Submitted to the 10th International Symposium on High Performance Computer Architecture

Reducing Memory Bandwidth Consumption Via Compiler-Driven Selective Sub-Blocking

Deepak Agarwal, Wanli Liu, 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, 1]. 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 compiler-driven approach 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 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 compiler-driven selective sub-blocking removes between 31% 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 a comparable amount of traffic as hardware selective sub-blocking using an 8 Mbyte predictor for 6 out of 7 applications. Overall, our technique achieves a 15% performance gain when used alone, and a 29% performance gain when combined with software prefetching, compared to an 8% performance degradation when prefetching without compiler-driven selective sub-blocking.

1 Introduction

Several researchers have observed that insufficient memory bandwidth limits performance 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. In addition, Ding and Kennedy [7] 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, 1]. 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 compiler-driven approach to selective sub-blocking. We make the key observation that wasteful data fetching inside long cache blocks arises due to sparse memory reference patterns, and that such reference patterns can often 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 compiler to identify these reference patterns, and to extract spatial reuse information associated with such references. Through special size-annotated memory instructions, the compiler 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 the compiler, thus eliminating the predictor tables. The savings can be significant–we show a compiler reduces more memory traffic for a 64 Kbyte cache than a hardware predictor with a 32 Kbyte table, and achieves comparable memory traffic on 6 out of 7 applications to a hardware predictor with an 8 Mbyte table. The advantage of the hardware approach, however, is that it is transparent to software. 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 a compiler (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 into C programs to convey spatial reuse information to the hardware, and implement the algorithm in a prototype compiler based on the SUIF framework [10]. Second, we describe the hardware necessary to support our annotated memory instructions. Finally, we conduct an experimental evaluation of our technique. Our results show compiler-driven selective sub-blocking removes between 31% and 71% of the memory traffic for 7 applications. These traffic reductions lead to a 15% overall performance gain. When coupled with software prefetching, our technique provides a 29% performance gain, compared to an 8% performance degradation when prefetching without compiler-driven selective sub-blocking.

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 work is motivated by hardware selective sub-blocking [14, 1]. 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) [1]. 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 using a compiler, 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) [18] uses a fixed cache block size for normal references, and fetches multiple sequential cache blocks when high spatial reuse is detected. Like our technique, VCL uses a compiler to detect spatial reuse. The Dynamically Variable Line-Size (D-VLS) cache [11] and stride prefetching cache [8] propose similar dynamic fetch sizing techniques, but use hardware to detect spatial reuse. Finally, the dual data cache [9] 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 compiler handles both irregular and regular access patterns and thus works for non-numeric codes as well.

Impulse [3] 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 [6]. 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 [7] propose compiler optimizations specifically for reducing memory traffic. Chilimbi *et al* propose cache-conscious data layout [5] and field reordering [4] 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 Compiler Support for Conserving Memory Bandwidth

We present a compiler for C programs that extracts information about the programs' memory access patterns, and conveys it to the memory system. The memory system uses this information

```
// a).  Affine Array       // b).  Indexed Array      // c).  Pointer-Chasing
for (i=0; i<N; i+=8) {     for (i=0; i<N; i++) {      for (ptr=root; ptr; ptr=ptr->next) {
  ... = A[i];                ... = A[C[i]];             ... = ptr->data;
}                          }                          }
```

Figure 1: Loop traversals that exhibit sparse memory reference patterns: a). a large-stride affine array traversal, b). an indexed array traversal, and c). a pointer-chasing traveral.

to customize the transfer size for each cache miss to match the degree of spatial locality associated with the missing reference. Our approach targets memory references that are likely to exhibit *sparse memory access patterns*. For these memory references, the memory system uses a compiler-specified cache miss transfer size that is smaller than a cache block to avoid fetching useless data, hence conserving memory bandwidth. For all other memory references, the memory system uses the default cache block transfer size to exploit spatial locality as normal.

In this section, we describe the details of our compiler support. Sections 3.1 and 3.2 describe the compiler algorithms. Then, Section 3.3 discusses several implementation issues.

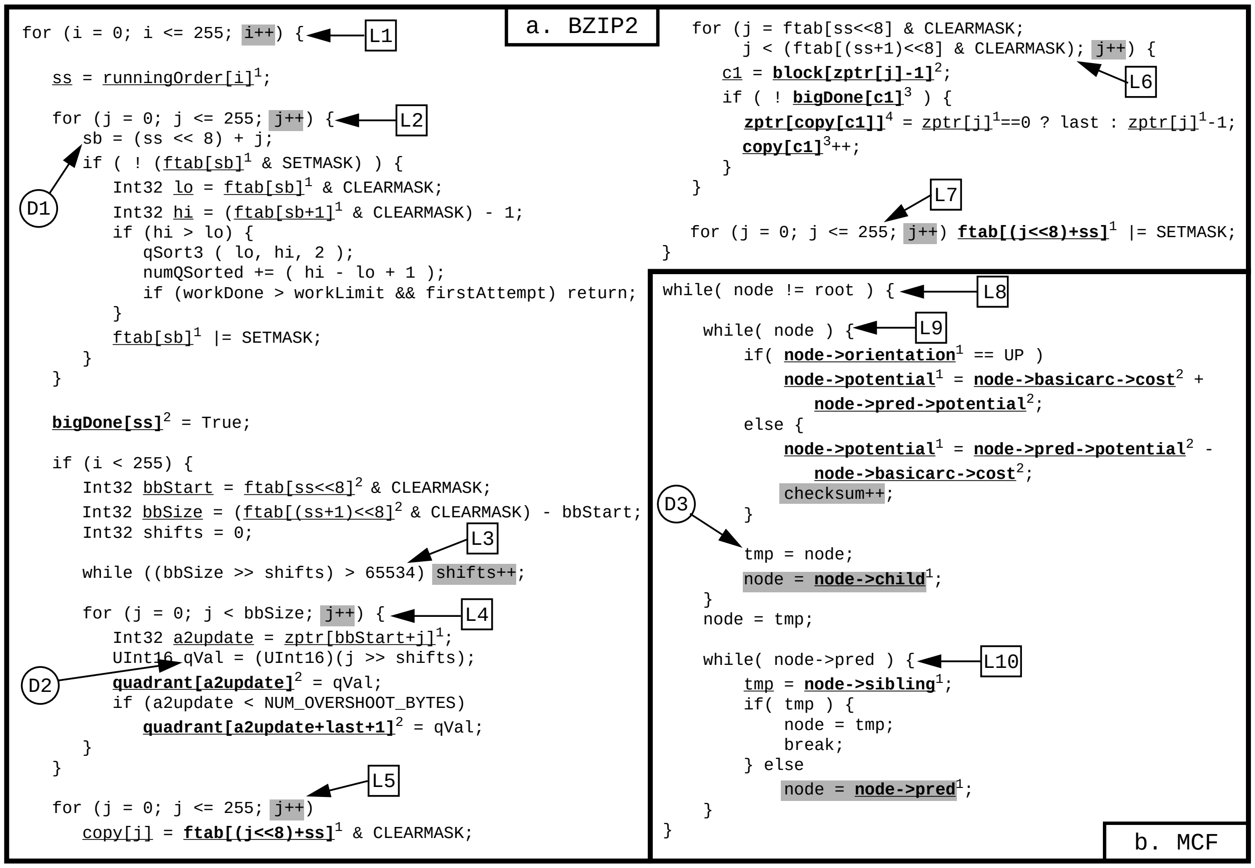
3.1 Identifying Sparse Memory References

3.1.1 Loop-Based Approach

Our compiler examines loops to identify frequently executed memory references that exhibit poor spatial reuse. From our experience, three types of loop traversals found in many applications commonly lead to sparse memory reference patterns: large-stride affine array, indexed array, and pointer-chasing loop traversals. C code examples illustrating these traversals appear in Figure 1.

All three loop traversals in Figure 1 share a common characteristic: 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. Large strides can occur if the loop induction variable used as an array index is incremented by a large value each loop iteration. Figure 1a illustrates this case.

Indexed array and pointer-chasing traversals exhibit low spatial locality due to *indirect memory references*. In indexed array traversals, a data array is accessed using an index provided by another array, called the "index array." Figure 1b illustrates such an indexed array traversal. 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 illustrated in Figure 1c. Logically contiguous link nodes are usually not physically



```
for (i = 0; i <= 255; i++) {

   ss = runningOrder[i]^1;

   for (j = 0; j <= 255; j++) {
      sb = (ss << 8) + j;
      if ( ! (ftab[sb]^1 & SETMASK) ) {
         Int32 lo = ftab[sb]^1 & CLEARMASK;
         Int32 hi = (ftab[sb+1]^1 & CLEARMASK) - 1;
         if (hi > lo) {
            qSort3 ( lo, hi, 2 );
            numQSorted += ( hi - lo + 1 );
            if (workDone > workLimit && firstAttempt) return;
         }
         ftab[sb]^1 |= SETMASK;
      }
   }

   bigDone[ss]^2 = True;

   if (i < 255) {
      Int32 bbStart = ftab[ss<<8]^2 & CLEARMASK;
      Int32 bbSize = (ftab[(ss+1)<<8]^2 & CLEARMASK) - bbStart;
      Int32 shifts = 0;

      while ((bbSize >> shifts) > 65534) shifts++;

      for (j = 0; j < bbSize; j++) {
         Int32 a2update = zptr[bbStart+j]^1;
         UInt16 qVal = (UInt16)(j >> shifts);
         quadrant[a2update]^2 = qVal;
         if (a2update < NUM_OVERSHOOT_BYTES)
            quadrant[a2update+last+1]^2 = qVal;
      }
   }

   for (j = 0; j <= 255; j++)
      copy[j] = ftab[(j<<8)+ss]^1 & CLEARMASK;
   for (j = ftab[ss<<8] & CLEARMASK;
        j < (ftab[(ss+1)<<8] & CLEARMASK); j++) {
      c1 = block[zptr[j]-1]^2;
      if ( ! bigDone[c1]^3 ) {
         zptr[copy[c1]]^4 = zptr[j]^1==0 ? last : zptr[j]^1-1;
         copy[c1]^3++;
      }
   }

   for (j = 0; j <= 255; j++) ftab[(j<<8)+ss]^1 |= SETMASK;
}
```

```
while( node != root ) {

   while( node ) {
      if( node->orientation^1 == UP )
         node->potential^1 = node->basicarc->cost^2 +
            node->pred->potential^2;
      else {
         node->potential^1 = node->pred->potential^2 -
            node->basicarc->cost^2;
         checksum++;
      }
      tmp = node;
      node = node->child^1;
   }
   node = tmp;

   while( node->pred ) {
      tmp = node->sibling^1;
      if( tmp ) {
         node = tmp;
         break;
      } else
         node = node->pred^1;
   }
}
```

Figure 2: Analysis for a. Bzip2 and b. MCF. Boxed labels indicate loops. Shaded boxes highlight LIVs. Circled labels indicate derived LIVs. Underlined identifiers indicate LMRs and derived LMRs. Superscripted values indicate levels of indirection for LMRs. Boldfaced identifiers indicate sparse memory references.

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.

3.1.2 Analysis

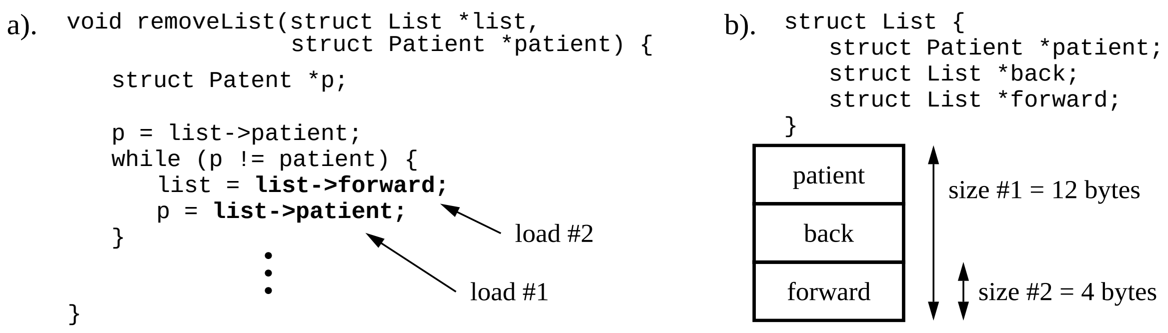
We now present compiler algorithms to identify sparse memory references in affine array, indexed array, and pointer-chasing loop traversals. For illustrative purposes, Figure 2 applies our analyses on several loops from Bzip2 and MCF, two benchmarks from the SPECint2000 suite. The boxes in Figure 2, labeled “L1” to “L10,” indicate 10 loops that have been analyzed.

The first step is to identify loop induction variables (LIVs). Our compiler identifies two kinds of LIVs based on the update method. Scalar integers that are updated arithmetically inside a loop are *affine LIVs*, while pointers that are updated via indirection through the same pointer variable are *pointer-chasing LIVs*. (For affine LIVs, we also identify the update value, which we refer to as the *stride*). In addition, any scalar integer or pointer computed as a function of an LIV using one of the recognized update methods is also an LIV (we call these *derived LIVs*). Figures 2a and b illustrate the LIVs identified by our compiler for Bzip2 and MCF, respectively. The shaded boxes highlight the LIV update expressions, and the circled labels indicate the derived LIVs. For Bzip2, our analysis identified 7 affine LIVs, one for each of the loops "L1" to "L7," and 2 derived LIVs, "D1" and "D2." For MCF, our analysis identified an affine and pointer-chasing LIV in loop "L9," a derived LIV "D3" also in loop "L9," and a pointer-chasing LIV in loop "L10." No LIVs were detected for loop "L8." Note, when analyzing each loop, our compiler considers code local to that loop only. Code within nested loops and called procedures are ignored.

After identifying LIVs, our compiler identifies loop-varying memory references (LMRs). An LMR is either an array reference that uses an affine LIV as an index, or a dereference of a pointer-chasing LIV. We call these *affine LMRs* and *pointer-chasing LMRs*, respectively. In addition, our compiler also identifies all scalar integer or pointer variables computed as a function of LMRs. We call these *derived LMRs*. Since derived LMRs are themselves functions of LIVs, memory references that perform indexing or indirection through derived LMRs are also LMRs. Hence, our compiler must identify LMRs iteratively, using derived LMRs to discover additional LMRs and vice versa, and stopping only when no new LMRs are found. Figure 2 reports the identified LMRs via underlining: underlined array and pointer expressions are LMRs, while underlined scalar variables are derived LMRs. For example, the array reference "zptr[bbStart+j]" in loop "L4" is an affine LMR yielding the derived LMR, "a2update." In turn, "a2update" yields two additional affine LMRs since it is used as an index into the array, "quadrant," in two different expressions.

Next, our compiler computes the *level of indirection* for each identified LMR. This is the number of chained array references for affine LMRs, or the number of chained pointer dereferences for pointer-chasing LMRs. Figure 2 reports the level of indirection for each LMR using a superscripted value. Notice chaining can occur transitively through derived LMRs, increasing the levels of indirection. For example, the affine LMR "bigDone[c1]" in loop "L6" has 3 levels of indirection because "c1" is derived from the LMR "block[zptr[j]-1]" which itself has 2 levels of indirection.

Finally, our compiler identifies the sparse memory references. Sparse memory references are LMRs that satisfy one of three criteria. First, affine LMRs with 1 level of indirection are sparse only if their affine LIVs are updated with a large stride. These are the large-stride affine array traversals (*i.e.* Figure 1a). Second, all affine LMRs with 2 or more levels of indirection are sparse. These are the indexed array traversals (*i.e.* Figure 1b). Finally, all pointer-chasing LMRs, regardless of how many levels of indirection, are sparse. These are the pointer-chasing traversals (*i.e.* Figure 1c). In



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

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

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

Figure 3: 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.

Figure 2, the final sparse memory references identified by our analysis appear in bold-face.

3.2 Computing Cache Miss Transfer Size

To realize the potential bandwidth savings afforded by sparse memory references, the compiler must determine the amount of data that 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 reuse and increased cache misses, offsetting the gains of conserving memory bandwidth. Our approach analyzes spatial reuse within a *single loop iteration*, and then selects a transfer size that exploits the detected spatial locality.

Our compiler computes a cache miss transfer size for each sparse memory reference in the following manner. For each array element or link node referenced within a loop body, our compiler examines all the sparse memory references identified by the analysis in Section 3.1.2 that contribute an access. If there is only one contributing sparse memory reference to the array element or link node, we assume there is no spatial reuse and we set the transfer size for the memory reference equal to the size of the memory reference itself. For example, all the sparse memory references identified for Bzip2 in Figure 2a fall into this category.

If, however, each array element or link node is accessed by multiple sparse memory references in the loop body, then we compute 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 3a shows a linked-list traversal loop from Health, a benchmark from the Olden suite [17], 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 reuse.

	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: Mnemonics 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.

To select the transfer size for multiple memory references to a compound structure, our compiler considers 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 3b demonstrates our compiler's transfer size selection algorithm on the Health benchmark. As illustrated in Figure 3a, 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 3b 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.

3.3 Implementation Issues

Annotated Memory Instructions. Our compiler requires several new memory instructions to encode the transfer size information described in Section 3.2. These annotated memory instructions replace normal memory instructions at the points in the code where sparse memory references have been identified by the analysis described in Section 3.1.2. 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.

The first 4 columns of Table 1 list the annotated load and store instructions assumed by our compiler. We support annotations for load and store word, as well as load and store double word.

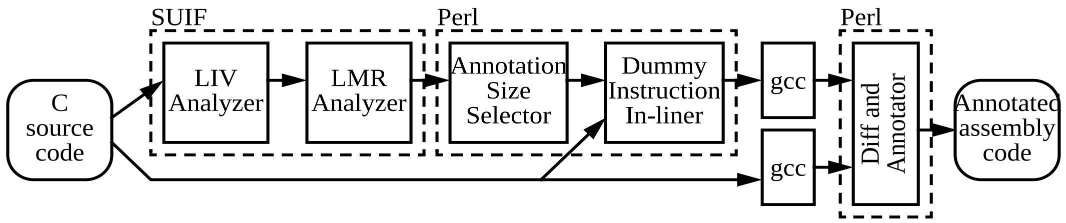


Figure 4: Prototype compiler. LIV and LMR analyses are performed in SUIF. Annotation size selection is performed in Perl. Annotation insertion is performed using a combination of Perl and gcc.

For each instruction type, we support 3 power-of-two annotation sizes: 8, 16, and 32 bytes. Since the size annotations are restricted, we cannot annotate a memory reference with an arbitrary size value. Consequently, all transfer sizes computed using the technique described in Section 3.2 must be rounded up to the next available power-of-two size annotation.

Prototype Compiler. We implemented our algorithms for instrumenting sparse memory references in a prototype C compiler. Figure 4 illustrates the major components in our compiler and the tools used to implement them. The algorithms for identifying LIVs, LMRs, and sparse memory references are implemented using the Stanford University Intermediate Format (SUIF) framework [10]. For each identified sparse memory reference, our SUIF tool outputs a variable name and type, and a module name and line number where the memory reference appears. A Perl script uses this information to select an annotation size for each identified sparse memory reference. Currently, our compiler analyzes indexed array and pointer chasing traversals only. We did not implement the analsis for large-stride affine array traversals because these traversals almost never occur in the benchmarks we studied (see Section 5.1).

The last step performed by our compiler is to replace each normal memory instruction in the program assembly code corresponding to an identified sparse memory reference with an appropriate annotated memory instruction. Unfortunately, SUIF provides only symbolic and source code line number information, which is not enough to uniquely locate the assembly instructions requiring size annotations. Rather than map SUIF's source-level coordinates into assembly-level coordinates, we in-line a dummy instruction into the C code using an "`asm`" directive that artificially consumes the data loaded by each candidate sparse memory reference. When compiled using `gcc`, the dummy instructions reveal the sparse loads and stores in the assembly code. To remove the effect of the dummy instructions, we also generate the assembly code without the `asm` directives. A third Perl script compares the two to identify and annotate the sparse loads and stores in the assembly code from the original program source code.

Software Prefetching. Although our compiler only analyzes normal memory instructions, our technique can also be applied to software prefetching. In software prefetching, prefetch instructions are inserted to prefetch memory references that cache miss frequently. Similar to load and store instructions, memory bandwidth can be conserved if prefetch instructions fetch only the data that is likely to be referenced by the processor. We provide special prefetch instructions that carry size annotations of 8, 16, and 32 bytes, as indicated by the last column of Table 1. Furthermore, we use the same information extracted by our compiler for normal memory instructions to instrument annotated prefetch instructions. For each prefetch instruction, we examine the corresponding load or store being prefetched. If our compiler identified that load or store as a sparse reference, we replace the prefetch with a special prefetch whose size annotation matches the size computed by our compiler. Otherwise, we use a conventional (non-annotated) prefetch instruction.

In this paper, we analyze and instrument normal memory instructions automatically using our prototype compiler. To evaluate the impact of our technique on software prefetching, we create software prefetching versions of our benchmarks by hand, and use the information computed by our compiler for normal memory references to guide annotation of prefetch instructions. Although the prefetch instructions are inserted manually, it is important to emphasize that the analysis of the sparse memory references is still automatic. Hence, the performance gains due to memory bandwidth conservation, even for our prefetching experiments, can be attributed to our compiler.

4 Hardware Support for Conserving Memory Bandwidth

Annotated memory instructions fetch 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, 1, 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

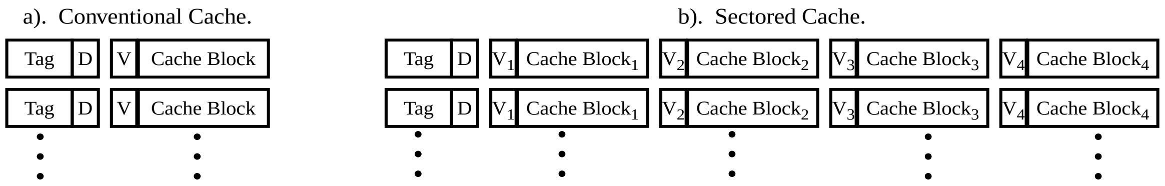


Figure 5: a). A conventional cache provides one tag for each cache block. b). A sectored cache provides one tag for all the cache blocks in a sector, hence reducing the tag overhead. “V” denotes a valid bit, and “D” denotes a dirty bit.

discusses these two hardware issues in more detail.

4.1 Sectored Caches

To exploit the potential bandwidth savings afforded by our compiler, a small cache block is required. One drawback of small cache blocks is high tag overhead. Fortunately, high tag overhead can be mitigated using a *sectored cache*, as is done in [14] and [1]. 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 5.[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 by multiple blocks, however, cache tag overhead is small even when the cache block size is small.

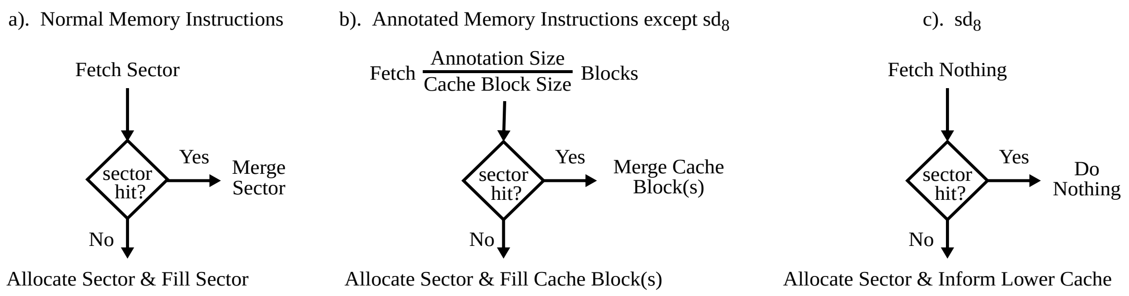
4.2 Variable Fetch Size

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 the compiler. Figure 6 specifies the actions taken on a cache miss that implements a compiler-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 6a-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 6a), 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 [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.



a). Normal Memory Instructions
Fetch Sector
sector hit?
Yes
Merge Sector
No
Allocate Sector & Fill Sector
b). Annotated Memory Instructions except sd8
Fetch Annotation Size / Cache Block Size Blocks
sector hit?
Yes
Merge Cache Block(s)
No
Allocate Sector & Fill Cache Block(s)
c). sd8
Fetch Nothing
sector hit?
Yes
Do Nothing
No
Allocate Sector & Inform Lower Cache

Figure 6: 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 6b), 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 6c. 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 compiler-driven selective sub-blocking for uniprocessors. Our tech-

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)	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).

nique can also be applied to multiprocessors; however, doing so requires integrating sectored caches with cache coherence protocols. (Note, this problem is common to all selective sub-blocking techniques, not just ours). One approach is to maintain coherence at the cache block granularity, and to break sector-level transactions into multiple cache block transactions. 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 our compiler algorithms 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 [19]. The next two benchmarks are from the Olden suite [17]. 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 the SPECint2000 suite. BZIP2 is a data compression algorithm that performs indexed array accesses, and MCF is an optimization solver that traverses a highly irregular linked

Cache Model	Sector Size	Block Size	Associativity	Replacement Policy	Write-Hit Policy
Conventional	-	64 bytes	2-way	LRU	Write-back
Compiler-Driven	64 bytes	8 bytes	2-way	LRU	Write-back
Oracle	64 bytes	8 bytes	2-way	LRU	Write-back
SFP	64 bytes	8 bytes	2-way	LRU	Write-back

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

data structure. Only versions of our benchmarks without software prefetching are considered in this section. (The software prefetching versions exhibit the same cache behavior since the size annotations for prefetch instructions are derived from normal memory instructions).

Using these benchmarks, we perform a series of cache simulations designed to characterize the benefits and drawbacks of compiler-driven selective sub-blocking. We first compare our technique against a conventional cache to study the memory traffic reductions achieved by our compiler. We then compare our technique against a perfect "Oracle cache" to quantify the opportunities for traffic reduction missed by our compiler. 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 for the PISA ISA to provide the cache models necessary for our experiments. We added sectored caches and the annotated memory instructions in Table 1 to model our technique. Finally, we also built models for the Oracle and SFP caches, which we will describe later in Sections 5.3 and 5.4, respectively. Table 3 presents the cache parameters used for our experiments. Notice, the sector size belonging to the sectored caches (under Compiler-Driven, Oracle, 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

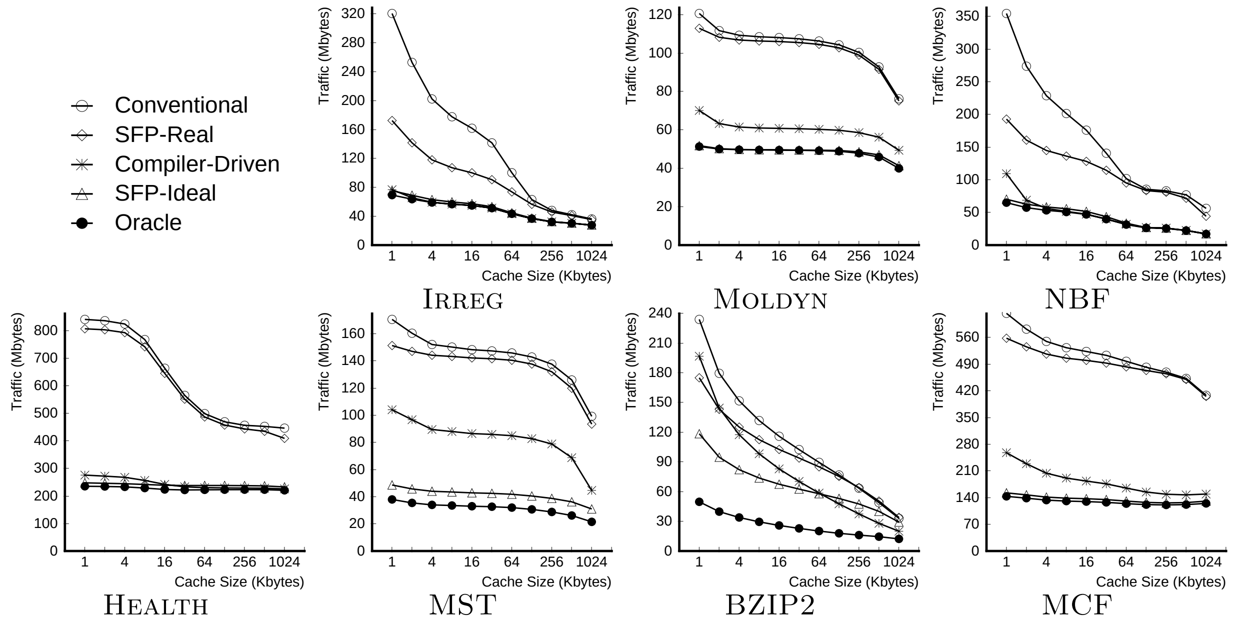


Figure 7: Cache traffic as a function of cache size. Traffic is reported for a conventional cache (Conventional), a compiler-driven selective sub-blocking cache (Compiler-Driven), an oracle cache (Oracle), and two SFP caches (SFP-Ideal and SFP-Real). All traffic values are in units of MBytes.

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 7 plots cache traffic as a function of cache size for a conventional cache (Conventional), a compiler-driven selective sub-blocking cache (Compiler-Driven), an Oracle cache (Oracle), 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 Compiler-Driven and Conventional curves in Figure 7, we see our compiler reduces cache traffic significantly compared to a conventional cache. For NBF, Health, and MCF, our compiler reduces between 58% and 71% of the cache traffic. Furthermore, these reductions are fairly constant across cache size, indicating our technique is effective in both small and large caches for these benchmarks. For Irreg, our compiler is effective at small cache sizes, reducing traffic by

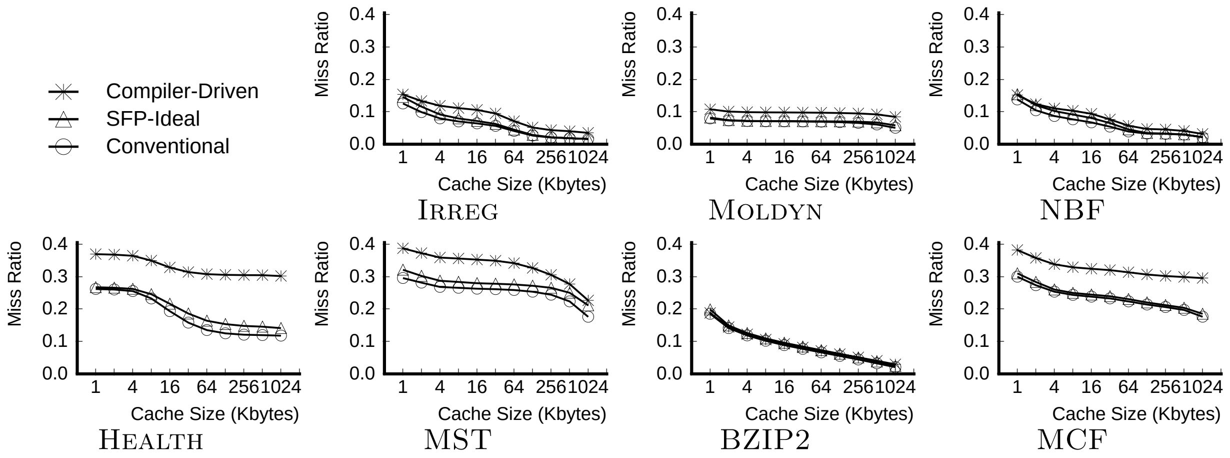


Figure 8: Cache miss rate as a function of cache size. Miss rates are reported for a conventional cache (Conventional), a compiler-driven selective sub-blocking cache (Compiler-Driven), and an SFP cache (SFP-Ideal).

57% or more for caches 64K or smaller, but loses its effectiveness for larger caches. Irreg accesses 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 spatial locality exploitation increases in Irreg, our compiler loses its advantage. Finally, for Moldyn, MST, and BZIP2, the traffic reductions are 42%, 43%, 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 MST and BZIP2 will be explained later in Section 5.3.

Figure 8 plots cache miss rate as a function of cache size for the "Conventional," "Compiler-Driven," and "SFP-Ideal" caches in Figure 7. Comparing the Compiler-Driven and Conventional curves in Figure 8, we see that the traffic reductions achieved by our compiler come at the expense of increased cache miss rates. Miss rate increases range between 11% and 43% for Moldyn, NBF, MST, BZIP2, and MCF, and 74% and 98% for Irreg and Health, respectively.

The higher miss rates incurred by our compiler are due to the inexact nature of its spatial locality analysis described in Section 3.2. For indexed array and pointer-chasing references, our compiler performs analysis only between references within a single loop iteration. It does not detect spatial locality across different loop iterations because the separation of inter-iteration references depends on runtime values that are not known statically. Hence, our size annotations are overly conservative, missing opportunities to exploit spatial locality whenever multiple indirect references 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.

5.3 Missed Opportunities for Traffic Reduction

We now study to what extent our compiler misses opportunities for reducing memory traffic. To facilitate our study, we model a perfect sectored cache, called the "Oracle cache," which represents the best memory traffic that *any* selective sub-blocking technique can possibly achieve. The 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 simulate the Oracle cache, 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*.

Figure 7 shows for IRREG, NBF, and HEALTH, our compiler essentially achieves the minimum traffic since the Oracle and Compiler-Driven curves overlap almost completely. However, for MOLDYN and MCF, the Oracle cache incurs roughly 25% less traffic, and for MST and BZIP2, roughly 60% less traffic, indicating our technique can be improved for these applications. Three factors contribute to the missed opportunities for further traffic reduction. The first is due to the restrictions placed on annotated memory instructions. Rounding up annotations to the nearest power-of-two size (Section 3.3) and aligning cache misses to an annotation-sized boundary (Section 4.2) leads to wasteful data fetching. This accounts for the additional improvements possible in MOLDYN.

The two remaining factors have to do with limitations in our compiler. As described in Section 3.1.2, our compiler performs local (*e.g.* intra-procedural) analysis only. Hence, sparse memory references contained in loops that straddle multiple procedures may elude identification by our compiler, limiting the achievable memory traffic reduction. This accounts for much of the additional improvements possible in both MST and MCF. Finally, our compiler only looks for the access patterns illustrated in Figure 1. Other sparse memory reference patterns are not analyzed. In BZIP2, for example, most sparse memory references occur in unit-stride affine array loops. Normally, such loops access memory densely, so our compiler does not identify them. But in BZIP2, these loops execute only 1 or 2 iterations, resulting in sparse memory accesses.

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, 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 proposed in [14].

The "SFP-Ideal" curves in Figure 7 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 7 shows our compiler achieves close to the same traffic as SFP-Ideal for Irreg, NBF, and Health. For Moldyn, MCF, and MST, however, SFP-Ideal reduces 19%, 24%, and 49% more traffic, respectively. Finally, in BZIP2, SFP-Ideal outperforms our compiler for small caches, but is slightly worse for large caches. Figure 7 demonstrates that except for MST, our compiler achieves comparable traffic with an aggressive SFP despite using much less hardware. Comparing miss rates, however, Figure 8 shows SFP-Ideal outperforms our compiler, essentially matching the miss rate of a conventional cache. As discussed in Section 5.2, our compiler 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 7 report the traffic of an SFP using an 8K-entry SHT. The SHT in SFP-Real is 32 Kbytes. Figure 7 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 compiler 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. The additional simulations show the same qualitative results as those presented in Figures 7 and 8.

6 Performance Evaluation

This section continues our evaluation of compiler-driven selective sub-blocking by measuring performance on a detailed cycle-accurate simulator.

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 = 1/10 cycles 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.

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 "Compiler-Driven" cache models from Section 5 only. 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 our processor and cache models.

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 [15]). These parameters correspond to a 60 ns (120 processor cycles) L2 sector fill latency and a 2 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 our compiler benefit the memory sub-system only (though the cache miss increases impact both the L1 and L2 caches). We expect traffic reductions across the L1-L2 bus provided by our compiler can also increase performance, but our simulator does not quantify these effects.

As discussed in Section 3.3, we manually inserted software prefetching into our benchmarks to study the impact of reduced memory traffic afforded by our compiler in the context of prefetching.

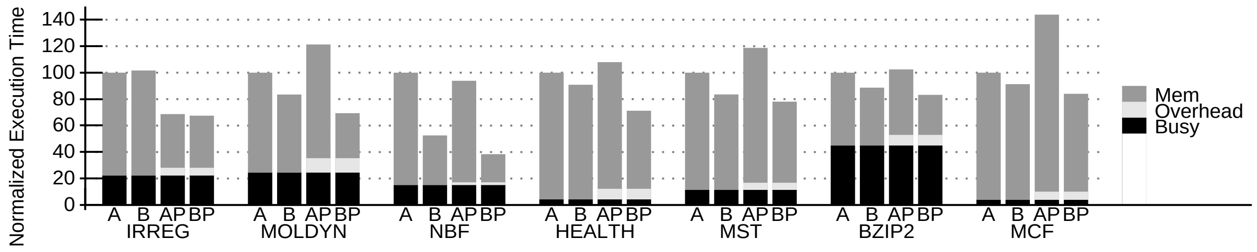


Figure 9: Execution time breakdown for compiler-driven selective sub-blocking. Individual bars show performance using a conventional cache versus a compiler-driven selective sub-blocking cache, first without prefetching (bars "A" and "B"), then with prefetching (bars "AP" and "BP").

For affine array and indexed array references, we use Mowry's prefetching algorithms [16]. For pointer-chasing references, we use Karlsson's prefetch arrays technique [13].

6.2 Performance of Compiler-Driven Selective Sub-Blocking

Figure 9 shows the performance of compiler-driven selective sub-blocking on the baseline memory system described in Section 6.1. For each application, we compare the execution time using a conventional cache versus a compiler-driven selective sub-blocking cache, first without prefetching ("A" and "B" bars), and then with prefetching ("AP" and "BP" 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 "A" bars assuming a perfect memory system (*e.g.* all memory accesses take 1 cycle). "Overhead" is the incremental increase in execution time of the "AP" and "BP" bars over "Busy," again on a perfect memory system. "Mem" is the incremental increase in execution time over "Busy"+"Overhead" on a real memory system. All times are normalized against the "A" bars.

First, we examine performance without prefetching. Comparing the "A" and "B" bars in Figure 9, we see our compiler increases performance for 6 out 7 applications, reducing execution time by as much as 47% (MCF), and by 15% on average. The cache traffic reductions achieved by our compiler 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 our compiler increases the cache miss rate due to reduced exploitation

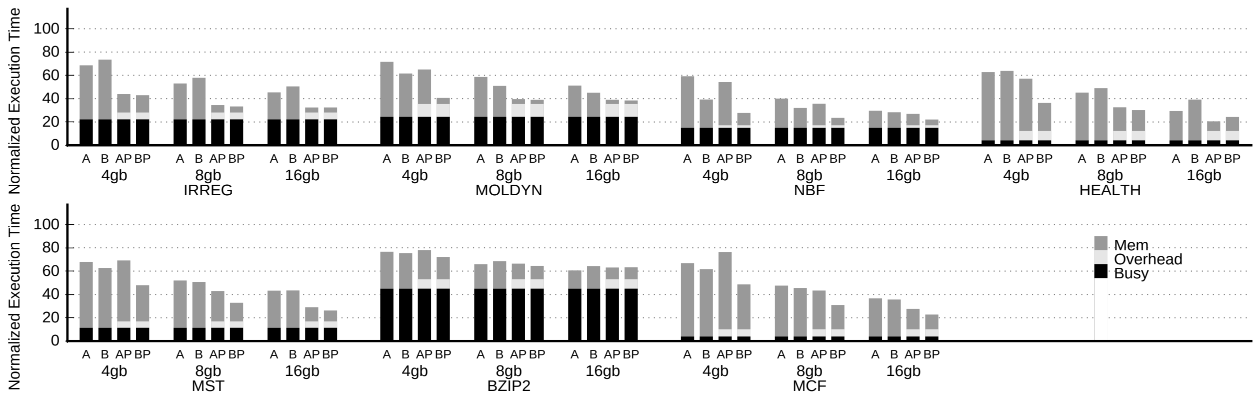


Figure 10: Execution time breakdown for compiler-driven selective sub-blocking at 2, 4, and 8 GB/s. Individual bars show performance using a conventional cache versus a compiler-driven selective sub-blocking cache, first without prefetching (bars "A" and "B"), then with prefetching (bars "AP" and "BP").

of spatial locality. Figure 9 shows 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, our compiler does not provide much traffic reduction for Irreg at large cache sizes. Hence, the increased cache miss rate results in a 2% performance loss for Irreg.

Next, we examine prefetching performance. Comparing the "AP" and "A" bars in Figure 9, we see that software prefetching on a conventional cache degrades performance for 5 applications (Moldyn, Health, MST, BZIP2, and MCF), resulting in an 8% degradation averaged across all benchmarks. Prefetching adds software overhead. It also increases memory traffic due to speculative prefetches and the addition of prefetch pointers for prefetching linked data structures [13]. 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 "BP" and "A" bars, however, we see software prefetching coupled with compiler-driven selective sub-blocking achieves a performance gain for all 7 applications, 29% on average. The addition of prefetching increases memory contention, thus magnifying the importance of reduced traffic provided by our compiler. Also, the increase in cache miss rate incurred by our inexact compiler analysis is less important when performing prefetching because the additional cache misses will themselves get prefetched, hiding their latency. Thus, compiler-driven selective sub-blocking becomes more effective when coupled with software prefetching.

6.3 Bandwidth Sensitivity

This section examines the sensitivity of our baseline results reported in Section 6.2 to available memory bandwidth. We run the "A," "B," "AP," and "BP" versions from Figure 9 using higher memory system bus bandwidths. 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 Access Strobe" and "Column Access Strobe" parameters by 30%.

In Figure 10, we report performance at 4, 8, and 16 GB/s memory system bus bandwidths. All bars have been normalized against the "A" versions at 2 GB/s from Figure 9. Without prefetching (*i.e.* comparing the "A" and "B" bars), Figure 10 shows that the performance boost achieved by our compiler decreases with increasing bandwidth. By 16 GB/s, three applications (Irreg, Health, and BZIP2) perform noticeably worse. As memory bandwidth increases, there is less opportunity to reduce memory contention and data transfer latency; hence, our compiler loses its benefit, and the increased cache miss rates reported in Section 5.2 result in performance degradations.

With prefetching (*i.e.* comparing the "AP" and "BP" bars), we see that again our compiler's performance advantage reduces as memory bandwidth increases. However, our compiler always achieves higher or equal performance compared to a conventional cache, with the exception of Health at 16 GB/s where we suffer a degradation of 18%. When combined with software prefetching, the performance of our compiler is robust to available bandwidth because prefetching neutralizes the drawbacks of increased cache misses. This result reinforces the earlier claim that compiler-driven selective sub-blocking is particularly effective in combination with software prefetching.

Although our compiler becomes 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 9 given the trends in processor and memory speeds.

7 Conclusion

This paper studies compiler-driven selective sub-blocking. We present compiler support for conserving memory bandwidth by identifying sparse memory references via static analysis, and conveying spatial reuse information associated with such references to the memory system. Our compiler removes between 31% 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 comparable amount of traffic as hardware selective sub-blocking using an 8 Mbyte predictor for 6 applications. These traffic reductions come at the expense of increased cache miss rates, ranging between 11% and 43% for 5 applications, and 74% and 98% for 2 applications. Overall, we show our compiler provides a 15% performance gain. Furthermore, performance gains improve when our technique is coupled with software prefetching, enabling a 29% performance gain, compared to an 8% performance degradation when prefetching without compiler-driven selective sub-blocking. Based on our results, we conclude that using a compiler to control selective sub-blocking is a promising approach, 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] D. Burger. Hardware Techniques to Improve the Performance of the Processor/Memory Interface. Technical report, Computer Science Department, University of Wisconsin-Madison, December 1998.

[2] 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.

[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 HPCA-5*, pages 70–79, Jan '99.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[10] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S.-W. Liao, E. Bugnion, and M. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE COMPUTER*, 29(12), December 1996.

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

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

[17] 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.

[18] 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.

[19] 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

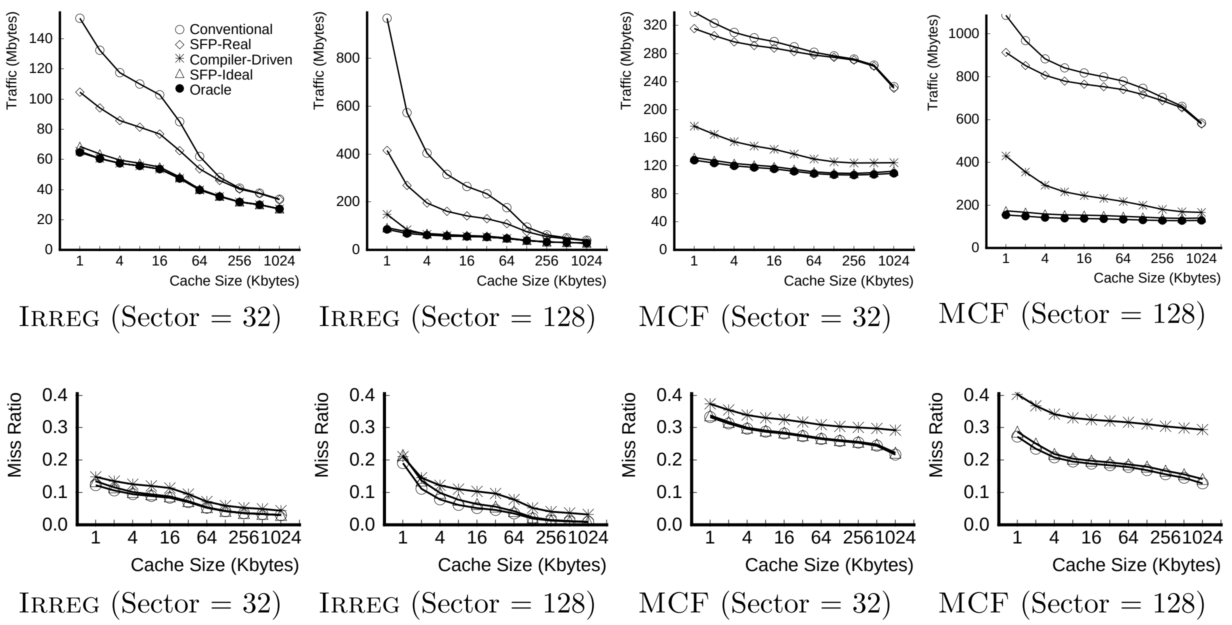


Figure 11: Sensitivity to sector size variations. Cache traffic and miss rate results, similar to those in Figures 7 and 8, for Irreg and MCF with sector sizes of 32 bytes and 128 bytes.

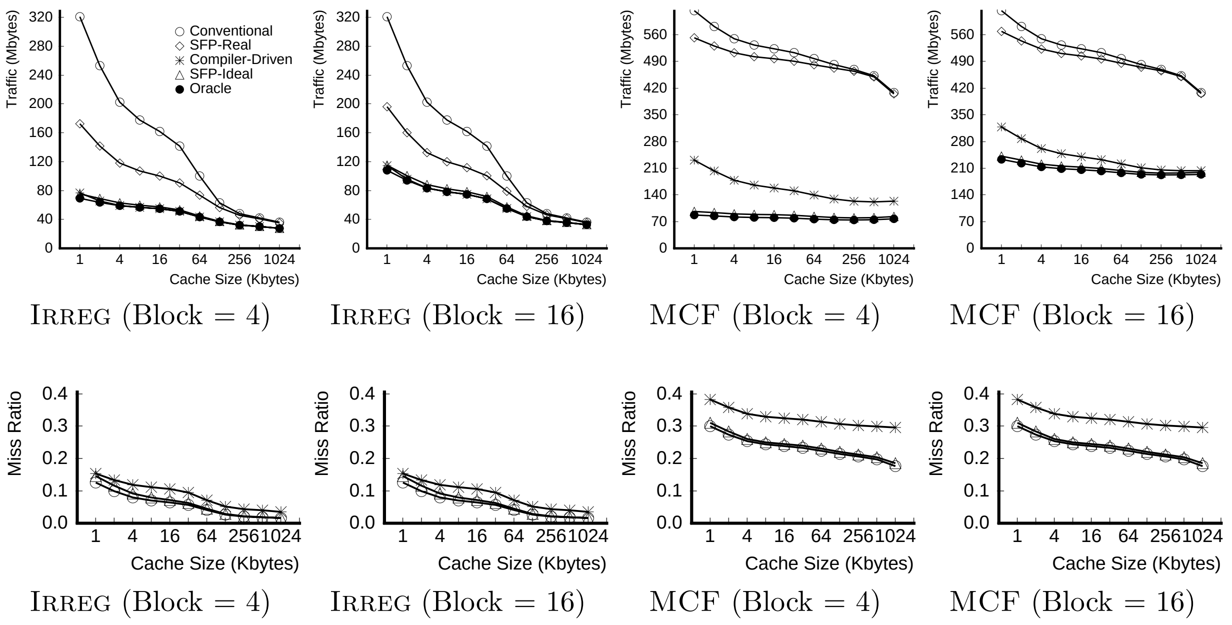


Figure 12: Sensitivity to cache-block size variations. Cache traffic and miss rate results, similar to those in Figures 7 and 8, for Irreg and MCF with cache-block sizes of 4 and 16 bytes.