the application-level information required by our technique, we perform static analysis of the source code to identify memory references that have the potential to access memory sparsely. Our code analysis looks for three traversal patterns that frequently exhibit poor spatial reuse: large-stride affine array traversals, indexed array traversals, and pointer-chasing traversals. For each memory reference in one of these traversals, our analysis extracts a data access size that reflects the degree of spatial locality associated with the memory reference. Sections 2.2 and 2.3 will describe how our code analysis identifies the traversal patterns and extracts the size information.

For each sparse memory reference identified by our code analysis, we replace the original memory instruction at the point in the code where the sparse memory reference occurs with an *annotated memory instruction* that carries a size annotation as part of its opcode. We assume ISA support that provides size annotations for load, store, and prefetch instructions. When an annotated memory instruction suffers a cache miss at runtime, the size annotation is transmitted along with the cache miss request, causing only the amount of data specified by the size annotation to be transferred into the cache rather than transferring an entire cache block. Section 2.4 will present the annotated memory instructions used in our study.

2.2 Identifying Sparse Memory References

Our code analysis examines loops to identify frequently executed memory references that exhibit poor spatial reuse. Specifically, we look for three types of loops that commonly occur in applications: affine array traversal with large stride, indexed array traversal, and pointer-chasing traversal. C code examples of these loops appear in Figure 1.

All three loops in Figure 1 have one thing in common: the memory references executed in adjacent loop iterations access non-consecutive memory locations, giving rise to sparse memory reference patterns. In affine array traversals, a large stride causes consecutively referenced array elements to be separated by a large distance in memory. A large stride can occur in several ways. If the loop induction variable is incremented by a large value each iteration, then using it as an array index results in a large stride. Alternatively, a loop induction variable used to index an outer array dimension also results in a large stride, assuming row-major ordering of array indices. These two cases are illustrated by statements S2 and S3, respectively, in Figure 1a.

Indexed array and pointer-chasing traversals exhibit low spatial locality due to *irregular memory addressing*. In indexed array traversals, a data array is accessed using an index provided by another array, called the “index array.” Statements S4 and S5 in Figure 1b illustrate indexed array references. Since the index for the data array is a runtime value, consecutive data array references often

```
   // a).  Affine Array
   double A[N], B[N][N];
   int i, j;
   int S; /* large */
   for (i=0, j=0;
        i<N; i+=S, j++) {
S1:  prefetch(&A[i+D]);
S2:  ... = A[i];
S3:  ... = B[j][0];
   }
```

```
   // b).  Indexed Array
   double A[N], B[N];
   int C[N];
   int i;
   for (i=0; i<N; i++) {
S3:  prefetch(&A[C[i+D]]);
S4:  ... = A[C[i]];
S5:  B[C[i]] = ...
   }
```

```
   // c).  Pointer-Chasing
   struct node {
     int data;
     struct node *jump, *next;
   } *root, *ptr;
   for (ptr=root; ptr; ) {
S6:  prefetch(ptr->jump);
S7:  ... = ptr->data;
S8:  ptr = ptr->next;
   }
```

Figure 1: Memory references that exhibit sparse memory access patterns frequently occur in three types of loops: a). affine array traversal with large stride, b). indexed array traversal, and c). pointer-chasing traversal.

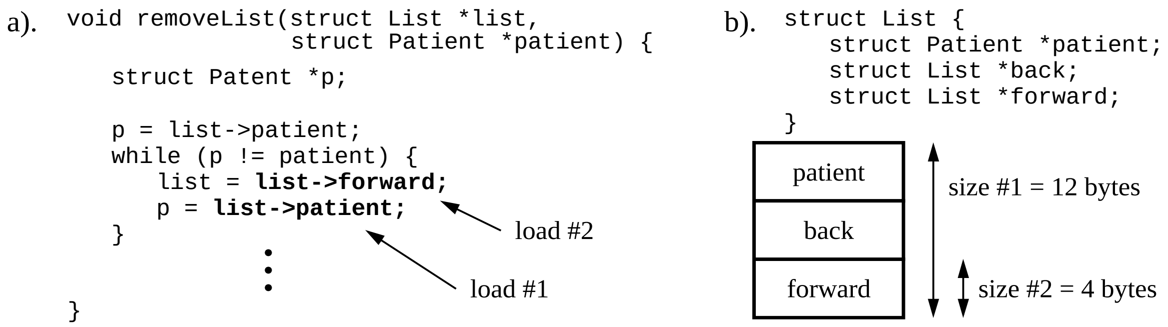
access random memory locations. In pointer-chasing traversals, a loop induction variable traverses a chain of pointer links, as in statements S7 and S8 of Figure 1c. Logically contiguous link nodes are usually not physically contiguous in memory. Even if link nodes are allocated contiguously, frequent insert and delete operations can randomize the logical ordering of link nodes. Consequently, the pointer accesses through the induction variable often access non-consecutive memory locations.

Due to their sparse memory access characteristics, our analysis selects the memory references associated with large-stride affine arrays, indexed arrays, and pointer chain traversals as candidates for our bandwidth-reduction techniques. All memory references participating in these loop traversals are selected. This includes normal loads and stores. It also includes prefetches if software prefetching has been instrumented in the loops to provide latency tolerance benefits. Statements S1, S3, and S6 in Figure 1 illustrate "sparse prefetches" that would be selected by our code analysis.

2.3 Computing Cache Miss Transfer Size

To realize the potential bandwidth savings afforded by sparse memory references, we must determine the amount of data the memory system should fetch each time a sparse memory reference misses in the cache. Proper selection of the cache miss transfer size is crucial. The transfer size should be small to conserve bandwidth. However, selecting too small a transfer size may result in lost opportunities to exploit spatial locality and increased cache misses, offsetting the gains of conserving memory bandwidth. We use code analysis of the memory reference patterns to determine the degree of spatial reuse, and then we select a transfer size that exploits the detected spatial locality.

Our code analysis computes a cache miss transfer size for each sparse memory reference in the following manner. For each array element or link node accessed, we examine the number of unique sparse memory references identified in Section 2.2. If only one sparse memory reference occurs to



```
void removeList(struct List *list,
                struct Patient *patient) {
    struct Patent *p;

    p = list->patient;
    while (p != patient) {
        list = list->forward;
        p = list->patient;
    }
            •
            •
            •
}
```

```
struct List {
    struct Patient *patient;
    struct List *back;
    struct List *forward;
}
```

Figure 2: Extracting size information from compound structures. a). Pointer-chasing loop from the Health benchmark containing two sparse memory references. b). List node structure declaration and memory layout. Vertical arrows indicate size information for the sparse memory references.

each array element or link node, we assume there is no spatial locality and we set the transfer size equal to the size of the memory reference itself. The affine array and indexed array examples in Figures 1a-b fall into this category. The transfer size for each memory reference in these two loops should be set to the size of a double floating point value, which we assume to be 8 bytes.

If, however, each array element or link node is accessed by multiple unique memory references in each loop iteration, then we must determine the degree of spatial locality that exists between intra-iteration references, and select a transfer size that exploits the spatial reuse. This case occurs when an array element or link node contains a compound structure. For example, Figure 2a shows a linked-list traversal loop from Health, a benchmark in the Olden suite [13], in which the loop body references two different fields in the same "List" structure. Because structure elements are packed in memory, separate intra-structure memory references exhibit spatial locality.

To select the transfer size for multiple memory references to a compound structure, we consider each static memory reference in program order. For each static memory reference, we compute the extent of the memory region touched by the memory reference and all other static memory references proceeding it in program order that access the same structure. The size of this memory region is the transfer size for the memory reference. A transfer size computed in this fashion increases the likelihood that each memory reference fetches the data needed by subsequent memory references to the same structure, thus exploiting spatial locality.

Figure 2b demonstrates our transfer size selection algorithm on the Health benchmark. As illustrated in Figure 2a, each "List" structure is referenced twice: the "patient" field is referenced first, and then the "forward" field is referenced second, labeled "load #1" and "load #2," respectively. (Notice the temporal order of references to each structure is inverted compared to the order in which the memory references appear in the source code.) Figure 2b shows the definition of the

	load word	load double	store word	store double	prefetch
8 bytes	lw_8	ld_8	sw_8	sd_8	$pref_8$
16 bytes	lw_{16}	ld_{16}	sw_{16}	sd_{16}	$pref_{16}$
32 bytes	lw_{32}	ld_{32}	sw_{32}	sd_{32}	$pref_{32}$

Table 1: Mneumonics for instructions that carry size annotations. We assume all combinations of load and store word, load and store double word, and prefetch instructions, and 8, 16, and 32 byte annotations.

"List" structure, and illustrates the memory layout of structure elements. We consider the loads in program order. For load #1, we compute the extent of the memory region bounded by both load #1 and load #2 since load #2 follows load #1 in program order. The size of this region is 12 bytes. For load #2, we compute the extent of the memory region consisting of load #2 alone since there are no other accesses to the same structure in program order. The size of this region is 4 bytes. Consequently, loads #1 and #2 should use a transfer size of 12 and 4 bytes, respectively.

2.4 Annotated Memory Instructions

We augment the instruction set with several new memory instructions to encode the transfer size information described in Section 2.3. These annotated memory instructions replace normal memory instructions at the points in the code where sparse memory references have been identified, as described in Section 2.2. When an annotated memory instruction executes at runtime, it passes its size annotation to the memory system, where it is used to reduce the cache miss fetch size. Section 3 will discuss how the hardware uses the size information in greater detail.

Table 1 lists the annotated memory instructions we assume in our study. To minimize the number of new instructions, we restrict the type of memory instructions that carry size annotations. We have found in practice that annotating a few memory instruction types is adequate. In our study, we assume size annotations for load and store word, load and store double word, and prefetch. Each column of Table 1 corresponds to one of these memory instruction types.

We also limit the size annotations to a power-of-two value. In our study, we assume 3 different size annotations: 8, 16, and 32 bytes. Each row of Table 1 corresponds to one of these annotation sizes. Since the number of size annotations is restricted, we cannot annotate a memory reference with an arbitrary size value. Consequently, all transfer sizes computed using the technique described in Section 2.3 must be rounded up to the next available power-of-two size annotation.

2.5 Discussion

In this paper, we perform the code analyses described in Sections 2.2 and 2.3 by hand since our primary goal is to understand the potential gains of our technique. An important question is can these analyses be automated? The most challenging step in the analysis is the identification of sparse memory references. Existing compilers for instrumenting software prefetching in affine loops [9], indexed array loops [10], and pointer-chasing loops [8] already extract this information automatically. Extraction of size information is a simple mechanical process once the sparse memory references have been identified, following the steps in Section 2.3. Consequently, we believe the analyses outlined in this paper are well within the reach of existing compilers, though this assertion is certainly not conclusive until a compiler implementation is undertaken.

3 Hardware Support for Bandwidth Reduction

An annotated memory instruction, as described in Section 2.4, fetches a variable-sized narrow-width block of data on each cache miss. Furthermore, these fine-grained fetches are intermixed with cache misses from normal memory instructions that fetch a full cache block of data. Hardware support is needed to enable the memory hierarchy to handle multiple fetch sizes.

We address variable fetch size introduced by annotated memory instructions in the following manner. First, we reduce the cache block size to match the smallest fetch size required by the annotated memory instructions. Second, we allow a software-specified number of contiguous cache blocks to be fetched on each cache miss. Normal memory instructions should request a fixed number of cache blocks whose aggregate size equals the cache block size of a conventional cache, hence exploiting spatial locality. Annotated memory instructions should request the number of cache blocks required to match the size annotation specified by the instruction opcode, hence conserving memory bandwidth. This section discusses these two hardware issues in more detail.

3.1 Sectored Caches

To exploit the potential bandwidth savings afforded by sparse memory references, a very small cache block size is required. One drawback of small cache blocks is high tag overhead. For example, to support the annotated memory instructions in Table 1, an 8-byte cache block is required. Assuming 32-bit addresses, each tag would contain 29 bits, resulting in a cache tag array that is 45% as large as the cache data array. Fortunately, such high cache tag overheads associated with small cache

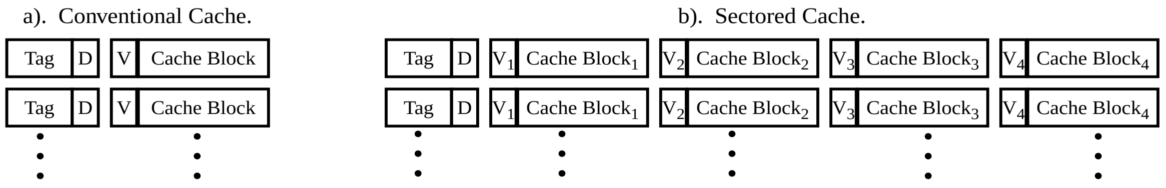

a). Conventional Cache.
b). Sectored Cache.
Tag
D
V
Cache Block
V_1 Cache Block_1
V_2 Cache Block_2
V_3 Cache Block_3
V_4 Cache Block_4

Figure 3: a). A conventional cache provides one tag for each cache block. b). A sectored cache provides one tag for all the cache blocks in a sector, hence reducing the tag overhead. "V" denotes a valid bit, and "D" denotes a dirty bit.

blocks can be mitigated using a *sectored cache*.

Figure 3 illustrates the organizations of conventional and sectored caches. A conventional cache provides a cache tag for every cache block. In contrast, a sectored cache provides a cache tag for every *sector*, which consists of multiple cache blocks that are contiguous in the address space.[1] Each cache block has its own valid bit, so cache blocks from the same sector can be fetched independently. We also assume a single dirty bit is provided for the entire sector. Because each tag is shared between multiple blocks in a sectored cache, cache tag overhead is small even when the cache block size is small. Consequently, sectored caches provide a low-cost implementation of the small cache blocks required by our annotated memory instructions.

3.2 Variable Fetch Size

In addition to providing a small cache block size, the sectored cache must also support fetching a variable number of cache blocks on each cache miss, controlled by software. Figure 4 specifies the actions taken on a cache miss that implements a software-controlled variable fetch size.

Normally, a sectored cache fetches a single cache block on every cache miss. To support our technique, we should instead choose the number of cache blocks to fetch based on the opcode of the memory instruction at the time of the cache miss. Our sectored cache performs three different actions on a cache miss depending on the type of cache-missing memory instruction, as illustrated in Figures 4a-c. Moreover, each action is influenced by the type of miss. Sectored caches have two different cache-miss types: a *sector miss* occurs when both the requested cache block and sector are not found in the cache, and a *cache block miss* occurs when the requested sector is present (*i.e.* a sector hit) but the requested cache block within the sector is missing.

When a normal memory instruction suffers a cache miss (Figure 4a), the cache requests an

[1]Referring to each group of blocks as a *sector* and individual blocks as *cache blocks* is a naming convention used by recent work in sectored caches [7]. In the past, these have also been referred to as *cache block* and *sub-blocks*, respectively, and the overall technique as *sub-blocking*. We choose to use the more recent terminology in our paper, though there is no general agreement on terminology.

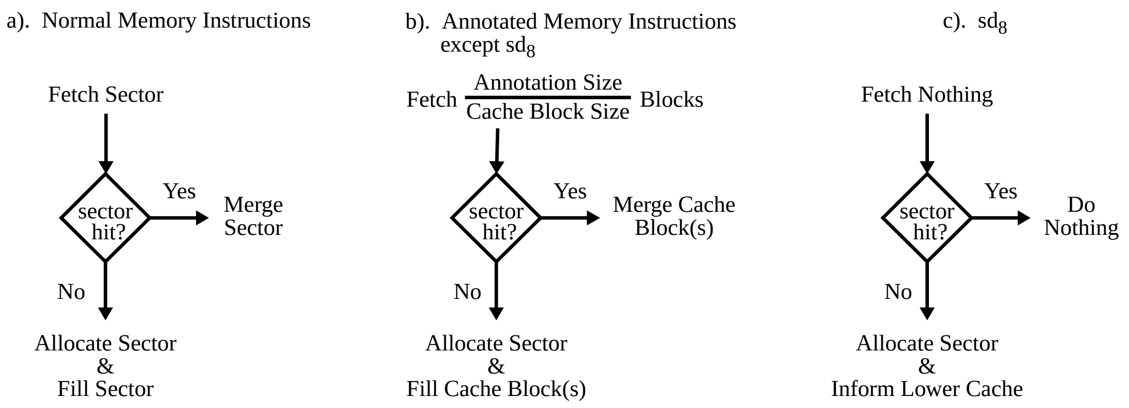

a). Normal Memory Instructions
Fetch Sector
sector hit?
Yes
Merge Sector
No
Allocate Sector & Fill Sector
b). Annotated Memory Instructions except sd$_8$
Fetch $\frac{\text{Annotation Size}}{\text{Cache Block Size}}$ Blocks
sector hit?
Yes
Merge Cache Block(s)
No
Allocate Sector & Fill Cache Block(s)
c). sd$_8$
Fetch Nothing
sector hit?
Yes
Do Nothing
No
Allocate Sector & Inform Lower Cache

Figure 4: Action taken on a cache miss for a). normal memory instructions, b). annotated memory instructions except sd_8, and c). sd_8.

entire sector of data from the next level of the memory hierarchy. For sector misses, a new sector is also allocated, evicting an existing sector if necessary. When the requested sector arrives, it is filled into the sector if the miss was a sector miss. Otherwise, if the miss was a cache block miss, a "merge" operation is performed instead. The merge fills only those cache blocks inside the sector that are invalid. All other cache blocks from the fetched sector are discarded to prevent overwriting already present (and possibly dirty) cache blocks in the sector.

When an annotated memory instruction suffers a cache miss (Figure 4b), the cache requests the number of cache blocks specified by the instruction's size annotation, or $\frac{Annotation\_Size}{Cache\_Block\_Size}$. Since we restrict the annotation size to a power of two, this ratio will itself be a power of two (assuming cache blocks are a power-of-two size as well). In the event that the annotation is smaller than a cache block or larger than a sector, we fetch a single cache block or sector, respectively. Also, we align the request to an annotation-sized boundary (*i.e.* 8-byte annotations are double-word aligned, 16-byte annotations are quad-word aligned, etc.). These simplifying assumptions guarantee that all fetched cache blocks reside in the same sector. Eventually, the variable-sized fetch request returns from the next level of the memory hierarchy. If the miss was a sector miss, we allocate a sector and fill the requested cache blocks into the new sector. If the miss was a cache block miss, we merge the requested cache blocks into the already present sector.

Finally, there is one exception to this case, shown in Figure 4c. For annotated store instructions whose store width matches the size annotation, the store instruction itself overwrites the entire region specified by the size annotation. Consequently, there is no need to fetch data on a cache miss. Notice however, if the store miss is a sector miss, the cache must allocate a new sector for the store, which can violate inclusion if a fetch request is not sent to the next memory hierarchy level.

Program	Input	Sparse Refs	Init	Warm	Data
Irreg	144K nodes, 20K itrs	Indexed array	993.1	5.2(3.9)	5.2(3.9)
Moldyn	131K mols, 14K itrs	Indexed array	761.5	6.7(3.3)	6.7(3.3)
NBF	144K mols, 6.4K itrs	Indexed array	49.6	4.2(2.1)	4.2(2.1)
Health	5 levels, 5 itrs	Ptr-chasing	160.7	0.5(0.27)	7.6(4.2)
MST	1024 nodes, 100 itrs	Ptr-chasing	197.2	9.8(3.7)	3.9(1.5)
BZIP2	"ref" input	Indexed array	147.2	77.3(22.2)	18.5(5.25)
MCF	"ref" input	Affine array, Ptr-chasing	6909.5	1.3(0.54)	5.1(2.17)

Table 2: Benchmark summary. The first three columns report the name, the data input set, and the source(s) of sparse memory references for each benchmark. The last three columns report the number of instructions (in millions) in the initialization, warm up, and data collection phases. Values in parentheses report data reference counts (in millions).

For this case, there is still no need to fetch data, but the next level of the memory hierarchy should be informed of the miss so that inclusion can be maintained. Amongst the annotated memory instructions used in our study (see Table 1), sd_8 is the only one for which this exception applies.

4 Cache Behavior Characterization

Having described our technique in Sections 2 and 3, we now evaluate its effectiveness. We choose cache simulation as the starting point for our evaluation because it permits us to study the behavior of annotated memory instructions independent of system implementation details.

4.1 Evaluation Methodology

Table 2 presents our benchmarks. The first three make heavy use of indexed arrays. Irreg is an iterative PDE solver for computational fluid dynamics problems. Moldyn is abstracted from the non-bonded force calculation in CHARMM, a molecular dynamics application. And NBF is abstracted from the GROMOS molecular dynamics code [17]. The next two benchmarks are from Olden [13]. Health simulates the Columbian health care system, and MST computes a minimum spanning tree. Both benchmarks traverse linked lists frequently. Finally, the last two benchmarks are from SPECInt CPU2000. BZIP2 is a data compression algorithm that performs indexed array accesses, and MCF is an optimization solver that traverses a highly irregular linked data structure.

Using these benchmarks, we perform a series of cache simulations designed to characterize the benefits and drawbacks of annotated memory instructions. We first compare our technique against a conventional cache to quantify the memory traffic reductions afforded by annotated memory instructions. We then compare our technique against a perfectly managed cache to study the limits on achievable traffic reduction. Finally, we compare our technique against an existing hardware-

Cache Model	Sector Size	Block Size	Associativity	Replacement Policy	Write-Hit Policy
Conventional	-	32 bytes	2-way	LRU	Write-back
Annotated	32 bytes	8 bytes	2-way	LRU	Write-back
MTC	-	4 bytes	full	MIN	Write-back
SFP	32 bytes	8 bytes	2-way	LRU	Write-back

Table 3: Baseline cache model parameters used in our cache simulations.

based prefetching cache, called Spatial Footprint Predictors (SFP) [7].

We modified SimpleScalar v3.0's cache simulator to provide the cache models necessary for our experiments. We added sectored caches and augmented the PISA ISA with the annotated memory instructions in Table 1 to model our technique. Our technique also requires instrumenting annotated memory instructions for each benchmark. We followed the algorithms in Sections 2.2 and 2.3 for identifying sparse memory references and computing size annotations, and then inserted the appropriate annotated memory instructions into the application assembly code. All instrumentation was performed by hand.

Next, we implemented Burger's minimal traffic cache (MTC) [2]. MTC is fully associative, uses a write-back policy, employs a 4-byte cache block, and bypasses the cache for low-priority loads. Furthermore, MTC uses Belady's MIN replacement policy [1], which selects for replacement the block that will be referenced furthest in the future. We use MTC to provide an aggressive lower bound on traffic for our limit study in Section 4.3. Finally, we implemented an SFP cache. We will describe the details of our SFP cache later in Section 4.4.

Our simulations consist of three phases. First, we identified each benchmark's initialization code. During this code, our simulators perform functional simulation only. Next, our simulators turn on the appropriate cache model, and continue simulating to warm up the cache. Then, data collection is turned on to acquire statistics. In Table 2, the columns labeled "Init," "Warm," and "Data" report the length of the initialization, warm up, and data collection phases, respectively. Finally, Table 3 presents the baseline cache parameters for the cache models used in our experiments. Note the sector size belonging to the sectored caches (for both our technique and SFP) is set to the cache block size of the conventional cache, 32 bytes, to facilitate a meaningful comparison.

4.2 Traffic and Miss Rate Characterization

Figure 5 plots cache traffic as a function of cache size for a conventional cache (Conventional), a sectored cache using annotated memory instructions (Annotated), an MTC (MTC), and two SFP caches (SFP-Ideal and SFP-Real). We report traffic to the next memory hierarchy level for each cache, including fetch and write-back traffic but excluding address traffic. Cache size is varied from 1 Kbyte to 1 Mbyte in powers of two; all other cache parameters use the baseline values in Table 3.

Comparing the Annotated and Conventional curves in Figure 5, we see that annotated memory instructions reduce cache traffic significantly compared to a conventional cache. For NBF, HEALTH, MST, and MCF, annotated memory instructions reduce cache traffic by roughly one half, between 40% and 59%. Furthermore, these percentage reductions are fairly constant across all cache sizes, indicating that our technique is effective in both small and large caches for these benchmarks. For IRREG, annotated memory instructions are effective at small cache sizes, reducing traffic by 45% or more for caches 32K or smaller, but lose their effectiveness at large cache sizes. IRREG performs accesses to a large data array through an index array. Temporally related indexed references are sparse, but over time, the entire data array is referenced. Large caches can exploit the spatial locality between temporally distant indexed references because cache blocks remain in cache longer. As the exploitation of spatial locality increases in IRREG, annotated memory instructions lose their advantage. Finally, for MOLDYN and BZIP2, the traffic reductions are more modest–18% and 22% averaged over all cache sizes. The memory reference patterns in MOLDYN lack sparsity, providing less opportunity to reduce traffic. The behavior of BZIP2 will be explained later in Section 4.3.

Figure 6 plots cache miss rate as a function of cache size for the "Conventional," "Annotated," and "SFP-Ideal" caches in Figure 5. Comparing the Annotated and Conventional curves in Figure 6, we see that the traffic reductions achieved by annotated memory instructions come at the expense of increased cache miss rates. Miss rate increases range between 3.2% and 22.8% for MOLDYN, NBF, MST, BZIP2, and MCF, and roughly 40% for IRREG and HEALTH.

The higher miss rates incurred by annotated memory instructions are due to the inexact nature of our spatial locality detection algorithm described in Section 2.3. For indexed array and pointer-chasing references, we perform analysis only between references within a single compound structure, *i.e.* within a single loop iteration. Our technique does not detect spatial locality between references in different loop iterations because the separation of such inter-iteration references depends on runtime values that are not available statically. Hence, our size annotations are overly conservative, missing opportunities to exploit spatial locality whenever multiple indirect references through index arrays or pointers coincide in the same sector. Fortunately, as we will see in Section 5, the benefit of traffic reduction usually outweighs the increase in miss rate, resulting in performance gains.

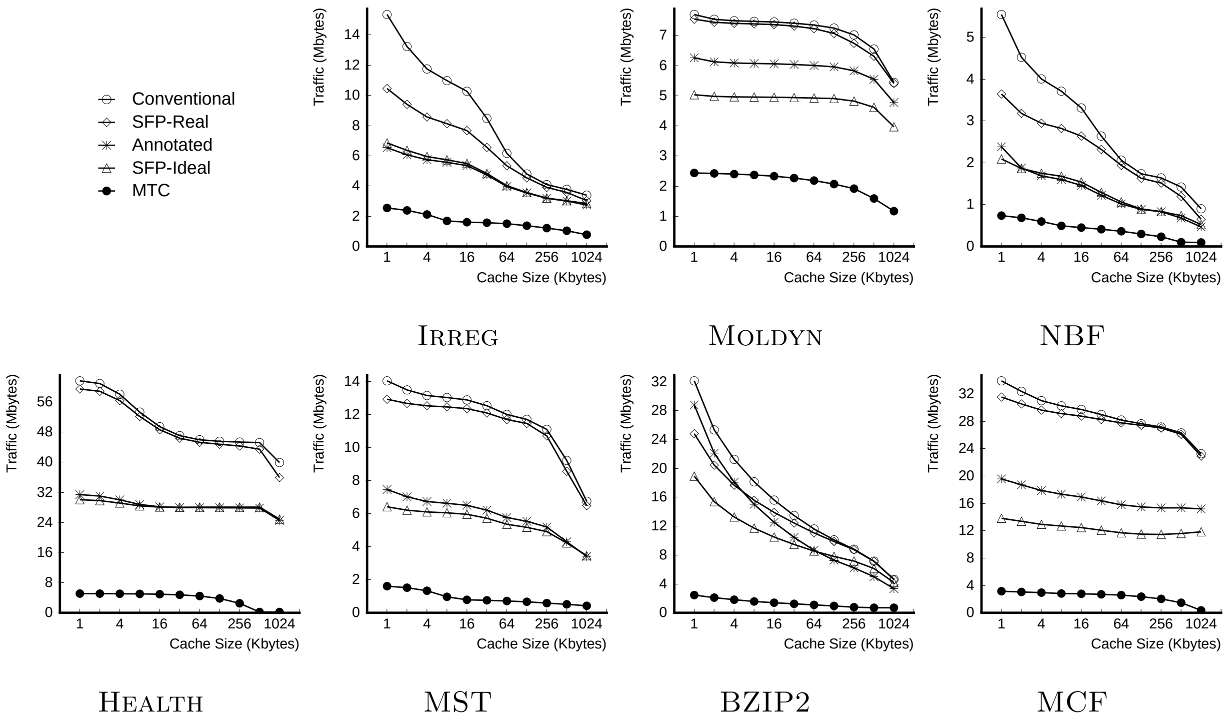


Figure 5: Cache traffic as a function of cache size. Traffic is reported for a conventional cache (Conventional), a sectored cache with annotated memory instructions (Annotated), an MTC (MTC), and two SFP caches (SFP-Ideal and SFP-Real). All traffic values are in units of MBytes.

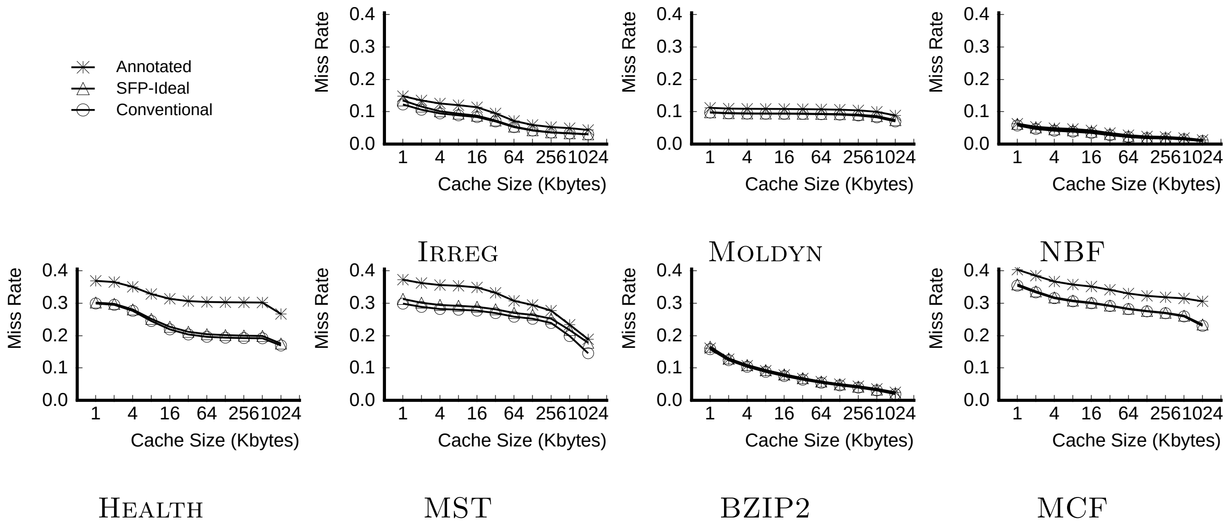


Figure 6: Cache miss rate as a function of cache size. Miss rates are reported for a conventional cache (Conventional), a sectored cache with annotated memory instructions (Annotated), and an SFP cache (SFP-Ideal).

4.3 Limits on Traffic Reduction

Comparing the Annotated and MTC curves in Figure 5, we see that MTC achieves significantly lower traffic than our technique. For Irreg, Moldyn, and NBF, our technique generates roughly 3 times (between 2.9 and 3.6) more traffic than MTC averaged over all cache sizes. For the other 4 applications, the average increase goes up to roughly 7 times (between 6.7 and 8.6). In this section, we study this traffic disparity, and quantify the limits on the additional gain of our technique.

To facilitate our study, we modified our cache simulator to measure the traffic of the best sectored cache achievable, which we call an "oracle cache." An oracle cache fetches only those cache blocks on a sector miss that the processor will reference during the sector's lifetime in the cache. To measure oracle cache traffic, we maintain a bit mask for each sector, 1 bit for every cache block in the sector, that tracks all cache blocks referenced by the processor while the sector is resident in cache. When the sector is evicted, we update the write-back traffic as normal, but we also update the fetch traffic "retroactively" by examining the bit mask to determine which cache blocks the oracle cache *would have fetched*.

Using the oracle cache simulator, we run two simulations. Both use the "Annotated" parameters in Table 3, except that one simulation uses an 8-byte cache block while the other uses a 4-byte cache block–"Oracle8" and "Oracle4," respectively. Since an oracle cache is optimal for a given configuration, Oracle8's traffic is the lowest traffic our technique can possibly achieve using an 8-byte cache block. The additional traffic reduced by Oracle4 compared to Oracle8 represents the improvement achieved using a 4-byte cache block. Finally, the traffic disparity between MTC and Oracle4 quantifies the benefits of ideal cache organization and omnicient cache management employed by MTC. Figure 7 breaks down the traffic disparity between our technique and MTC from Figure 5 by reporting the incremental traffic reduction of Oracle8, Oracle4, and MTC, normalized to the total traffic disparity. Breakdowns are shown for 8K, 64K, and 512K cache sizes.

Figure 7 shows that for Irreg, NBF, and Health, our technique essentially achieves the minimum traffic since the "Oracle8" components are negligible. However, for Moldyn, MST, and MCF, the Oracle8 components are roughly 30%, and for BZIP2, about 50%, indicating our technique can be improved for these applications. Two factors contribute to these Oracle8 components. First, our annotated memory instructions often fetch data unnecessarily. Rounding up annotations to the nearest power-of-two size (Section 2.4), and alignment of cache misses to an annotation-sized boundary (Section 3.2) result in unnecessary fetches. Second, our technique is effective only for

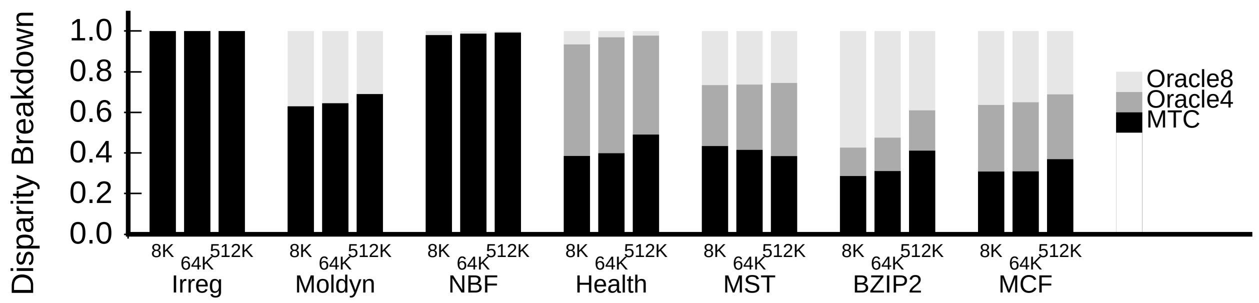


Figure 7: Breakdown of the traffic disparity between our technique and MTC in Figure 5 for 8K, 64K, and 512K cache sizes. The disparity is broken down into 3 components labeled “Oracle8,” “Oracle4,” and “MTC.”

the access patterns described in Section 2.2. Other sparse memory reference patterns do not get optimized. In BZIP2, for example, most sparse memory references occur in unit-stride affine array loops. Normally, such loops access memory densely, but in BZIP2, these loops execute only 1 or 2 iterations, resulting in sparse accesses. Figure 7 also shows significant “Oracle4” components for HEALTH, MST, BZIP2, and MCF. These applications perform sparse references to word-size data; hence, using 4-byte cache blocks can benefit these applications. Finally, the MTC components are the largest, and are particularly significant for IRREG, MOLDYN, and NBF.

We conclude the following from Figure 7. First, on a sectored cache with 8-byte cache blocks, our technique gets most of the *achievable* traffic reduction gain. Further improvements are possible, perhaps by using more sophisticated annotations to eliminate unnecessary fetches, and by extending our analysis to increase coverage. Second, smaller cache blocks can provide improvements. However, this requires using a 4-byte cache block which may be impractical on some systems since the cache block size cannot be smaller than the memory bus width. Finally, there are large gains in the MTC components, but our technique cannot exploit these traffic reductions since they are a consequence of better cache organization and management policies, rather than better fetch policies.

4.4 Spatial Footprint Predictors

Spatial Footprint Predictors (SFP) perform prefetching into a sectored cache using a Spatial Footprint History Table (SHT). The SHT maintains a history of cache block “footprints” within a sector. Much like the bitmasks in our oracle cache simulator, each footprint records the cache blocks referenced within a sector during the sector’s lifetime in the cache. The SHT stores all such footprints for every load PC and cache-missing address encountered. On a sector miss, the SHT is consulted to predict those cache blocks in the sector to fetch. If no footprint is found in the SHT, the entire sector is fetched. Our SFP cache models the $SFP_1^{IA,DA}$ configuration in [7].

The "SFP-Ideal" curves in Figure 5 report the traffic of an SFP cache using a 2M-entry SHT. Assuming 4-byte SHT entries, this SHT is 8 Mbytes, essentially infinite for the workloads we simulated. Figure 5 shows annotated memory instructions achieve close to the same traffic as SFP-Ideal for Irreg, NBF, Health, and MST. For Moldyn and MCF, however, SFP-Ideal reduces 29.4% and 35.7% more traffic, respectively, than our technique. Finally, in BZIP2, SFP-Ideal outperforms annotated memory instructions at small cache sizes, but is worse at large cache sizes. Figure 5 demonstrates our technique achieves comparable traffic with an aggressive SFP despite using much less hardware. Comparing miss rates, however, Figure 6 shows that SFP-Ideal outperforms our technique, essentially matching the miss rate of a conventional cache.

The "SFP-Real" curves in Figure 5 report the traffic of an SFP using an 8K-entry SHT. The SHT in SFP-Real is 32 Kbytes. Figure 5 shows that SFP-Real is unable to reduce any traffic for Moldyn, Health, MST, and MCF. In Irreg, NBF, and BZIP2, small traffic reductions are achieved, but only at small cache sizes. In all cases, our technique achieves larger traffic reductions compared to SFP-Real. The large working sets of our workloads give rise to a large number of unique footprints. A 32K SHT lacks the capacity to store these footprints, so it frequently fails to provide predictions, missing traffic reduction opportunities.

We note that the original SFP cache in [7] used a 128-byte sector, but in this study, we use a 32-byte sector to facilitate a comparison against our other caches. We also ran simulations to compare our technique against SFP using 64 and 128-byte sectors, and found that larger sectors result in increased cache traffic; however, the *relative* traffic between our technique and SFP is unaffected by sector size.

5 Performance Evaluation

This section continues our evaluation of annotated memory instructions by measuring performance on a detailed cycle-accurate simulator.

5.1 Simulation Environment

Like the cache simulators from Section 4, our cycle-accurate simulator is also based on SimpleScalar v3.0. We use SimpleScalar's out-of-order processor module without modification, configured to model a 1 GHz dynamically scheduled 4-way issue superscalar. We also simulate a two-level cache

Processor Model: 1 cycle = 1.0 ns			
Issue Width	4	Integer Latency	1 cycle
Instruction Window Size	32	Floating Add/Mult/Div Latency	2/4/12 cycles
Load-Store Queue Size	8	Branch Predictor	gshare
Fetch Queue Size	4	Branch Predictor Size	2048 entries
Integer/Floating Point Units	4/4	BTB Size	2048 entries
Cache Model: 1 cycle = 1.0 ns			
L1/L2 Cache Size	16K-split/512K-unified	L1/L2 Associativity	2/4 cycles
L1/L2 Sector Size	32/64 bytes	L1/L2 Latency	1/10 cycles
L1/L2 Cache Block Size	8/8 bytes	L1/L2 MSHRs	32/32
Memory Sub-System Model			
DRAM Banks	64	Row Access Strobe	14 ns
Memory System Bus Width	8 bytes	Column Access Strobe	14 ns
Address Send	8 ns	Data Transfer (per 8 bytes)	8 ns

Table 4: Simulation parameters for the processor, cache, and memory sub-system models. Latencies are reported either in processor cycles or in nanoseconds. We assume a 1-ns processor cycle time.

hierarchy. Our cycle-accurate simulator implements the "Conventional" and "Annotated" cache models from Section 4 only. We do not model the SFP cache. For our technique, we use sectored caches at both the L1 and L2 levels. The top two portions of Table 4 list the parameters for the processor and cache models used in all our simulations.

Our simulator faithfully models a memory controller and DRAM memory sub-system. Each L2 request to the memory controller simulates several actions: queuing of the request in the memory controller, RAS and CAS cycles between the memory controller and DRAM bank, and data transfer across the memory system bus. We simulate concurrency between DRAM banks, but bank conflicts require back-to-back DRAM accesses to perform serially. When the L2 cache makes a request to the memory controller, it specifies a transfer size along with the address (as does the L1 cache when requesting from the L2 cache), thus enabling a variable-sized transfer to support annotated memory instructions. Finally, our memory controller always fetches the critical-word first for both normal and annotated memory accesses. The bottom portion of Table 4 lists the parameters for our baseline memory sub-system model. These parameters correspond to a 100 ns (100 processor cycles) L2 sector fill latency and a 1 GB/s memory system bus bandwidth.

Our memory sub-system model simulates contention, but we assume infinite bandwidth between the L1 and L2 caches. Consequently, the cache traffic reductions afforded by annotated memory instructions benefit the memory sub-system only (though the cache miss increases impact both the L1 and L2). We expect traffic reductions across the L1-L2 bus provided by annotated memory

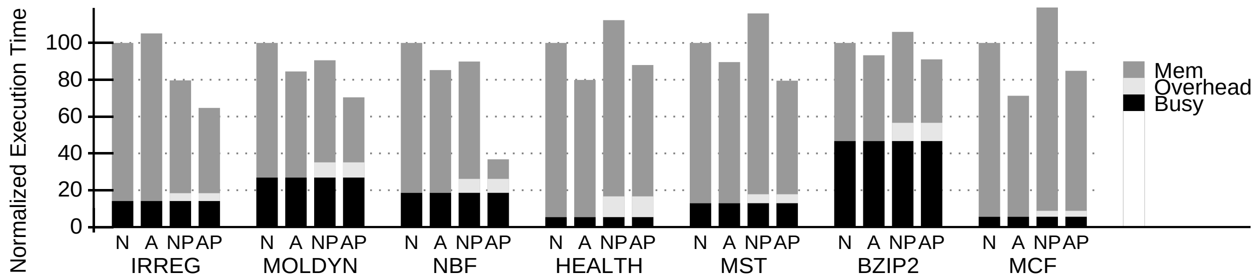


Figure 8: Execution time breakdown for annotated memory instructions. Individual bars show performance without prefetching using normal and annotated memory instructions, labeled "N" and "A", and performance with prefetching using normal and annotated memory instructions, labeled "NP" and "AP."

instructions can also increase performance, but our simulator does not quantify these effects.

Our evaluation considers the impact of annotated memory instructions on software prefetching, so we created software prefetching versions of our benchmarks. For affine array and indexed array references, we use the prefetching algorithms in [9] and [10], respectively. For pointer-chasing references, we use the prefetch arrays technique [6]. Instrumentation of annotated memory instructions for prefetch, load, and store instructions occurs after software prefetching has been applied.

5.2 Performance of Annotated Memory Instructions

Figure 8 shows the performance of our annotated memory instructions on the baseline memory system described in Section 5.1. Each bar in Figure 8 reports the normalized execution time for one of four versions for each application: without prefetching using normal and annotated memory instructions ("N" and "A" bars), and with prefetching using normal and annotated memory instructions ("NP" and "AP" bars). Each execution-time bar has been broken down into three components: useful computation, prefetch-related software overhead, and memory stall, labeled "Busy," "Overhead," and "Mem," respectively. "Busy" is the execution time of the "N" version assuming a perfect memory system (*e.g.* all memory accesses complete in 1 cycle). "Overhead" is the incremental increase in execution time of the "NP" and "AP" versions over "Busy," again on a perfect memory system. "Mem" is the incremental increase in execution time over "Busy"+"Overhead" assuming a real memory system. All times are normalized against the "N" bars.

First, we examine performance without prefetching. Comparing the "N" and "A" bars in Figure 8, we see that annotated memory instructions increase performance for 6 out 7 applications, reducing execution time by as much as 28.7% (MCF), and by 13% on average. The cache traffic reductions of our technique reported in Section 4.2 result in two performance benefits. First, reduced cache traffic lowers contention in the memory system. For applications that generate simultaneous misses to memory, this lowers the effective cache miss penalty. Second, reduced cache traffic benefits pointer-intensive applications (Health, MST, and MCF). Pointer-chasing loops suffer serialized cache misses; hence, their throughput is dictated by the latency of back-to-back cache misses. Because annotated memory instructions transfer less data, they experience a lower cache miss latency (*e.g.* on our simulator, filling an L2 cache block takes only 44 cycles, compared to 100 cycles for filling an L2 sector), thus increasing the throughput of pointer-chasing loops.

Recall from Section 4 that annotated memory instructions increase the cache miss rate due to reduced exploitation of spatial locality. Figure 8 demonstrates that the benefit of reduced cache traffic outweighs the increase in cache miss rate, resulting in a net performance gain for most applications. Irreg is the one exception. As discussed in Section 4.2, annotated memory instructions do not provide a significant traffic reduction for Irreg at large cache sizes. Hence, the increased cache miss rate results in a 5.2% performance loss for Irreg. Although Figure 5 shows Irreg enjoys a much larger traffic reduction at small cache sizes, this does not benefit Irreg in our simulator since we do not simulate contention across the L1-L2 bus.

Next, we examine prefetching performance. Comparing the "NP" and "N" bars in Figure 8, we see that software prefetching with normal memory instructions degrades performance for 4 applications (Health, MST, BZIP2, and MCF), resulting in a 2.3% degradation averaged across all benchmarks. Prefetching adds software overhead, and can increase memory traffic due to speculative prefetches. The 1 GB/s bandwidth of our baseline memory sub-system is insufficient for software prefetching to tolerate enough memory latency in these applications to offset the overheads.

Comparing the "AP" and "N" bars, however, we see that software prefetching with annotated memory instructions achieves a performance gain for all 7 applications, 26.4% on average. Annotated memory instructions are particularly effective when coupled with software prefetching. The addition of prefetching increases memory contention, thus magnifying the importance of reduced traffic provided by annotated memory instructions. Also, the increase in cache miss rate incurred by annotated memory instructions is less important when performing prefetching because the additional cache misses will themselves get prefetched, thus hiding their latency.

5.3 Bandwidth Sensitivity

This section examines the sensitivity of our previous results to memory bandwidth. We run the "N," "A," "NP," and "AP" versions from Figure 8 using higher memory system bus bandwidths.

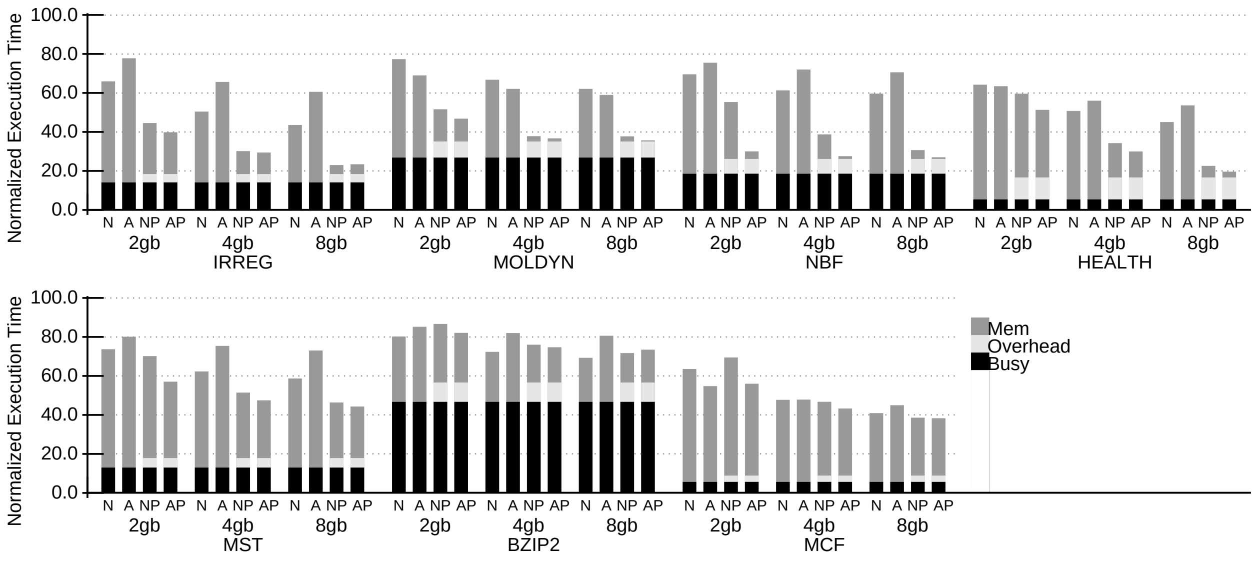


Figure 9: Execution time breakdown for annotated memory instructions at 2, 4, and 8 GB/s. Individual bars show performance without prefetching using normal and annotated memory instructions, labeled "N" and "A" respectively, and performance with prefetching using normal and annotated memory instructions, labeled "NP" and "AP" respectively.

We increase the memory system bus bandwidth by decreasing the "Data Transfer" parameter in Table 4, or the time to perform a single 8-byte transfer across the memory system bus. All other parameters in Table 4 are held constant.

In Figure 9, we report performance at 2, 4, and 8 GB/s memory system bus bandwidths for all the applications and all versions. All bars have been normalized against the "N" versions at 1 GB/s from Figure 8. Without prefetching (*i.e.* comparing the "N" and "A" bars), Figure 9 shows that 4 of our applications (Irreg, NBF, MST, and BZIP2) perform worse at 2 GB/s with annotated memory instructions, and all applications except for Moldyn perform worse by 8 GB/s. As available memory bandwidth increases, there is less opportunity to reduce memory contention and data transfer latency; hence, our technique loses its benefit, and the increased cache miss rates reported in Section 4.2 result in performance degradations.

With prefetching, (*i.e.* comparing the "NP" and "AP" bars), we see that again annotated memory instructions have a reduced performance advantage as memory bandwidth increases. However, our technique always achieves higher or equal performance compared to normal memory instructions, with the exception of BZIP2 at 8 GB/s where we suffer a degradation of 2%. When combined with software prefetching, the performance of our technique is robust to available bandwidth because prefetching makes annotated memory instructions resilient to increased cache misses. This result suggests our technique is quite effective in combination with software prefetching.

Although annotated memory instructions become less effective with increasing memory bandwidth, we note that most memory systems today are closer to our baseline memory system (1 GB/s) rather than our high-end memory system (8 GB/s). More importantly, we believe behavior on future systems will more closely resemble our baseline memory system results in Figure 8 given the trends in processor and memory speeds.

6 Related Work

Our work is most similar to Spatial Footprint Predictors (SFP) [7]. Like SFP, we use a sectored cache and variable-size fetches. However, instead of using hardware predictors, our approach uses static analysis of the source code to identify what to fetch. The advantage of SFP is that it does not require compiler or programmer support, and it can exploit runtime information to identify sparse memory reference patterns. On the other hand, the advantage of annotated memory instructions is that it does not require a hardware prediction table, significantly reducing hardware cost.

Other techniques have exploited application-level information to improve memory performance. Impulse [3] provides address translation in the memory controller to alter the layout of data structures in memory, improving both cache utilization and memory bandwidth consumption. Compared to Impulse, our approach only requires the software to provide hints to the memory system, whereas Impulse requires code transformations to alias the original and remapped memory locations whose correctness must be verified. However, our approach only reduces memory bandwidth consumption. Impulse's memory remapping also improves cache utilization.

Temam and Drach [15] propose Virtual Cache Lines (VCL), a technique that uses a modest cache block size for normal memory references (32 bytes), and fetches multiple sequential cache blocks on a cache miss when the compiler has detected spatial locality beyond a single cache block. Compared to VCL, we use a finer-grained baseline fetch size (8 bytes) and more flexible software control to reduce memory bandwidth consumption for sparse memory references. VCL instead focuses on increasing the cache hit rate for applications with spatial locality.

In contrast to hardware-based techniques, Ding and Kennedy [5] propose compiler optimizations to reduce memory traffic. These techniques do not require hardware support; however, they are applicable only when the correctness of the code transformations can be guaranteed. Our approach reduces memory bandwidth even for loops that are not amenable to code transformations.

7 Conclusion

This paper presents a hybrid hardware/software technique that enables software to control its own memory bandwidth consumption through annotated memory instructions. Our results show that annotated memory instructions remove between 20% and 59% of the cache traffic for 7 applications. These traffic reductions, however, come at the expense of increased cache miss rates, ranging between 3% and 23% for 5 applications, and roughly 40% for 2 applications. Using a cycle-accurate simulator, we find that annotated memory instructions achieve a 13% performance gain when used alone, and a 26.4% performance gain when combined with software prefetching, compared to a 2.3% performance degradation when prefetching with normal memory instructions. As available memory bandwidth increases, our technique loses its advantage, though annotated software prefetching maintains a performance gain at all bandwidths due to its resilience to increased cache misses.

On future systems where processor and memory speeds continue to diverge and latency tolerance techniques become widespread, memory bandwidth will become exposed as a key determiner of application performance. Our work provides memory systems designers in that environment with a cost-effective approach for addressing the memory bandwidth bottleneck.

References

[1] L. A. Belady. A Study of Replacement Algorithms for a Virtual-Storage Computer. *IBM Systems Journal*, 5(2):78–101, 1996.

[2] Doug Burger, James R. Goodman, and Alain Kagi. Memory Bandwidth Limitations of Future Microprocessors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 78–89, Philadelphia, PA, May 1996. ACM.

[3] J.B. Carter, W.C. Hsieh, L.B. Stoller, M.R. Swanson, L. Zhang, E.L. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M.A. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a Smarter Memory Controller. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, pages 70–79, January 1999.

[4] Tien-Fu Chen and Jean-Loup Baer. Effective Hardware-Based Data Prefetching for High-Performance Processors. *Transactions on Computers*, 44(5):609–623, May 1995.

[5] Chen Ding and Ken Kennedy. Memory Bandwidth Bottleneck and its Amelioration by a Compiler. In *Proceedings of the International Parallel and Distributed Processing Symposium*, Cancun, Mexico, May 2000.

[6] Magnus Karlsson, Fredrik Dahlgren, and Per Stenstrom. A Prefetching Technique for Irregular Accesses to Linked Data Structures. In *Proceedings of the 6th International Conference on High Performance Computer Architecture*, Toulouse, France, January 2000.

[7] Sanjeev Kumar and Christopher Wilkerson. Exploiting Spatial Locality in Data Caches using Spatial Footprints. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 357–368, Barcelona, Spain, June 1998. ACM.

[8] Chi-Keung Luk and Todd C. Mowry. Compiler-Based Prefetching for Recursive Data Structures. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, Cambridge, MA, October 1996. ACM.

[9] Todd Mowry. Tolerating Latency in Multiprocessors through Compiler-Inserted Prefetching. *Transactions on Computer Systems*, 16(1):55–92, February 1998.

[10] Todd C. Mowry. Tolerating Latency Through Software-Controlled Data Prefetching, PhD Thesis. Technical report, Stanford University, March 1994.

[11] Subbarao Palacharla and R. E. Kessler. Evaluating Stream Buffers as a Secondary Cache Replacement. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 24–33, Chicago, IL, April 1994. ACM.

[12] Sharon E. Perl and Richard L. Sites. Studies of Windows NT Performance Using Dynamic Execution Traces. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, October 1996.

[13] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren. Supporting Dynamic Data Structures on Distributed Memory Machines. *ACM Transactions on Programming Languages and Systems*, 17(2), March 1995.

[14] Anne Rogers and Kai Li. Software Support for Speculative Loads. In *ASPLOS-V*, pages 38–50. ACM, October 1992.

[15] O. Temam and N. Drach. Software-Assistance for Data Caches. In *Proceedings of the First Annual Symposium on High-Performance Computer Architecture*, Raleigh, NC, January 1995. IEEE.

[16] Dean Tullsen, Susan Eggers, and Henry Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, Santa Margherita Ligure, Italy, June 1995. ACM.

[17] R. v. Hanxleden. Handling Irregular Problems with Fortran D–A Preliminary Report. In *Proceedings of the 4th Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, December 1993.