University of Maryland Inistitute for Advanced Computer Studies Technical Report

UMIACS-TR-2002-64

Exploiting Application-Level Information to Reduce Memory Bandwidth Consumption

Deepak Agarwal and Donald Yeung
Electrical and Computer Engineering Department
Institute for Advanced Computer Studies
University of Maryland at College Park

Abstract

As processors continue to deliver 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 performance bottleneck. Rather than rely solely on wider and faster memories to address memory bandwidth shortages, an alternative is to use existing memory bandwidth more efficiently. A promising approach is *hardware-based selective sub-blocking* [14, 2]. In this technique, hardware predictors track the portions of cache blocks that are referenced by the processor. On a cache miss, the predictors are consulted and only previously referenced portions are fetched into the cache, thus conserving memory bandwidth.

This paper proposes a software-centric approch to selective sub-blocking. We make the key observation that wasteful data fetching inside long cache blocks arises due to certain sparse memory references, and that such memory references can be identified in the application source code. Rather than use hardware predictors to discover sparse memory reference patterns from the dynamic memory reference stream, our approach relies on the programmer or compiler to identify the sparse memory references statically, and to use special annotated memory instructions to specify the amount of spatial reuse associated with such memory references. At runtime, the size annotations select the amount of data to fetch on each cache miss, thus fetching only data that will likely be accessed by the processor. Our results show annotated memory instructions remove between 54% and 71% of cache traffic for 7 applications, reducing more traffic than hardware selective sub-blocking using a 32 Kbyte predictor on all applications, and reducing as much traffic as hardware selective sub-blocking using an 8 Mbyte predictor on 5 out of 7 applications. Overall, annotated memory instructions achieve a 17% performance gain when used alone, and a 22.3% performance gain when combined with software prefetching, compared to a 7.2% performance degradation when prefetching without annotated memory instructions.

1 Introduction

Several researchers have observed in the past that insufficient memory bandwidth limits performance in many important applications. For example, Burger *et al* [3] report between 11% and 31% of the total memory stalls observed in several SPEC benchmarks are due to insufficient memory bandwidth. In addition, Ding and Kennedy [8] observe several scientific kernels require between 3.4 and 10.5 times the L2-memory bus bandwidth provided by the SGI Origin 2000.

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, the growing gap between processor and memory speeds will force architects to more aggressively employ memory latency tolerance techniques, such as prefetching, streaming, multi-threading, and speculative loads. These techniques hide memory latency, but they do not reduce memory traffic. Consequently, performance gains achieved through latency tolerance directly increase memory bandwidth consumption. Without sufficient memory bandwidth, memory latency tolerance techniques become ineffective.

These trends will pressure future memory systems to provide increased memory bandwidth in order to realize the potential performance gains of faster processors and aggressive latency tolerance techniques. Rather than rely solely on wider and faster memory systems, an alternative is to use existing memory resources more efficiently. Caches can be highly inefficient in how they utilize memory bandwidth because they fetch long cache blocks on each cache miss. While long cache blocks exploit spatial locality, they also wastefully fetch data whenever portions of a cache block are not referenced prior to eviction. To tailor the exploitation of spatial locality to application reference patterns, researchers have proposed *hardware-based selective sub-blocking* [14, 2]. This approach relies on hardware predictors to track the portions of cache blocks that are referenced by the processor. On a cache miss, the predictors are consulted and only previously referenced (and possibly discontiguous) portions are fetched into the cache, thus conserving memory bandwidth.

In this paper, we propose a software-centric approach to selective sub-blocking. We make the key observation that wasteful data fetching inside long cache blocks arises due to certain sparse memory reference patterns, and that such reference patterns can be detected statically by examining application source code. Specifically, we identify three common memory reference patterns that lead to sparse memory references: large-stride affine array references, indexed array references, and pointer-chasing references. Our technique relies on the programmer or compiler to identify these reference patterns, and to extract spatial reuse information associated with such references. 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 on each cache miss. Finally, we use a sectored cache to fetch and cache variable-sized fine-grained data accessed through the annotated memory instructions, similar to hardware selective sub-blocking techniques.

Compared to previous selective sub-blocking techniques, our approach uses less hardware, leading to lower system cost and lower power consumption. Using hardware to drive selective sub-blocking can be expensive because predictors must track the selection information for *every unique cache-missing memory block*. In contrast, our approach off-loads the discovery of selection information onto software, thus eliminating the predictor tables. The savings can be significant–we show software annotations reduce more memory traffic for a 64 Kbyte cache than a hardware predictor with a 32 Kbyte table, and achieves similar memory traffic on 5 out of 7 applications to a hardware predictor with an 8 Mbyte table. The advantage of the hardware approach, however, is that it is fully automatic. Furthermore, the hardware approach uses exact runtime information that is not available statically, and can thus identify the referenced portions of cache blocks more precisely than the software approach (but only if sufficiently large predictors are employed).

This paper makes the following contributions. First, we present an off-line algorithm for inserting annotated memory instructions to convey spatial reuse information to the hardware. Second, we propose the hardware necessary to support our annotated memory instructions. Finally, we conduct an experimental evaluation of our technique. Our results show annotated memory instructions remove between 54% and 71% of the memory traffic for 7 applications, comparing favorably to hardware selective sub-blocking as discussed above. These traffic reductions lead to a 17% overall performance gain. When coupled with software prefetching, annotated memory instructions enable a 22.3% performance gain, compared to a 7.2% performance degradation when prefetching without annotated memory instructions.

The rest of this paper is organized as follows. Section 2 discusses related work. Then, Sections 3 and 4 present our technique. Section 5 characterizes the bandwidth reductions our technique achieves, and Section 6 evaluates its performance gains. Finally, Section 7 concludes the paper.

2 Related Work

Our technique is based on hardware selective sub-blocking [14, 2]. In this paper, we compare our technique against Spatial Footprint Predictors (SFP) [14], but an almost identical approach was also proposed independently by Burger, called Sub-Block Prefetching (SBP) [2]. Another similar approach is the Spatial Locality Detection Table (SLDT) [12]. SLDT is less flexible than SFP and SBP, dynamically choosing between two power-of-two fetch sizes rather than fetching arbitrary footprints of (possibly discontiguous) sub-blocks. Similar to these three techniques, our approach adapts the fetch size using spatial reuse information observed in the application. However, we extract spatial reuse information statically from application source code, whereas SFP, SBP, and

SLDT discover the information dynamically by monitoring memory access patterns in hardware.

Prior to SFP, SBP, and SLDT, several researchers have considered adapting the cache-line and/or fetch size for regular memory access patterns. Virtual Cache Lines (VCL) [20] uses a fixed cache block size for normal references, and fetches multiple sequential cache blocks when the compiler detects high spatial reuse. The Dynamically Variable Line-Size (D-VLS) cache [11] and stride prefetching cache [9] propose similar dynamic fetch sizing techniques, but use hardware to detect the degree of spatial reuse. Finally, the dual data cache [10] selects between two caches, also in hardware, tuned for either spatial locality or temporal locality. These techniques employ line-size selection algorithms that are designed for affine array references and are thus targeted to numeric codes. In comparison, our algorithms handle both irregular and regular access patterns and thus work for non-numeric codes as well.

Impulse [4] performs fine-grained address translation in the memory controller to alter data structure layout under software control, permitting software to remap sparse data so that it is stored densely in cache. Our approach only provides 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 reduces memory bandwidth consumption only. Impulse improves both memory bandwidth consumption and cache utilization. Another approach, similar in spirit to Impulse, is to use data compression hardware across memory interconnect to compact individual words as they are being transmitted [7]. Our approach could be combined with this approach to further reduce memory traffic.

Finally, all-software techniques for addressing memory bandwidth bottlenecks have also been studied. Ding and Kennedy [8] propose compiler optimizations specifically for reducing memory traffic. Chilimbi *et al* propose cache-conscious data layout [6] and field reordering [5] optimizations that increase data locality for irregular access patterns, and hence also reduce memory traffic. The advantage of these techniques is they require no special hardware support. However, they are applicable only when the correctness of the code transformations can be guaranteed by the compiler or programmer. Since our approach only provides hints to the memory system, it can reduce memory traffic even when such code transformations are not legal or safe.

3 Controlling Memory Bandwidth Consumption in Software

In this section, we present our software approach for controlling memory bandwidth consumption. First, Section 3.1 presents an overview. Then, Sections 3.2 and 3.3 describe how to extract spatial locality information for driving selective data fetching. Finally, Section 3.4 discusses ISA support, and Section 3.5 discusses the implementation issues associated with our extraction algorithms.

3.1 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 as normal.

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 reuse associated with the memory reference.

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.

3.2 Identifying Sparse Memory References

Our code analysis examines loops to identify frequently executed memory references that exhibit poor spatial reuse. As discussed in Section 3.1, we look for three types of loops–large-stride affine array, indexed array, and pointer-chasing loops. 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 have a high potential to 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

```
   // 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.

in two 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 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.

3.3 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

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

    p = list->patient;
    while (p != patient) {
        list = list->forward;
        p = list->patient;
    }
        .
        .
        .
}
```
load #2

load #1

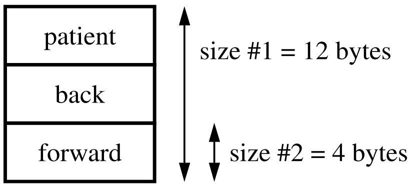
b).
```
struct List {
    struct Patient *patient;
    struct List *back;
    struct List *forward;
}
```


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.

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 reuse 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 3.2. If only one sparse memory reference occurs to each array element or link node, we assume there is no spatial reuse 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 reuse 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 [19], 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

	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.

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 "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.

3.4 Annotated Memory Instructions

We augment the instruction set with several new memory instructions to encode the transfer size information described in Section 3.3. 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 3.2. 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.

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 3.3 must be rounded up to the next available power-of-two size annotation.

3.5 Discussion

In this paper, we perform the code analyses described in Sections 3.2 and 3.3 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 [18], indexed array loops [18], and pointer-chasing loops [15] 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 3.3. 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.

4 Hardware Support for Bandwidth Reduction

An annotated memory instruction, as described in Section 3.4, fetches a variable-sized narrow-width block of data on each cache miss. Furthermore, 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.

Previous hardware techniques for adaptively exploiting spatial locality have addressed this variable fetch size problem [14, 2, 12]; we adopt a similar approach. 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.

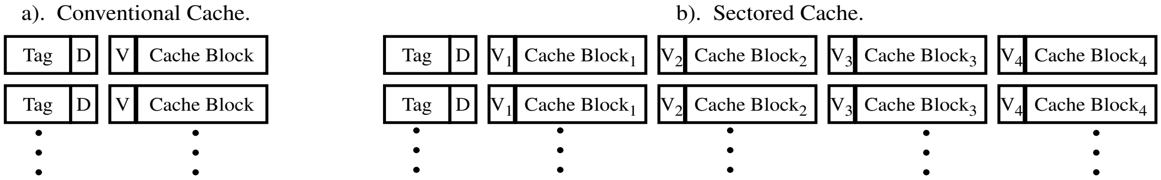


Figure 3: a). A conventional cache provides one tag for each cache block. b). A sectored cache provides one tag for all the cache blocks in a sector, hence reducing the tag overhead. "V" denotes a valid bit, and "D" denotes a dirty bit.

4.1 Sectored Caches

To exploit the potential bandwidth savings afforded by sparse memory references, a small cache block is required. One drawback of small cache blocks is high tag overhead. Fortunately, the tag overhead of small cache blocks can be mitigated using a *sectored cache*, as is done in [14] and [2]. Compared to a conventional cache, a sectored cache provides a cache tag for every *sector*, which consists of multiple cache blocks that are contiguous in the address space, as shown in Figure 3.[1] Each cache block has its own valid bit, so cache blocks from the same sector can be fetched independently. Because each tag is shared between multiple blocks, 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.

4.2 Variable Fetch Size

In addition to supporting small cache blocks, 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.

[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 [14]. 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.

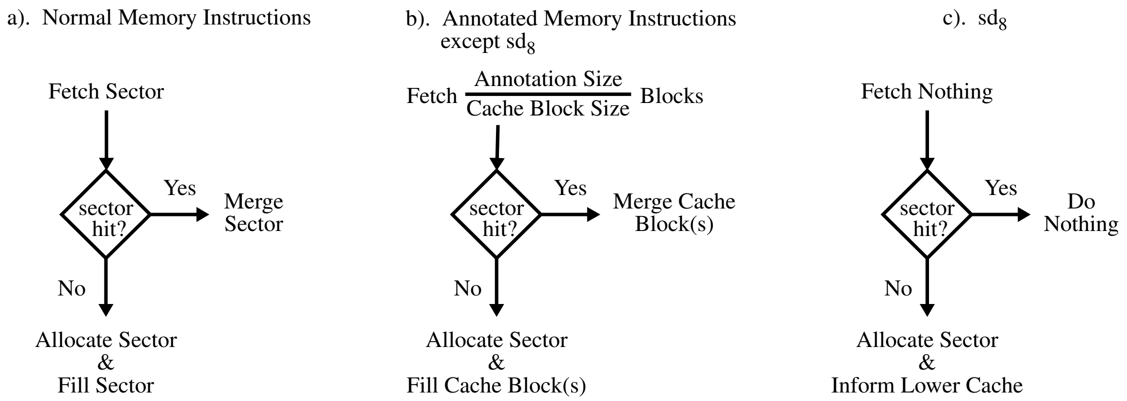


Figure 4: Action taken on a cache miss for a). normal memory instructions, b). annotated memory instructions except sd_8, and c). sd_8.

When a normal memory instruction suffers a cache miss (Figure 4a), the cache requests an 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, and we perform a sector fill or merge depending on whether the cache miss was a sector miss or cache block miss, as described above.

Finally, there is one exceptional 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.

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.

In this paper, we evaluate annotated memory instructions for uniprocessors. Our technique can also be applied to multiprocessors by integrating sectored caches with cache coherence protocols. One approach is to maintain coherence at the cache block granularity, and to break sector-level transactions into multiple cache block transactions. However, this approach can be expensive given the small size of cache blocks. Another approach is to maintain coherence at the sector granularity, and to handle sectors with missing cache blocks. Coherence operations that go through main memory do not require significant modification because all cache blocks are available in memory at all times. Protocols that allow cache-to-cache transfers, however, would require modification to detect when a cache block is missing from a requested sector, and to perform additional transactions with memory to acquire the missing cache block(s).

5 Cache Behavior Characterization

Having described our technique in Sections 3 and 4, 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.

5.1 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 [22]. The next two benchmarks are from Olden [19]. 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

Program	Input	Sparse Refs	Init	Warm	Data
IRREG	144K nodes, 11 sweeps	Indexed array	993.1	5.2 (3.9)	52.8 (13.2)
MOLDYN	131K mols, 11 sweeps	Indexed array	761.5	6.7 (3.3)	66.7 (10.1)
NBF	144K mols, 11 sweeps	Indexed array	49.6	4.2 (2.1)	41.5 (11.3)
HEALTH	5 levels, 106 itrs	Ptr-chasing	160.7	0.5 (0.27)	56.9 (31.0)
MST	1024 nodes, 1024 itrs	Ptr-chasing	184.2	9.8 (3.7)	19.5 (7.3)
BZIP2	"ref" input	Indexed array	147.2	77.3 (22.2)	60.4 (17.0)
MCF	"ref" input (41 itrs)	Affine array, Ptr-chasing	6909.5	1.3 (0.54)	50.7 (21.8)

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).

Cache Model	Sector Size	Block Size	Associativity	Replacement Policy	Write-Hit Policy
Conventional	-	64 bytes	2-way	LRU	Write-back
Annotated	64 bytes	8 bytes	2-way	LRU	Write-back
MTC	-	4 bytes	full	MIN	Write-back
SFP	64 bytes	8 bytes	2-way	LRU	Write-back

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

limits on achievable traffic reduction. Finally, we compare our technique against Spatial Footprint Predictors (SFP), an existing hardware-based selective sub-blocking technique.

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 3.2 and 3.3 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) [3]. 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 5.3. Finally, we implemented an SFP cache which we will describe in Section 5.4. Table 3 presents the cache parameters used for 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, 64 bytes, to facilitate a meaningful comparison.

Each of our cache simulations contain three phases: we perform functional simulation during an *initialization phase*, we turn on modeling and perform a cache *warm up phase*, and then we enter the actual *data collection phase*. Each phase is chosen in the following manner. We identify each benchmark's initialization code and simulate it entirely in the first phase. After initialization, Irreg, Moldyn, and NBF perform a computation repeatedly over a static data structure. For these benchmarks, the warm-up and data collection phases simulate the first and next 10 iterations, respectively. Health and MCF also perform iterative computations, but the data structure is dynamic. For Health and MCF, we include several compute iterations in the first phase (500 and 5000, respectively) to "build up" the data structure. Then, we warm up the cache for 1 iteration and collect data over several iterations (105 and 40, respectively). For MST, another iterative computation, we simulate the entire program after initialization, performing the first 100 out of 1024 compute iterations in the warm up phase. Finally, for BZIP2, we warm up the cache and collect data over large regions to capture representative behavior because we could not find a single "compute loop" in this benchmark. Table 2 reports the phase sizes in the "Init," "Warm," and "Data" columns. Although the warm up and data collection phases are small, they accurately reflect cache behavior due to the iterative nature of our benchmarks. We verified for several cases that increasing warm up or data collection time does not qualitatively change our results.

5.2 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 values from 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 over half of the cache traffic, between 54% and 71%. 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 55% or more for caches 64K or smaller, but lose their effectiveness for larger caches. 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.

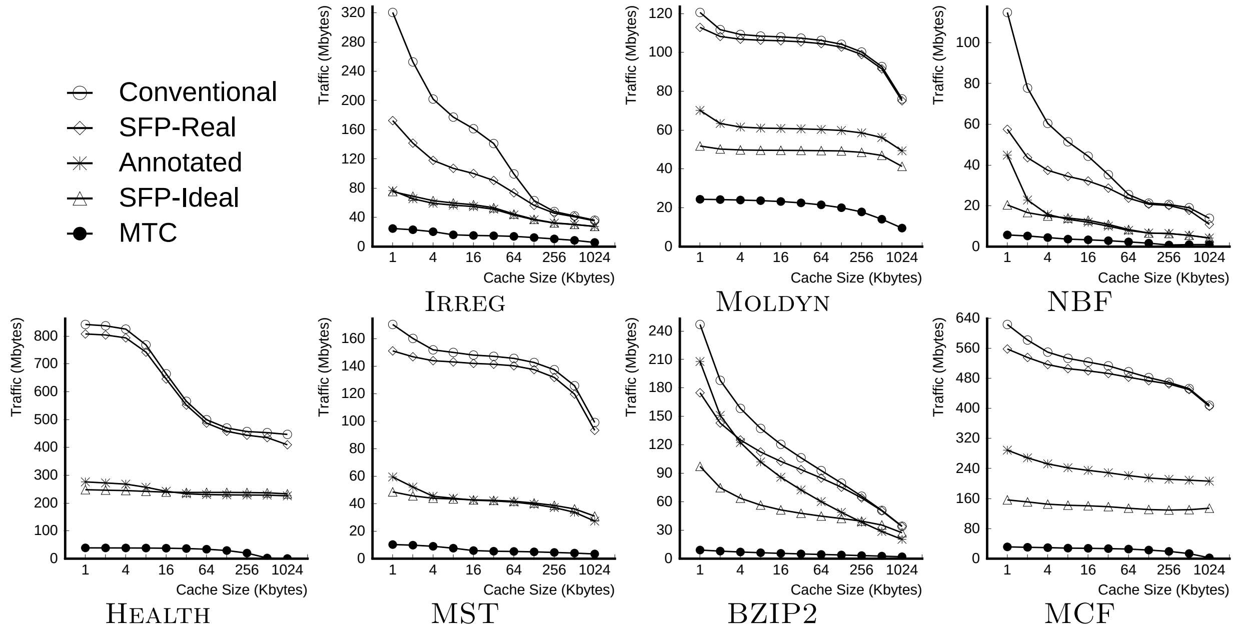


Figure 5: Cache traffic as a function of cache size. Traffic is reported for a conventional cache (Conventional), a sectored cache with annotated memory instructions (Annotated), an MTC (MTC), and two SFP caches (SFP-Ideal and SFP-Real). All traffic values are in units of MBytes.

As the exploitation of spatial locality increases in Irreg, annotated memory instructions lose their advantage. Finally, for Moldyn and BZIP2, the traffic reductions are 42% and 31%, respectively, averaged over all cache sizes. The memory reference patterns in Moldyn are less sparse, providing fewer opportunities to reduce traffic. The behavior of BZIP2 will be explained later in Section 5.3.

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 10.7% and 43.1% for Moldyn, NBF, MST, BZIP2, and MCF, and roughly 85% 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 3.3. 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 6, the benefit of traffic reduction usually outweighs the increase in miss rate, resulting in performance gains.

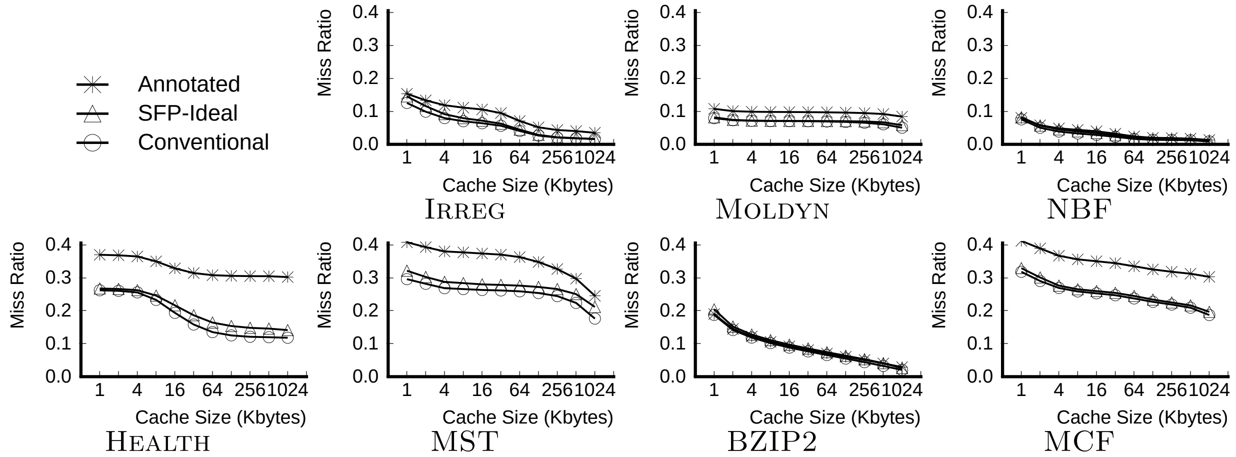


Figure 6: Cache miss rate as a function of cache size. Miss rates are reported for a conventional cache (Conventional), a sectored cache with annotated memory instructions (Annotated), and an SFP cache (SFP-Ideal).

5.3 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 4 times more traffic than MTC averaged over all cache sizes. For HEALTH, MST, and MCF, the average increase is between 5.4 and 9.7 times, and for BZIP2 it is 15 times. 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

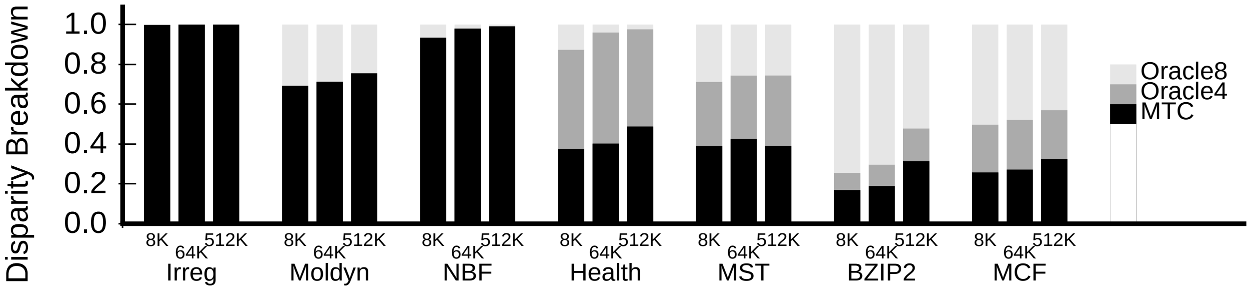


Figure 7: Breakdown of the traffic disparity between our technique and MTC in Figure 5 for 8K, 64K, and 512K cache sizes. The disparity is broken down into 3 components labeled "Oracle8," "Oracle4," and "MTC."

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 30-40%, and for BZIP2, 66%, 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 3.4), and alignment of cache misses to an annotation-sized boundary (Section 4.2) result in unnecessary fetches. Second, our technique is effective only for the access patterns described in Section 3.2. 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. Further improvements are possible, perhaps by using more sophisticated annotations, or by extending our analysis to increase coverage. Second, 4-byte cache blocks can provide improvements; however, this may be impractical on some systems since cache blocks cannot be smaller than the memory bus width. Finally, there are large gains in the MTC components, but our technique cannot exploit them since they are a consequence of better cache organization and management policies, rather than better fetch policies.

5.4 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 [14].

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 our workloads. 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 20.6% and 25.3% more traffic, respectively, than our technique. Finally, in BZIP2, SFP-Ideal outperforms annotated memory instructions for small caches, but is slightly worse for large caches. Figure 5 demonstrates our technique achieves comparable traffic with an aggressive SFP, despite using much less hardware. Comparing miss rates, however, Figure 6 shows SFP-Ideal outperforms our technique, essentially matching the miss rate of a conventional cache. As discussed in Section 5.2, our technique misses opportunities to exploit spatial locality due to spatial reuse that is undetectable statically. SFP can exploit such reuse because it observes application access patterns dynamically.

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 SFP-Real is unable to reduce any traffic for MOLDYN, HEALTH, MST, and MCF. In IRREG, NBF, and BZIP2, modest traffic reductions are achieved, but only at small cache sizes. In practically all cases, our technique reduces more traffic than SFP-Real. The large working sets in our benchmarks 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.

Finally, thus far we have only presented results using the cache parameters from Table 3. We also ran simulations that vary sector size (32-128 bytes) as well as cache-block size (4-16 bytes). A representative sample of these simulations appear in Appendix A and will be discussed if the paper is accepted. The additional simulations show the same qualitative results as those presented

Processor Model 1 cycle = 0.5ns	8-way issue Superscalar processor. Gshare predictor with 2K entries. Instruction Fetch queue = 32. Instruction Window = 64. Load-Store Queue = 32 . Integer /Floating Point units = 4/4. Integer latency = 1 cycle. Floating Add/Mult/Div latency = 2/4/12 cycles.
Cache Model 1 cycle = 0.5ns	L1/L2 cache size = 16 K-split/512K-unified. L1/L2 associativity = 2-way. L1/L2 hit time = 1 cycle. L1/L2 Sector size = 32/64 bytes. L1/L2 block size = 8/8 bytes. L1/L2 MSHRs = 32/32.
Memory Sub-System Model	DRAM banks = 64. Memory System Bus width = 8 bytes. Address send = 4ns Row Access Strobe = 12 ns. Column Access Strobe = 12 ns. Data Transfer(per 8 bytes) = 4ns.

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 0.5-ns processor cycle time.

in Figures 5 and 6.

6 Performance Evaluation

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

6.1 Simulation Environment

Like the cache simulators from Section 5, 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 2 GHz dynamically scheduled 8-way issue superscalar. We also simulate a two-level cache hierarchy. Our cycle-accurate simulator implements the "Conventional" and "Annotated" cache models from Section 5 only (unfortunately, we did not have time to implement the SFP cache model). 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 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 variable-sized transfers. 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 (the timing parameters closely model Micron's DDR333 [17]). These parameters correspond to a 60 ns (120 processor cycles) L2 sector fill latency and a 2 GB/s memory system bus bandwidth.

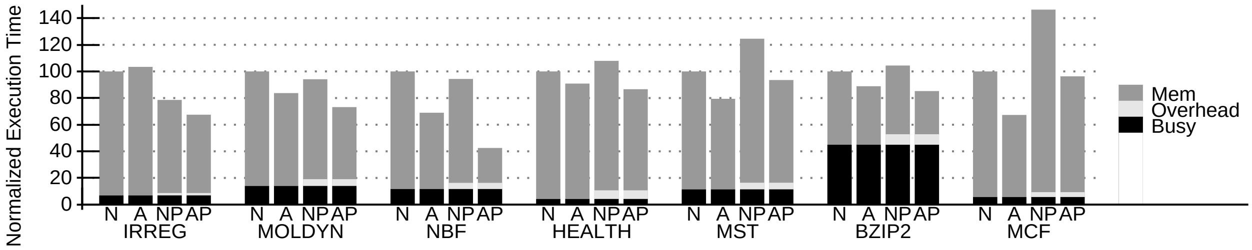


Figure 8: Execution time breakdown for annotated memory instructions. Individual bars show performance without prefetching using normal and annotated memory instructions, labeled "N" and "A", and performance with prefetching using normal and annotated memory instructions, labeled "NP" and "AP."

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 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 [18]. For pointer-chasing references, we use the prefetch arrays technique [13]. Instrumentation of annotated memory instructions for prefetch, load, and store instructions occurs after software prefetching has been applied.

6.2 Performance of Annotated Memory Instructions

Figure 8 shows the performance of our annotated memory instructions on the baseline memory system described in Section 6.1. 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 32.7% (MCF), and by 17% on average. The cache traffic reductions of our technique reported in Section 5.2 result in two performance benefits. First, reduced cache traffic lowers contention in the memory system, reducing 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 lower cache miss latency (*e.g.* filling an L2 cache block takes 64 cycles, compared to 120 cycles for filling an L2 sector), thus increasing the throughput of pointer-chasing loops.

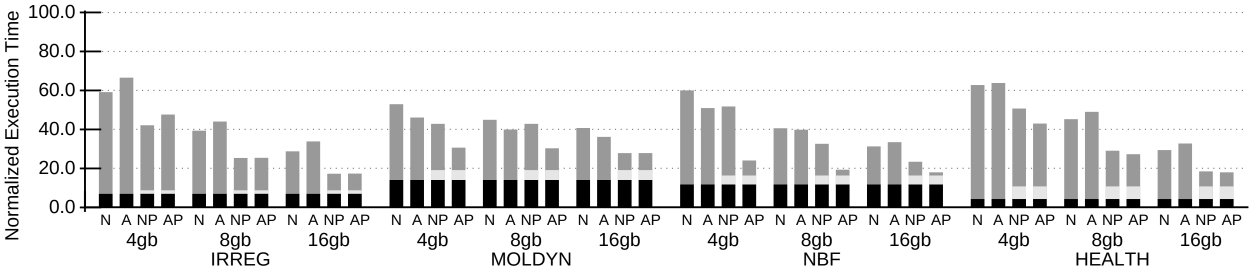
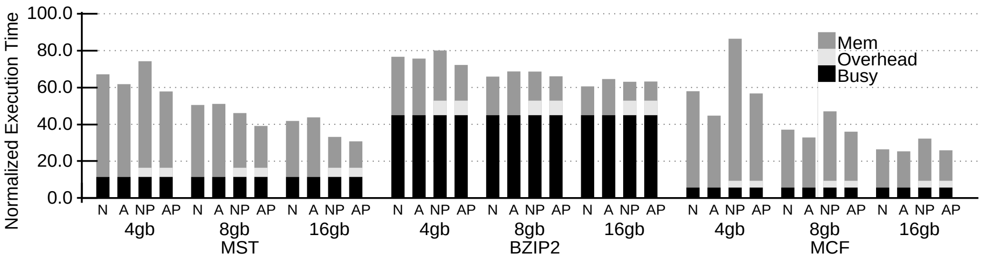
Recall from Section 5 that annotated memory instructions increase the cache miss rate due to reduced exploitation of spatial locality. Figure 8 demonstrates that the benefit of reduced cache traffic outweighs the increase in cache miss rate, resulting in a net performance gain for most applications. IRREG is the one exception. As discussed in Section 5.2, 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 3.4% performance loss for IRREG.

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 7.2% degradation averaged across all benchmarks. Prefetching adds software overhead, and can increase memory traffic due to speculative prefetches. The 2 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, 22.3% on average. 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, hiding their latency. Thus, annotated memory instructions are generally more effective when coupled with software prefetching.

6.3 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.

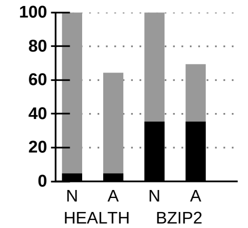


Figure 10: SMT Results

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. For every 2x reduction in the Data Transfer parameter, we also reduce the "Row" and "Column Access Strobe" parameters by 30%.

In Figure 9, we report performance at 4, 8, and 16 GB/s memory system bus bandwidths. All bars have been normalized against the "N" versions at 2 GB/s from Figure 8. Without prefetching (*i.e.* comparing the "N" and "A" bars), Figure 9 shows that the performance boost due to annotated memory instructions decreases with increasing bandwidth and by 16GB/s all applications except for MOLDYN and MCF perform worse. As 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 5.2 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 Irreg at 4 GB/s where we suffer a degradation of 13%. 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 bandwidth, 2

GB/s, rather than 16 GB/s. More importantly, we believe behavior on future systems will more closely resemble our baseline results in Figure 8 given the trends in processor and memory speeds.

6.4 Other Latency Tolerance Techniques

This section briefly studies our technique applied to Simultaneous Multithreading (SMT) [21]. To perform the study, we use the simulator from [16], which is derived from SimpleScalar (augmented to model SMT's multiple hardware contexts), so it runs our binaries without recompilation. We integrated the SMT processor model from this simulator with our memory system models described in Section 6.1. We configured the resulting simulator with parameters similar to those specified in Table 4, except that the simulator models 4 hardware contexts. In Figure 10, we report the execution time when running 4 copies of the same benchmark (Health or BZIP2) simultaneously, without and with annotated memory instructions. Figure 10 shows our technique provides a 35.6% and 30.6% performance gain for Health and BZIP2, respectively, on SMT. These results demonstrate that annotated memory instructions can benefit other latency tolerance techniques, such as multithreading, in addition to software prefetching.

7 Conclusion

Our work identifies several data structure traversals that access memory sparsely, including large-stride affine array and indexed array traversals, and pointer-chasing traversals. We extract spatial reuse information associated with these traversals, and convey this information to the memory system. Our technique removes between 54% and 71% of the cache traffic for 7 applications, reducing more traffic than hardware selective sub-blocking using a 32 Kbyte predictor, and reducing a similar amount of traffic as hardware selective sub-blocking using an 8 Mbyte predictor. These traffic reductions come at the expense of increased cache miss rates, ranging between 10.7% and 43.1% for 5 applications, and 85% for 2 applications. Overall, we show annotated memory instructions provide a 17% performance gain. Furthermore, performance gains improve when annotated memory instructions are coupled with software prefetching, enabling a 22.3% performance gain, compared to a 7.2% performance degradation when prefetching without annotated memory instructions.

We conclude that application-level information can be used to gainfully reduce memory bandwidth consumption and increase performance for irregular and non-numeric codes. Based on our results, we believe software should take an active role in managing memory traffic, particularly in the context of memory latency tolerance techniques where performance is highly sensitive to memory traffic volume, but robust to inexact static analysis.

References

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

[2] D. Burger. Hardware Techniques to Improve the Performance of the Processor/Memory Interface. Technical report, Computer Science Department, University of Wisconsin-Madison, December 1998.

[3] D. Burger, J. R. Goodman, and A. Kagi. Memory Bandwidth Limitations of Future Microprocessors. In *Proceedings of the 23rd Annual ISCA*, pages 78–89, Philadelphia, PA, May '96. ACM.

[4] 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 HPCA-5*, pages 70–79, Jan '99.

[5] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-Conscious Structure Defintion. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, GA, May 1999. ACM.

[6] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-Conscious Structure Layout. In *In Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, GA, May 1999. ACM.

[7] D. Citron and L. Rudolph. Creating a Wider Bus Using Caching Techniques. In *Proceedings of the 1st International Symposium on High-Performance Computer Architecture*, Raleigh, NC, January 1995.

[8] C. Ding and K. Kennedy. Memory Bandwidth Bottleneck and its Amelioration by a Compiler. In *Proceedings of the Int. Parallel and Distributed Processing Symposium*, Cancun, Mexico, May 2000.

[9] J. W. C. Fu and J. H. Patel. Data Prefetching in Multiprocessor Vector Cache Memories. In *Proceedings of the 18th Annual Symp. on Computer Architecture*, pages 54–63, Toronto, Canada, May '91. ACM.

[10] A. Gonzalez, C. Aliagas, and M. Valero. A Data Cache with Multiple Cacheing Strategies Tuned to Different Types of Locality. In *Proceedings of the ACM 1995 International Conference on Supercomputing*, pages 338–347, Barcelona, Spain, July 1995.

[11] K. Inoue, K. Kai, and K. Murakami. Dynamically Variable Line-Size Cache Exploiting High On-Chip Memory Bandwidth of Merged DRAM/Logic LSIs. *IEICE Transactions on Electronics*, E81-C(9):1438–1447, September 1998.

[12] T. L. Johnson, M. C. Merten, and W. W. Hwu. Run-time Spatial Locality Detection and Optimization. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 57–64, Research Triangle Park, NC, December 1997.

[13] M. Karlsson, F. Dahlgren, and P. Stenstrom. A Prefetching Technique for Irregular Accesses to Linked Data Structures. In *Proceedings of HPCA-6*, Toulouse, France, Jan 2000.

[14] S. Kumar and C. Wilkerson. Exploiting Spatial Locality in Data Caches using Spatial Footprints. In *Proceedings of the 25th Annual ISCA*, pages 357–368, Barcelona, Spain, June '98. ACM.

[15] C. Luk and T. 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.

[16] D. Madon, E. Sanchez, and S. Monnier. A Study of a Simultaneous Multithreaded Processor Implementation. In *Proceedings of EuroPar '99*, pages 716–726, Toulouse, France, August 1999. Springer-Verlag.

[17] MICRON. 256 Mb DDR333 SDRAM Part No. MT46V64M4. In *www.micron.com/dramds*, 2000.

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

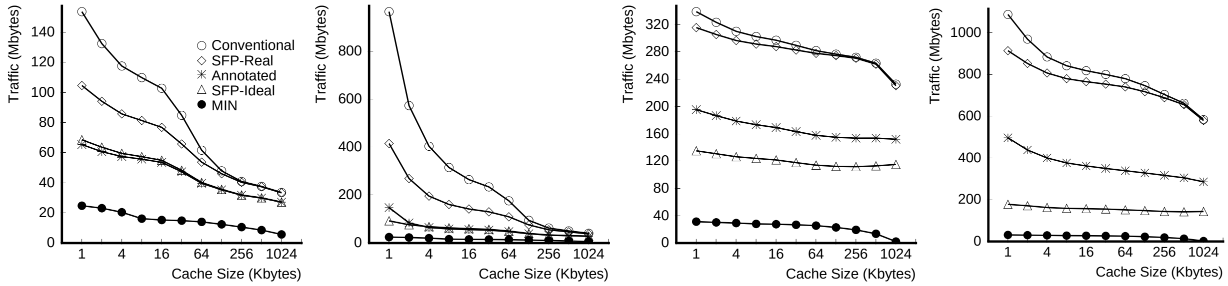
[19] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren. Supporting Dynamic Data Structures on Distributed Memory Machines. *ACM Transactions on Programming Languages and Systems*, 17(2), March 1995.

[20] 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.

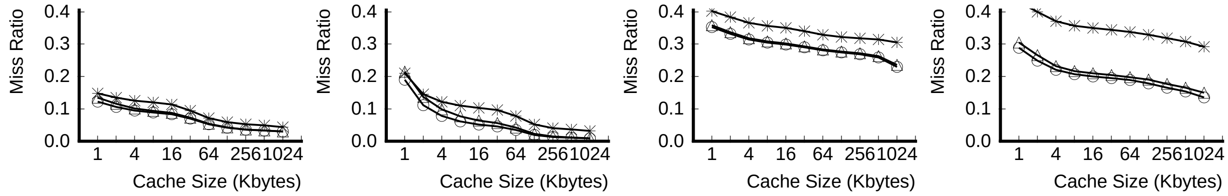
[21] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22nd Annual ISCA*, pages 392–403, Santa Margherita Ligure, Italy, June '95. ACM.

[22] 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.

A Additional Cache Simulations

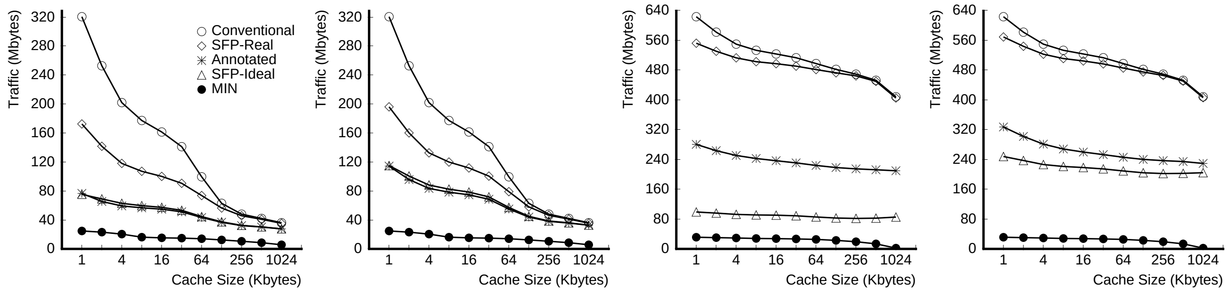


Irreg (Sector = 32) Irreg (Sector = 128) MCF (Sector = 32) MCF (Sector = 128)

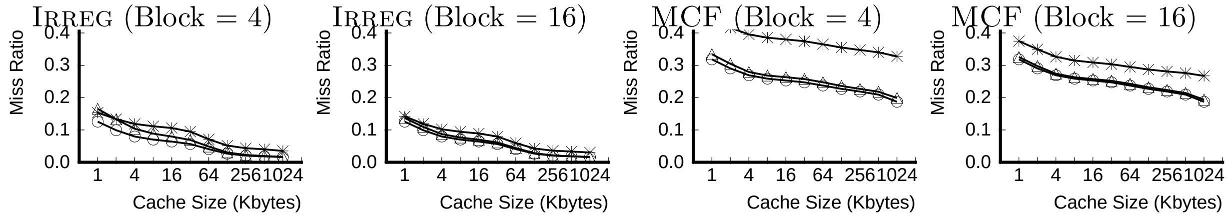


Irreg (Sector = 32) Irreg (Sector = 128) MCF (Sector = 32) MCF (Sector = 128)

Figure 10: Sensitivity to sector size variations. Cache traffic and miss rate results, similar to those in Figures 5 and 6, for Irreg and MCF with sector sizes of 32 bytes and 128 bytes.



Irreg (Block = 4) Irreg (Block = 16) MCF (Block = 4) MCF (Block = 16)



Irreg (Block = 4) Irreg (Block = 16) MCF (Block = 4) MCF (Block = 16)

Figure 11: Sensitivity to cache-block size variations. Cache traffic and miss rate results, similar to those in Figures 5 and 6, for Irreg and MCF with cache-block sizes of 4 and 16 bytes.