Guided Region Prefetching: A Cooperative Hardware/Software Approach

Zhenlin Wang[†] Doug Burger [§] Kathryn S. McKinley [§] Steven K. Reinhardt[‡] Charles C. Weems[†]

[†]Dept. of Computer Science, Univ. of Massachusetts, Amherst
[§]Dept. of Computer Sciences, The University of Texas at Austin
[‡]Dept. of EECS, University of Michigan

Abstract

Despite large caches, main-memory access latencies still cause significant performance losses in many applications. Numerous hardware and software prefetching schemes have been proposed to tolerate these latencies. Software prefetching typically provides better prefetch accuracy than hardware, but is limited by prefetch instruction overheads and the compiler's limited ability to schedule prefetches sufficiently far in advance to cover level-two cache miss latencies. Hardware prefetching can be effective at hiding these large latencies, but generates many useless prefetches and consumes considerable memory bandwidth. In this paper, we propose a cooperative hardware-software prefetching scheme called Guided Region Prefetching (GRP), which uses compiler-generated hints encoded in load instructions to regulate an aggressive hardware prefetching engine. We compare GRP against a sophisticated pure hardware stride prefetcher and a scheduled region prefetching (SRP) engine. SRP and GRP show the best performance, with respective 22% and 21% gains over no prefetching, but SRP incurs 180% extra memory traffic—nearly tripling bandwidth requirements. GRP achieves performance close to SRP, but with a mere eighth of the extra prefetching traffic, a 23% increase over no prefetching. The GRP hardware-software collaboration thus combines the accuracy of compiler-based program analysis with the performance potential of aggressive hardware prefetching, bringing the performance gap versus a perfect L2 cache under 20%.

1 Introduction

Modern out-of-order processors can tolerate latencies for multi-cycle level-one cache hits, and many of the level-one cache misses that result in level-two hits [42]. However, the hundreds of cycles that result from DRAM accesses cannot be tolerated, thus causing significant performance degradations. For the SPEC2000 benchmarks running on a modern, high-performance microprocessor, over half of the time is spent stalling for loads that miss in the level-two cache [28]. We observe similar results in our simulations for a subset of SPEC2000 benchmarks and Sphinx, a speech recognition application [27]. Figure 1 compares the performance of a system with a realistic memory hierarchy versus one with a perfect L1 cache and one with a perfect L2 cache, in the leftmost stacked bar for each benchmark. The benchmarks are sorted by the size of the gap between a realistic system and one with a perfect L2 cache, a geometric mean performance gap of 33.7%. As a summary of our results, we also show the performance afforded by the traffic-efficient GRP L2 prefetching scheme, displayed as the rightmost bar for each benchmark.

To tolerate these latencies, researchers have proposed a large number of both software and hardware prefetching schemes. Each of these two classes of prefetch solutions have distinct advantages and drawbacks. Pure software prefetching is typically highly accurate, but incurs runtime overhead and cannot issue prefetches sufficiently far in advance of a load to hide main memory access latencies [28]. Hardware only schemes can prefetch spatial regions [10, 11, 22, 34, 38], pointer chains [14, 21, 35], or recurring patterns [26]. While these schemes can hide much of the main memory access time, they can also consume substantial amounts of memory bandwidth. This additional traffic need not degrade uniprocessor performance, but it increases power consumption, and will likely degrade performance on multiprocessors. Since off-chip bandwidth will be the dominant limiter of scalability for future chip multiprocessors (CMPs) [20], prefetch schemes that consume bandwidth inefficiently will not be practical. While some schemes throttle prefetching when the accuracy drops below a threshold, they then miss opportunities for issuing useful prefetches [16].

In this paper, we propose a cooperative hardware/software prefetch framework called Guided Region Prefetching (GRP). In GRP, sophisticated compiler analysis is used to produce a rich set of load hints, including the presence or absence of spatial locality, pointer structures, or indirect array accesses. A runtime hardware engine, triggered by L2 cache misses, generates prefetches based on the compiler's hints. GRP thus benefits from compiler analysis of application reference patterns, but—unlike traditional software prefetching—the compiler is not required to generate or schedule individual prefetch addresses. Because the hardware generates the prefetches, it can run far ahead of the missing references. Because the compiler guides it, the hardware need not struggle to deduce future references with complex pattern matching on prior accesses stored in large tables.

Using previously proposed techniques [28], the GRP hardware prefetching engine keeps uniprocessor bus contention low by prefetching only when the memory bus is otherwise idle, and keeps cache pollution low by loading prefetches into the LRU set of the L2 cache. Without compiler support, this prefetching hardware is effective at improving performance, but consumes copious bandwidth. Through GRP, the compiler informs the hardware of application reference patterns, enabling the hardware to prefetch only when it is likely to be effective. We evaluate compiler hints that mark loads with the following hints: *spatial*–prefetch the spatial region around a load; *size*–how many lines to fetch on a spatial reference; *pointer*–prefetch by following the pointer in the load's cache line; *recursive*–prefetch this pointer data structure recursively. For *size* hints, the compiler can encode a *variable-size region* that specifies how much to prefetch based on enclosing loop bounds, instead of using a fixed value. The compiler also generates indirect prefetching instructions which trigger prefetching a set of references using an indirection array.

This cooperative GRP hardware/software interface improves the high performance of the previously proposed scheduled region prefetching (SRP) [28] by over 10% on two of the SPEC2000 benchmarks, and match the performance of SRP on the rest. Table 1 shows a summary of the GRP results using the geometric mean. We show GRP both with (GRP/Var) and without (GRP/Fix) variable-size region prefetching. Without prefetching, the mean

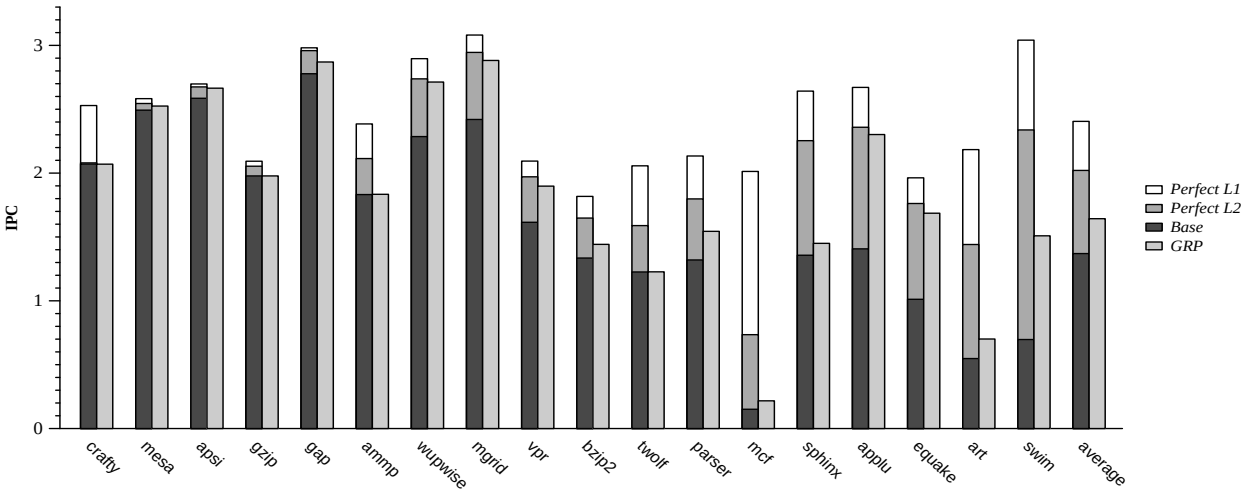

Figure 1: Processor performance

	Speedup	traffic increase	Performance gap from perfect L2
No prefetching	1	1	33.72
Stride prefetching	1.147	1.09	23.99
SRP	1.226	2.80	18.75
GRP/Fix	1.216	1.62	19.42
GRP/Var	1.212	1.23	19.69

Table 1: Summary of prefetching performance and traffic

performance across the benchmark suite is 33.7% lower than a perfect level-two cache. Stride prefetching (using the Sherwood et al. design [38]) provides a 15% speedup over no prefetching. SRP, which uses no compiler analysis, outperforms stride prefetching by 7%, but consumes excessive memory bandwidth, a 180% increase over a system with no prefetching. GRP provides near-equivalent performance to SRP but with substantially less traffic, an increase of only 23% over the not prefetching. This reduction in traffic saves power and is more amenable to multiprocessor systems, where additional traffic can directly affect performance. Both SRP and GRP still incur a 19% gap versus a perfect L2.

We review related work in Section 2, showing that much of it does not pursue a balance between aggressive prefetching and efficient use of memory bandwidth. Section 3 describes the hardware used for the GRP hardware prefetching engine, and how it uses the hints. Section 4 describes the compiler analysis in detail. Section 5 evaluates the degree to which a GRP engine can bring the performance of most benchmarks close to that of a perfect L2 cache while keeping memory traffic increases small. Section 5 also compares the performance of GRP to that of stride prefetching [38]. We conclude in Section 6 that GRP eliminates main memory accesses as a source of performance loss for all but four of the SPEC2000 benchmarks and *sphinx*. Of those four, one simply requires more memory bandwidth, and can benefit from more sophisticated software/hardware cooperation.

2 Related Work

In this section, we focus on the most pertinent aspects of the large body of literature on software and hardware data prefetching, along with the small number of previously proposed hybrid schemes.

Software prefetching relies on non-binding prefetch instructions that bring the indicated block of memory into the cache, much like a load instruction. Conceptually, the latency of a given load instruction is hidden by inserting a prefetch with the same effective address into the instruction stream sufficiently far in advance of the load. Because the compiler only inserts prefetches for known (or very likely) loads, software prefetch accuracy is typically high. In practice, the compiler faces two key challenges in data prefetching: *selection* and *scheduling*.

Because prefetch instructions occupy instruction cache space, pipeline slots, and data cache ports, the compiler must *select* a subset of the loads for which to generate prefetches. Accurate compile-time identification of the loads that will cause cache misses at run-time is complex, requiring both knowledge of hardware parameters (cache block size, capacity, and associativity) and sophisticated code analysis (e.g., to determine the volume of other data accessed between references to a particular block) [7, 17, 33, 45].

The compiler also faces the difficult challenge of issuing the prefetches sufficiently early to hide the memory latency, but not so early that useful data are needlessly evicted. To find that point, the compiler must estimate cache miss latencies and run-time instruction execution rates [25]. The compiler is further constrained in that it cannot schedule a prefetch until it can compute the effective address. While this constraint is not significant for arrays [6, 33], it limits compiler-based greedy pointer prefetching [5, 30, 36]. Jump pointers bypass this limitation by identifying records several links ahead in the structure, but require much more sophisticated analysis, dynamic updates, and the addition of a jump pointer to each object [5, 30, 36]. Other approaches prefetch pointer arguments at call sites [29], and decouple prefetches from the main program using a separate thread context [13, 24, 31].

The converse approach is hardware-only prefetching, in which the hardware predicts prefetch addresses by observing a program's runtime behavior. Since prefetches do not not incur overhead in the processor itself, the hardware need not be as selective about issuing prefetch operations. Recent work shows that simple dynamic prioritization techniques eliminates memory bandwidth contention and cache pollution problems [28]. However, unlike the compiler, the hardware has no direct knowledge of future memory references; the key challenge in hardware-based prefetching is determining a reasonable set of predicted addresses to use as prefetch targets. Hardware prefetching thus suffers relative to software prefetching in both accuracy (because the predictions may be wrong) and coverage (because some addresses may require the compiler's scope to

predict).

Many hardware prefetchers exploit only spatial locality, prefetching one or more subsequent blocks on a cache miss [15, 22, 40]. More sophisticated schemes detect non-unit strided access patterns, such as Chen and Baer's reference prediction table (RPT) [10] and Palacharla and Kessler's strided stream buffers [34]. Other approaches exploit pointer-based access sequences, as with correlation-based and Markov prefetching [1, 9, 21], or a broader class of patterns, using dead block information [26]. Another approach involves decoupling the data structure traversal from the computation, using specialized pointer-traversal hardware [35] or dedicated pre-execution hardware [2]. Researchers have also proposed memory-side prefetching to reduce latencies between prefetches [19, 41, 46].

Most pertinent to this work are two previous papers. First, predictor-directed stream buffering, proposed by Sherwood et al. [38], unifies strided stream buffers and Markov prefetching into a single, consistent hardware prefetching framework. In Section 5, we compare the GRP scheme to the strided stream buffers scheme only, since the Markov predictor consumes too much state to be practical. Second, Cooksey et al. [14] propose a stateless approach to pointer prefetching, foregoing explicit identification of pointer traversal patterns and simply prefetching any referenced memory value that could be reasonably interpreted as a memory address. Our hardware schemes are also stateless. We find that for our benchmarks, GRP with spatial hints usually performs better or the same as pointer prediction with or without pointer hints.

In the end, all hardware schemes are forced to trade coverage for accuracy (or vice versa), and focus either only on structured access patterns which can be predicted with high accuracy (forgoing coverage of less structured access patterns), or consume significant amounts of bandwidth with incorrect prefetches in an attempt to cover less-structured references.

The relative strengths and weaknesses of hardware and software prefetching are complementary and thus suggest a combined hardware/software approach. An ideal scheme would exploit the compiler's knowledge of future reference patterns, and use a low-overhead channel to convey this information to a hardware prefetching engine, which could then generate and schedule appropriate prefetches based on dynamic information regarding cache miss events and resource availability.

The limited previous work in this area has either exploited prefetching for restricted classes of access patterns, or provided an interface that is overly general and complex. On the conservative side, Gornish and Veidenbaum [18] let software select the number of contiguous blocks to prefetch upon a miss, whereas Chen and Baer [11, 12] use the compiler to supply address and stride information to augment a reference prediction table. Skeppstedt and Dubois use a trap handler to trigger prefetching using similar information [39]. Karlsson et al. [23] use *prefetch arrays* to enable a hardware engine to perform a generalized variant of greedy and jump-pointer prefetching. Zhang and Torrellas [47] use the compiler to mark blocks in memory as belonging to contiguous spatially local regions or containing indirection pointers. Their scheme requires additional bits in main memory and significant support in the memory controller. Finally, fully programmable prefetch engines provide flexibility but require significant memory system support and have not yet demonstrated that the required compiler support is realistic [41, 43, 46].

GRP combines the advantages of both software and hardware prefetching in a scheme that is simple yet effective. It conveys sophisticated compiler analysis by associating a range of hints with loads, which an aggressive, simple, and general hardware prefetcher uses only when necessary. Thus, the pertinent compiler analysis is communicated to the hardware without requiring extensive static lookahead, software guarantees, or high instruction overhead. In the subsequent sections, we describe first the hardware engine, then the software hints and analysis necessary for the hardware to balance prefetch coverage and accuracy.

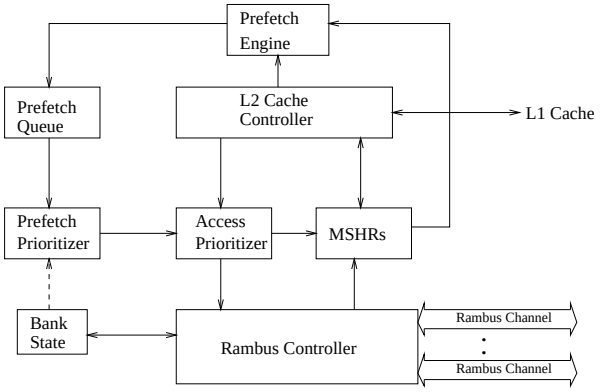

Figure 2: Prefetch Engine Organization

3 Hardware Prefetching Engine

The GRP hardware prefetching engine builds on the scheduled region prefetching design by Lin et al. [28]. We extend the original design with two capabilities. First, we add support for aggressive prefetching of pointer-based data structures. Second, we add the ability to prefetch indirect array references under software control.

3.1 Scheduled Region Prefetching

Scheduled region prefetching (SRP) aggressively exploits spatial locality by attempting to prefetch large (4 KB) memory regions on each L2 cache miss [28]. The two negative effects of aggressive prefetching—memory bus contention and cache pollution—are addressed directly by reducing the priority of prefetches in memory bus request scheduling and in replacement decisions, respectively. Unlike most prefetching schemes, which must maintain high prefetch accuracy to avoid degrading performance, SRP can identify and access prefetch candidates liberally without degrading uniprocessor performance.

Figure 2 shows the memory system with the SRP engine that forms the experimental baseline. The access prioritizer is the central component of the SRP prefetching engine. It forwards requests to the memory controller whenever the controller indicates that the memory channels are idle. The prioritizer forwards prefetch requests only when there are no outstanding demand misses from the L2 cache. Demand misses thus encounter contention only from prefetches the memory controller has already issued, and not from prefetch candidates buffered in the prefetch queue. The miss status holding registers (MSHRs) track all outstanding accesses, regardless of type.

On an L2 cache miss, the prefetching engine allocates a new entry in the prefetch queue representing the aligned memory region containing the accessed block. Each prefetch queue entry contains the base address of the region, a bit vector indicating the prefetch candidate blocks in the region, and an index field which identifies the next block within the region to prefetch. On the first miss to a region, the engine initializes the bit vector to identify the blocks not already present in the L2 cache, and sets the index field to indicate

the next prefetch candidate block after the miss block. It adds these new entries to the head of the queue, giving them priority over older, and thus typically less relevant, entries. The queue is a fixed size (32 in these experiments), and old entries fall off the bottom. On a miss to a region already in the queue, it clears the bit corresponding to the miss block, sets the index field to the next prefetch candidate block after the new miss block, and moves the prefetch bit vector entry to the head of the queue. In this work, we use a base region size of 4 KB and a cache block size of 64 bytes, resulting in a 64-bit vector and a 6-bit index field. Once the controller prefetches all the candidates, it deallocates the entry.

Although the access prioritizer practically eliminates performance loss from useless prefetches due to bandwidth contention, prefetching can still pollute the cache by generating a heavy prefetch stream. We address this issue by placing prefetched data in the lowest priority position of the replacement scheme. The controller puts prefetched data in the LRU position of the pertinent cache set, and moves a block to the MRU position only if it is referenced explicitly by the CPU. As a result, useless prefetches in an n-way associative cache can displace at most one nth of the useful data in the cache. (We use a 4-way set associative cache in our experiments.) The drawback is that the controller occasionally replaces potentially useful prefetches before they are referenced; however, previous work [28] shows this effect to be insignificant. As a final optimization, the queue issues prefetches first to those DRAM banks that already have the needed page open.

Scheduled region prefetching is highly effective at exploiting spatial locality to improve performance [28]. However, it has two shortcomings addressed by GRP. First, SRP does not provide any direct support for non-spatial reference patterns. We add a pure hardware pointer prefetching mechanism to address this issue (see Section 3.2). We also add an indirect array scheme that requires compiler support (see Section 3.3). However, for the SPEC benchmarks, we find that spatial prefetching works as well as pointer schemes—even for pointer-intensive benchmarks—because of the regular layout programmers use and memory allocation patterns for pointer data structures. Second, SRP can produce copious amounts of excess memory traffic. Although this useless traffic does not reduce uniprocessor performance due to SRP's prioritization techniques, it consumes energy, can cause contention from useful prefetches, and may reduce performance in a multiprocessor environment. We thus use compiler hints for spatial and pointer accesses to gain both low bandwidth and high accuracy. We describe the GRP hardware modifications and hints below in Section 3.3, and the compiler analysis itself in Section 4.

3.2 Hardware Prefetching of Pointer-Based Structures

As discussed in Section 2, hardware prefetching for pointer-based structures is challenging. Instead of using complex hardware to recognize pointer traversal patterns or store pointer correlations, the base GRP pointer prefetching scheme greedily generates a prefetch for any fetched value that falls within the ranges of legitimate heap memory addresses. The GRP implementation performs a simple base-and-bounds check using the start and end addresses of the heap. In the Alpha ISA, pointers are aligned 8-byte entities; thus the engine must check only eight values out of each 64-byte cache block.[1]

Once the controller identifies a datum as a possible pointer value, it translates the virtual address to a physical address and forwards the address to the SRP prefetch queue, which allocates a region-style entry for the prefetch. Because these pointer dereferences frequently do not exhibit spatial locality, it sets only two bits in the entry's prefetch bit vector, indicating the block containing the prefetch address and its immediate successor (which prefetches data structures that span two cache blocks). We generalize this mechanism to chase recursive pointers by scanning prefetched lines for addresses and generating additional prefetches.

[1] Cooksey et al. [14] describe a similar but more efficient pointer test using bit masks, and apply it to prefetching in the more challenging IA32 environment.

code in loop	spatial	indirect	pointer	recursive pointer	size
a[i]	√				√
a[b[i]]	√	√			
*p; p+=c	√				
p→f			√		
p = p→next				√	

Table 2: Compiler Hints for Representative References in Loops

3.3 GRP: Incorporating Compiler Prefetch Hints

This section describes the compiler hints used by GRP to improve the precision of L2 spatial and pointer prefetching.

The GRP compiler annotates load instructions with hints predicting whether spatial or pointer-based prefetches will be useful. In this study, the compiler conveys the hints with unused Alpha VAX-format floating point load opcodes. The memory system propagates the load's hint bits through the memory hierarchy with any resulting request. Table 2 presents the five hints and shows typical representative code snippets for each. We summarize the changes to the hardware for each hint below, and then describe the pointers, recursive pointers, and the indirection hardware in more detail.

- A *spatial* hint indicates that a reference is likely to exhibit spatial locality. GRP initiates a spatial prefetch only when the L2 miss is marked spatial.
- A *size* hint combined with a loop upper bound indicates how many cache lines prefetch.
- An *indirect* hint indicates that the program is using an array to index a second array. On an indirect L2 miss, GRP generates sets of prefetches based on the base address and the index values.
- A *pointer* hint indicates that the reference is to a structure that contains one or more other pointers that the program is likely to follow. If the reference is an L2 miss, GRP scans the returned block for pointer values and generates prefetches only for those values.
- A *recursive pointer* hint indicates not only that the reference is to a structure that contains other pointers, but that the program recursively follows these pointers. On a recursive pointer L2 miss, GRP scans the returned data for pointer values, generates prefetches for these addresses, and continues generating prefetches on the subsequent n levels into the recursive data structure. (We use $n = 6$ in our experiments.)

3.3.1 GRP for Pointer and Recursive Pointer References

GRP uses the same mechanism for pointer and recursive pointer hints. However, GRP applies the mechanism only to a pointer hint miss, and GRP applies it repeatedly to the resulting prefetched lines for recursive pointer hints.

We implement GRP for pointer and recursive pointer hints by adding a three-bit counter to both the L2 MSHRs and prefetch queue entries to control pointer and recursive pointer prefetching uniformly. GRP initializes the counter on the L2 miss: for *pointers*, it sets the value to one, and for *recursive pointers*, it sets the value to six. Thus the only difference between a pointer and recursive pointer prefetching is their initial counter value.

When GRP fetches a pointer hinted missing line, it starts the pointer prefetching engine on the returned line. The engine checks the counter. If it is zero, it stops queuing prefetches. Otherwise, it decrements the counter, and queues prefetches for pointers in the returned line. We prefetch two cache blocks for each pointer based on our statistics that the typical structure size in SPEC benchmarks is less than 64 bytes (one L2 cache block in our configuration). Two blocks are sufficient to cover structure alignment. The engine thus terminates after one level for pointers and six levels for recursive prefetching.[2]

3.3.2 GRP for Variable-Size Region Prefetching

GRP by default prefetches the same fixed region size as SRP. If the spatial reuse of a reference does not span the default region size, prefetching wastes bandwidth. We enhanced GRP to allow the compiler to control region sizes for references in singly nested loops. The compiler computes the loop upper bound for the primary induction variable and conveys the bound to the hardware using a special instruction. The compiler encodes a coefficient for each spatial reference in the loop. On a miss, the prefetch engine uses this bound and the coefficient to calculate the region size as *loop bound* $<<$ *coefficient value*.

3.3.3 GRP for Indirect Array References

Two of the benchmarks from the SPEC2000 suite (*vpr* and *bzip2*) incur a significant number of misses due to indirect array references of the form `a[b[i]]`. References to `a` are not amenable to spatial prefetching unless the `b[i]` values are clustered, which cannot be determined statically. Pointer prefetching for these references is ineffective since the desired addresses are computed, not contained in the memory as pointers. A specialized extension to GRP targets these patterns. A single *indirect prefetch* instruction conveys both a base address (`&a[0]`), an element size (`sizeof(a[0])`), and an index array address (`&b[i]`) to the prefetching engine. The prefetch engine reads the cache block containing `&b[i]` and, for each word in the block, generates a prefetch address by adding the scaled value to `&a[0]`. GRP then forwards these addresses to the prefetch queue, as in the pointer prefetching scheme. Currently, we assume the index array element size (`sizeof(b[0])`) is 4, which is typical on most systems, although the element size could be included in the instruction if necessary.

This scheme is distinct from the mechanisms proposed in this paper because the information is encoded as a separate instruction, not a hint on an existing load. Although the introduction of an explicit prefetch instruction adds overhead, the number of such instructions is small, and each one generates up to 16 prefetches (one for each index within a cache block of the indirection array). An alternate implementation could use a single instruction prior to a loop nest to set the base address, and an additional hint bit on the `b[i]` loads to trigger the indirect prefetches. This approach would reduce execution overhead at the cost of limiting an application to prefetching one single indirection array concurrently per base address/indirect hint pair.

[2]For *mcf*, we terminate recursion after three levels to make simulation tractable.

4 Compiler Analysis Framework

This section describes the analyses for the five classes of hints (*spatial*, *size*, *indirect*, *pointer*, *recursive pointer*) that guide the L2 prefetching engine. We implement these analyses in the Scale compiler and use it to generate these hints automatically for both C and Fortran codes.

4.1 Spatial Locality Analysis for Arrays

In GRP, the compiler predicts which misses truly have spatial locality, examining arrays in Fortran or C, and spatial pointer accesses to structures in C. The compiler uses locality analysis to mark references with the spatial hint annotation, and the compiler back-end augments the special load instruction with a spatial hint. The prefetch engine then only prefetches misses with marked spatial references and does not prefetch misses without spatial marks. We describe our array analysis and then spatial pointer analysis.

We augment prior work that statically detects spatial locality by extending dependence testing [32, 44]. Dependence testing first finds induction variables and then detects when the spatial dimension (the row in C, the column in Fortran) is accessed as a function of the index variable, and whether it is the inner or outer nesting level. The dependence testing detects locality only for *affine* subscription expressions, i.e., linear functions of loop induction variables. Our approach marks references with either inner or outer loop spatial locality. The typical array reference with spatial locality is accessed in its spatial dimension in an innermost loop. For example, we mark a(i,j) in Figure 3, assuming column-major Fortran storage. The compiler also marks arrays with spatial locality that crosses larger distances within a deep nest or between two nests (*inter-nest reuse*). We use the level 2 cache size as our upper bound on the distance of the spatial reuse we mark, assuming that the level-2 cache has sufficient set associativity to avoid conflict misses and exploit the reuse.

If the compiler determines the loop bounds and step sizes, it can compute the reuse distances accurately at compile time. For arrays with spatial intra- and inter-nest locality, it computes the reuse distances. It marks all array references with spatial locality with a known distance less than the level 2 cache size. When the compiler does not know the reuse distances statically due to symbolic loop bounds and uncertain executions paths, it estimates the reuse distance based on the nesting level of the loop. The compiler is conservative when reuse distance is unknown; we mark a reference as spatial only if its spatial reuse is in the innermost enclosing loop.

The above analysis works well for Fortran arrays and heap arrays in C if the array elements are referenced as subscript expressions. We handle heap arrays in C using the same analysis. In Figure 4, *buf* is a heap array with type *T***. In addition to detecting the obvious spatial reuse of $buf[i][j]$ when j is an loop induction variable, the compiler is able to find the spatial reuse of $buf[i][a*j+b]$ when a and b are constants.

4.2 Spatial Locality Analysis for Pointer Dereferences

To prefetch pointer references that show spatial locality, as illustrated in Figure 5, the compiler performs loop induction variable

```
integer a[N][M], B[N]
do j=1, m
 do i=1, n
   ... a(i,j)...
   ... c(b(i),j))...
```

Figure 3: Fortran Array

```
T ** buf;
...
buf = malloc(...);
...
buf[i] = malloc(...);
...
for (i=0; i<m; i++)
 for (j=0; j<n; j++)
      ... buf[i][j] ...
```

Figure 4: C Heap Array

```
T *p, *s;
...
for (; p < s; p += c) {
 /* if T is a primary type */
...*p...;
/* if T is a structure */
... p->f ...;
}
```

Figure 5: C Induction Pointer

```
struct t {
  T      f;
  struct t * next;
}
struct t *a;

while(...) {
  ... a->f ...;
  a = a->next;
  ...
}
```

Figure 6: C Recursive Pointer

```
generate_spatial_hints()
{
  /* recognize induction variables including pointers */
  induction_variable_recognition();
  /* perform dependence testing */
  dependence_testing();

  for (each loop) {
    /* generate basic spatial hints */
    for (each memory reference r in the loop) {
      if (r is an array reference) {
        if (r has spatial reuse in the enclosing innermost loop)
          mark r spatial;
        else {
          compute the reuse distance for r if applicable;
          if (reuse distance < the level 2 cache size)
            mark r spatial;
      } }
      if (r is an loop induction pointer)
        mark r spatial;
    }

    /* propagate spatial hints for loop induction pointers */
    do {
      for (each memory reference r) {
        if (r is a loop induction pointer)
          mark *r as spatial;
        else if (r is a->f && a is marked as spatial) {
          mark a->f as spatial;
      }
    } while (no new hints generated);
  }
}
```

Figure 7: Algorithm generating spatial hints

recognition on pointers that are repeatedly incremented by a constant. The type *T* in Figure 4 and Figure 5 does not have to be a primary type. We treat pointer *p* as a special integer, and insert spatial hints for $*p$ or $p \rightarrow f$, if constant *c* is small. This paper's analysis on L2 cache misses shows almost all spatial reuses in C code are covered by regular spatially local array references along with the cases in Figure 4 and Figure 5.

Figure 7 summarizes the algorithm used for generating spatial hints for both arrays and spatial pointer accesses. The first part of the algorithm inserts the spatial hints for arrays and loop induction pointers, and the second part propagates spatial hints to the uses of loop induction pointers. This algorithm is intra-procedural and flow insensitive, and it marks only references enclosed in loops.

4.3 Indirect Analysis

The compiler also detects and marks indirect array accesses, such as $c(b(i), j)$ in Figure 3. In particular, it looks for the access pattern in the form of $a(s * b(i) + e)$ where s and e are constants, and i is a loop induction variable. Dependence testing detects the spatial reuse on $b(i)$ in the standard way. We add a simple analysis that detects when a sequentially accessed array is used as an index into another array (c in this example), and generates an indirect prefetch instruction using the address of $b(i)$ and the base address of array

```
generate_pointer_hints()
{
  for (each field access) {
    if (a pointer field from the same structure
        is accessed in the same loop)
      mark the field access as pointer;
    if (the field access updates a recurrent pointer)
      mark the field access as recursive pointer;
  }

  for (each array reference marked as spatial) {
    if (the reference points to a heap array)
      mark the reference as pointer;
  }
}
```

Figure 8: Algorithm generating pointer and recursive pointer hints

c, as described in Section 3.3.

4.4 Variable-Size Region Analysis

The compiler detects and marks array references within singly nested loops for variable-size region prefetching. For an array access with a pattern of $a(b * i + c)$ and an array element size of e, the compiler encodes $b * e$ into a three-bit value x such that $x < 7$ and 2^x is closest to $b * e$. We reserve the encoding value 7 for fixed-size region prefetching. The compiler marks the upper bound of the loop induction variable i. The two hints are used to control the region size as described in Section 3.3.

4.5 Pointer and Recursive Pointer Analysis

As with spatial locality, the compiler can improve the accuracy of hardware-based pointer prefetching by restricting it to misses on a load to a field from a structure that contains a pointer or recursive field. We mark a field reference as *pointer* if a pointer field from the same structure is accessed in the same loop. We mark a pointer update to be *recursive* if it updates itself in a loop with an object of the same data type. For example, in Figure 6, a is updated with its *next* field which points to a structure of the same type *struct t*. This idiom analysis simply identifies pointer updates in a loop that use a field with the same type and marks them as recursive pointer updates.

We mark pointer accesses with the spatial hint for references to arrays of pointers. For example, Figure 4 shows an array reference *buf*$[i]$, whose access pattern results in a spatial hint from the compiler. Furthermore, each *buf*$[i]$ points to a heap array, so the compiler marks it with the pointer hint as well. GRP will then use the address to prefetch the pointed-to array.

The algorithm to generate pointer and recursive pointer reference hints is shown in Figure 8. It is complementary to the spatial marking algorithm for pointers shown in Figure 7.

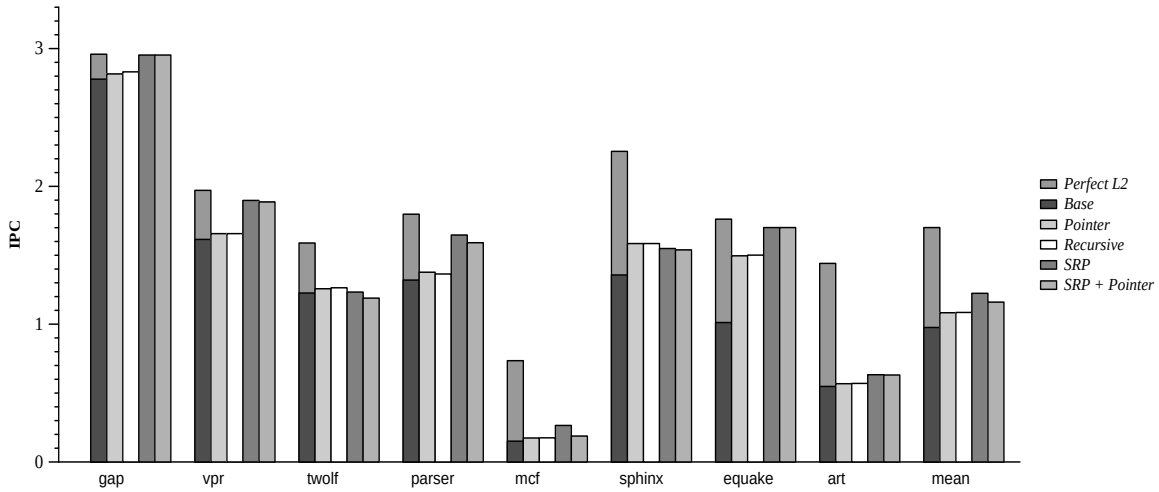

Figure 9: Performance gains from pointer prefetching

Benchmark	mem insts	spatial	pointer	recursive	ratio(%)	indirect
164.gzip	1873	433	268	0	37.1	9
168.wupwise	507	152	0	0	30.0	0
171.swim	250	115	0	0	46.0	0
172.mgrid	314	232	0	0	73.9	3
173.applu	1491	858	0	0	57.5	0
175.vpr	4230	1001	682	74	33.8	84
177.mesa	26777	4532	4419	76	32.8	9
179.art	1016	732	278	0	77.6	0
181.mcf	845	168	287	201	60.8	0
183.equake	1679	597	473	0	51.3	7
186.crafty	11702	1994	736	0	21.6	5
188.ammp	6271	1043	1158	0	33.2	5
197.parser	4090	915	932	1263	70.2	2
254.gap	29781	5102	11243	0	52.6	36
256.bzip2	698	279	59	0	48.3	14
300.twolf	12397	2080	2577	1398	45.1	38
301.apsi	3225	1001	0	0	31.0	0
sphinx	6335	2211	1129	364	46.8	106

Table 3: Number of compiler hints for each benchmark

5 Experimental Evaluation

In this section, we compare the performance benefits of SRP, GRP, and unified stride prefetching for the SPEC CPU2000 benchmarks, and one additional benchmark. We demonstrate that GRP provides a compelling balance between higher performance and increased memory traffic among the three prefetching techniques. We demonstrate the effectiveness of the compiler generated size information, and the sensitivity of our results to the compiler's heuristic for computing the useful distance of spatial locality. We conclude with a discussion of the characteristics of the remaining benchmarks for which GRP does not eliminate main memory accesses as a significant loss of performance.

5.1 Experimental Methodology

The Scale compiler infrastructure inserts the prefetch hints [3]. It performs a number of scalar optimizations such as constant propagation and common subexpression elimination. It compiles C and Fortran 77 code to Alpha assembly code, with the memory hints annotated as comments. We then post-process the annotated assembly code to generate binaries containing compiler-hinted instructions.

We simulate program binaries on a version of sim-outorder [4] with scheduled region prefetching (SRP) [28] added to the simulator. We added the GRP hardware pointer prefetching mechanisms, and modified the simulator to accept compiler hints and schedule prefetches accordingly if the binaries contain the hints. We use the Alpha-ISA and configure the simulator as a 1.6 GHZ, 4-way issue, 64-entry RUU (reorder buffer), out-of-order core with 64K 2-way split level one caches and a unified 4-way 1MB level 2 cache. This cache hierarchy is combined with an effective 800-Mhz, 4-channel Rambus memory system. The L1 and L2 latencies are 3 and 12 cycles [3], respectively. Each cache contains 8 MSHRs. For SRP, the prefetching queue size is 32 and uses LIFO scheduling. The stride predictor [38] uses a 4-way history table with 1K entries. There are 8 entries in each of 8 streaming buffers sharing the history table. Finally, we use the SimPoint [37] tool set to select a representative starting point beyond the program's initialization phase. We simulate for 200M instructions from that point.

We use the 17 SPEC CPU2000 C and Fortran benchmarks that the Scale infrastructure is able to compile correctly, plus *Sphinx*, a speech recognition application [27]. Table 3 lists these benchmarks, along with statistics on memory instructions and the number and type of compiler hints generated. The second column contains the total number of static memory reference instructions. Columns 3 to 5 show the number of instructions the compiler marks as *spatial*, *pointer*, and *recursive*. (Note that the compiler can mark an instruction both *spatial* and *pointer*.) Column 6 lists the fraction of static memory operations with hints, and Column 7 shows the static number of indirect prefetch instructions. We do not present the results for *crafty* in subsequent results because its L2 miss rate is negligible (0.4%).

5.2 Comparison of Stride Prefetching, SRP, and GRP

In this section, we first present the effects of both hardware pointer and recursive pointer prefetching. We show that explicit pointer prefetching is generally subsumed by aggressive spatial prefetching (SRP or GRP). We then compare stride prefetching with SRP and GRP. GRP uses all the compiler analysis including variable region sizes. The end of this section compares variable and fixed region sizes, and finds variable sizes decrease bandwidth requirements for 3 programs.

We apply pointer prefetching alone to all benchmarks, which unsurprisingly has little effect on the Fortran benchmarks. Eight C benchmarks show a significant performance improvement, notably a 48.3% boost for *equake*, a 15.9% increase for *mcf*, and a 14.4%

[3] We mistakenly put 1 and 16 in our published version. But they do not affect our conclusions.

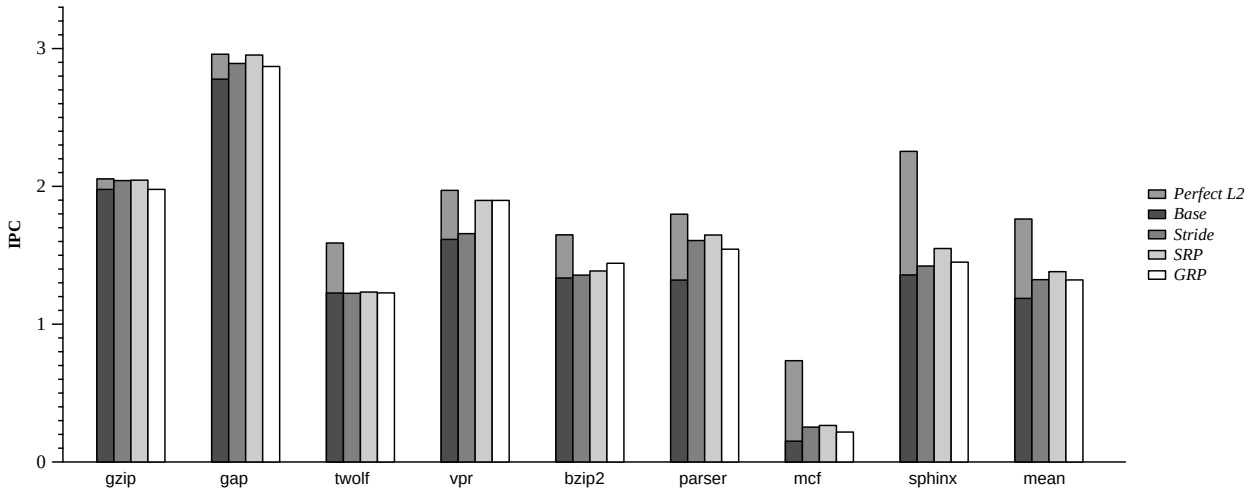

Figure 10: Performance gains from region prefetching and stride prefetching for integer benchmarks

improvement for *sphinx* as shown in Figure 9. For *equake*, the performance gain is not from the pointer structure traversal as expected. It stems instead from prefetching arrays of pointers from the heap arrays. Similarly, in *mcf*, the performance gain comes from a loop which sequentially resets a field in each object in a heap array. Pointer prefetching happens to prefetch the objects accessed later. Pointer prefetching outperforms SRP only for *twolf* and *sphinx*, by 2%. In all other cases, SRP performs much better than pointer or recursive prefetching. Applying SRP and pointer prefetching together gives little benefit and sometimes degrades the performance due to much higher bandwidth consumption, which can result in fewer successful prefetches. GRP with pointer and recursive hints shows performance gains similar to SRP for the seven benchmarks, but with lower average memory traffic .

Figure 10 and Figure 11 show the performance of SRP, GRP, and stride prefetching for integer and floating point benchmarks respectively. In most cases and on average, SRP and GRP both perform better than stride prefetching. For 10 benchmarks, SRP improves performance to within 10% of a perfect L2. For *swim*, GRP performs over 10% better than SRP due to its lower traffic. Due to the indirect prefetching, GRP is 4% faster than SRP for *bzip2*. It also outperforms SRP for *art* and *ammp*. For *gzip*, *mcf*, *parser*, and *gap*, the IPC of GRP is at least 2% less than that of SRP. A typical reason is that the compiler misses locality outside of loops.

Although we detect indirect references in 11 benchmarks, indirect prefetching shows significant speedups for only *vpr* and *bzip2*. For *vpr*, the indirect references show high spatial locality. SRP thus performs as well as GRP, but with 50% additional traffic. *bzip2* is one of the benchmarks where SRP does not perform well. With indirect prefetching, the gap from a perfect L2 is reduced to 12.5% from 15.9%, with only 15% of the memory traffic of SRP.

In terms of both performance and memory traffic, GRP using a variable region size (GRP/Var) and a fixed region size (GRP/Fix) only differ in three benchmarks, *mesa*, *bzip2*, and *sphinx*. Table 4 shows that for *mesa* and *bzip2*, both strategies deliver roughly the same performance while GRP/Var results in much less traffic than GRP/Fix, as we discuss in Section 5.3. For *sphinx*, GRP/Var has 5.8% lower performance than GRP/Fix, but benefits from an 82% traffic reduction. The compiler cannot guarantee that there is spatial locality, so it chooses small prefetch regions, and misses some opportunities.

	GRP Traffic		Region Size Distribution			
	Var	Fix	2	4	8	64
mesa	1.11	6.55	90.3	9.5	0.1	0.1
bzip2	1.47	4.97	76.8	22.4	0.0	0.8
sphinx	2.09	11.66	82.9	1.0	16.1	0.0

Table 4: GRP/Var versus GRP/Fix

5.3 Prefetching Accuracy, Coverage, and Memory Traffic

Although SRP and GRP provide comparable performance, SRP consumes much more bandwidth than does GRP. Figure 12 shows the normalized memory traffic for the three prefetch schemes. SRP increases memory traffic from 2% to a *factor of* 25.5 times over no prefetching. GRP generates a mean of only 23.0% additional traffic compared to no prefetching, versus an SRP increase of 180%. GRP eliminates over 20% of the total memory traffic for ten of the seventeen benchmarks compared to SRP, and over 50% for six benchmarks. The traffic for stride prefetching is 11% less than GRP, but stride prefetching only achieves 69% of the performance improvement that GRP does.

Compared to GRP/Fix, GRP (GRP/Var) cuts memory traffic significantly for three benchmarks while showing the same traffic for the others. Table 4 lists the three benchmarks and their traffic increase compared to no prefetching in columns one through three. The subsequent four columns show the distribution of prefetching requests by the region sizes (no regions of 16 or 32 blocks are produced). We observe that GRP/Var only prefetches one additional block (region size = 2) in most cases due to the poor spatial locality of these references.

Table 5 shows both prefetching accuracies and coverage for the three prefetching techniques that we implemented. We use the percentage reduction in L2 misses as a metric for coverage. On average, SRP provides the best coverage and the worst accuracy. Stride prefetching trades the lowest coverage with the highest accuracy. GRP obtains the best of both worlds: an accuracy that is closer to stride prefetching, but coverage closer to that of SRP.

Since the normalized traffic in Figure 12 does not reflect the absolute bandwidth consumption of each benchmark, we also list the actual memory traffic, in bytes, of each benchmark in Table 5. On average, SRP consumes 99.8% more memory bandwidth over the no-prefetching system. GRP and stride prefetching produce a

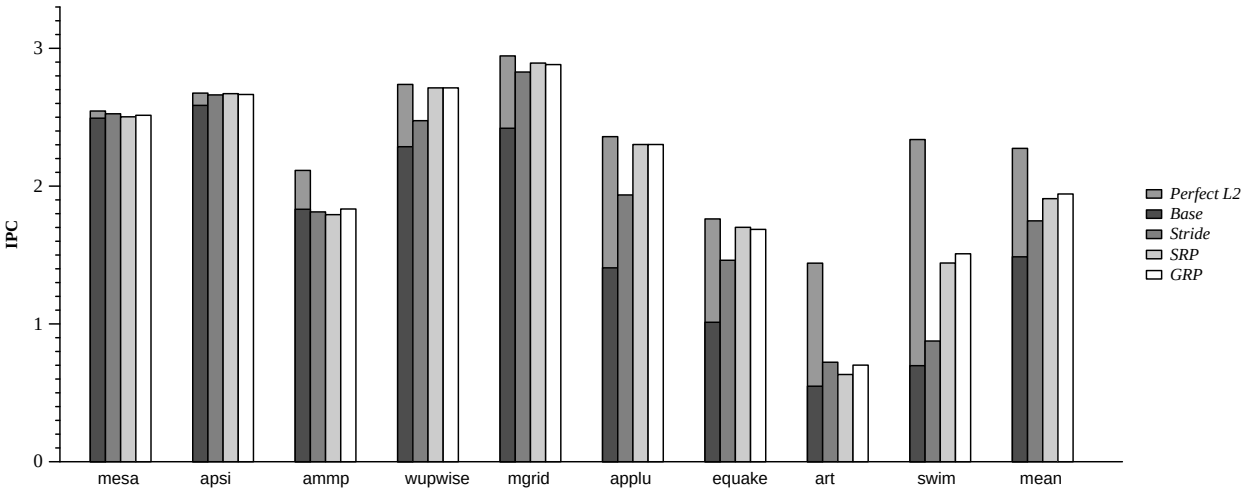

Figure 11: Performance gains from region prefetching and stride prefetching for floating-point benchmarks

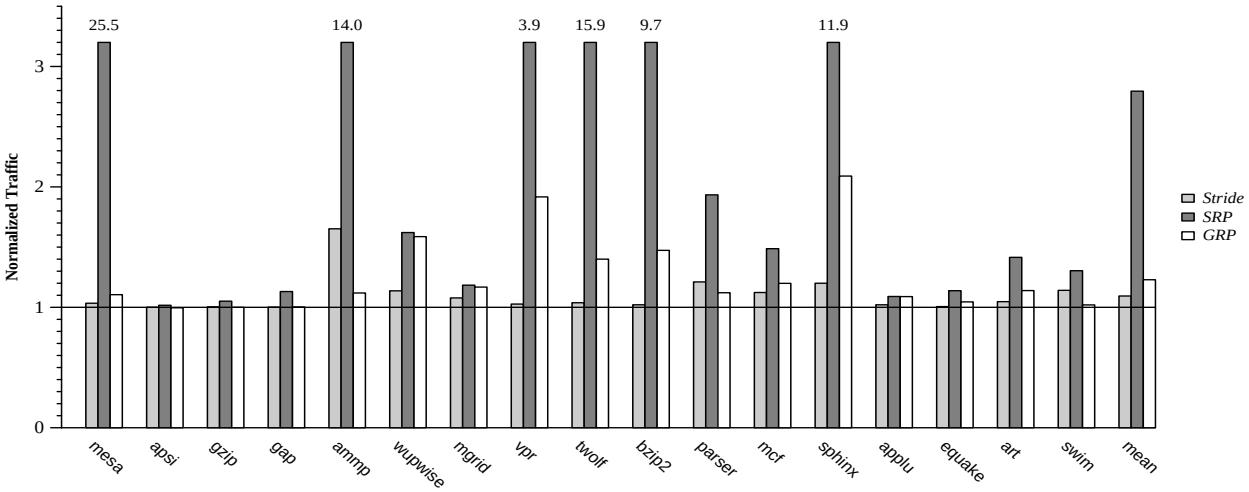

Figure 12: Normalized traffic

Benchmark	Base		Stride			SRP			GRP		
	Miss Rate	Traffic	Coverage	Accuracy	Traffic	Coverage	Accuracy	Traffic	Coverage	Accuracy	Traffic
mesa	9.3	51k	60.9	93.2	53K	29.3	0.8	1305K	43.5	70.1	56K
apsi	25.0	85K	79.2	99.8	85K	96.4	95.8	86K	88.8	97.6	84K
gzip	25.3	182K	65.2	99.8	183K	76.3	94.4	192K	0.0	91.2	182K
gap	46.8	179K	66.7	99.6	179K	97.6	86.3	202K	52.8	99.3	179K
ammp	15.3	594K	-7.8	23.1	982K	-7.8	0.9	8340K	0.7	27.5	665K
wupwise	73.1	486K	42.5	75.4	553K	96.3	60.2	788K	96.2	61.6	772K
mgrid	43.9	504K	77.9	89.9	544K	87.5	80.7	597K	85.6	81.7	589K
vpr	40.2	730K	15.9	85.5	749K	86.3	27.6	2820K	76.4	49.4	1399K
twolf	12.6	1125K	0.0	27.3	1167K	15.9	4.2	17878K	3.2	28.7	1575K
bzip2	22.4	1163K	8.4	85.7	1186K	27.2	5.3	11255K	37.1	51.6	1713K
parser	33.4	1450K	67.4	75.0	1756K	77.5	44.7	2804K	56.0	82.5	1625K
mcf	61.6	43901K	51.0	80.5	49284K	24.7	53.9	65263K	5.4	51.1	52656K
sphinx	65.9	1208K	12.6	27.3	1449K	42.8	4.7	14429K	21.7	20	2521K
applu	58.0	2578K	62.6	95.7	2631K	96.9	89.0	2810K	96.9	89.2	2806K
equake	59.8	3628K	75.6	99.2	3649K	96.3	86.9	4127K	95.2	95.3	3790K
art	44.4	20229K	17.3	99.7	21189K	8.6	40.6	28632K	20.9	78.0	23031K
swim	57.8	7861K	34.6	70.8	8966K	67.3	65.2	10249K	68.2	96.5	8021K
average	40.9	5057K	42.9	78.1	5565K	59.9	49.5	10105K	49.9	68.9	5981K

Table 5: Prefetching accuracy, coverage and memory traffic

Benchmark	GRP Performance Gap (%)	L2 Miss Causes	Ratio (%)
171.swim	38.32	transpose array access	92.08
179.art	56.07	bandwidth	24.26
		transpose heap array access	35.92
181.mcf	63.94	tree traversal	60.70
188.ammp	15.18	linked list traversal	88.64
256.bzip2	15.89	indirect array reference	49.68
300.twolf	22.40	linked list and random pointers	35.37
sphinx	31.28	hash table lookup	28.79

Table 6: Level 2 miss characteristics

18.3% and 10.1% increase in memory requests, respectively.

5.4 Compiler Sensitivity

We explored the sensitivity of our results to the compiler policy by implementing both more and less aggressive variants of the scheme described in Section 4. The more aggressive policy marks a reference as *spatial* even its reuse distance is greater than the L2 cache size. The more conservative scheme marks a reference as *spatial* only when its reuse sits in the innermost loop. Compared to our default GRP policy, the aggressive policy degrades performance by 2% overall and increases traffic by an additional 5%. The conservative scheme shows little effect on memory traffic compared with GRP, but causes moderate performance losses across four benchmarks: *applu*, *art*, *equake*, and *apsi*, and reduces performance by an average of 5% across the benchmark suite.

5.5 Remaining L2 Misses

Seven of the benchmarks show a gap of greater than 15% between SRP and a perfect L2. We list them in Table 6 with a description of the key causes of the misses, obtained by analyzing the source.

With its more accurate prefetching, coupled with indirect accesses and pointer prefetching, GRP is able to bring *bzip2* and *ammp* under 15%. *Swim* has a low IPC due to pathological array conflicts. We can prevent that benchmark from being memory-bound by manually applying loop distribution and loop permutation [8]. We observe that *art* is bandwidth bound. While GRP reduces traffic and increases performance over SRP by 10.7%, the performance gap is still large. Larger caches and wider channels improve *art* appreciably. For *sphinx*, the hash table lookup usually touches only a small number of adjacent hash slots in a short loop. Prefetches occur simply too late to tolerate the latencies. Finally, *mcf* and *twolf* contain heavy traversals of short linked lists and tree data structures, making them poor matches for the GRP pointer prefetching or spatially-based schemes.

6 Conclusions and Future Work

Purely compiler-based prefetching techniques have difficulty managing the large latencies of modern main memories. Previous work shows that aggressive hardware prefetching addresses this issue effectively for applications with spatial locality, at the cost of potentially significant increases in memory bandwidth. As the number of processors per chip increases, this bandwidth will become increasingly precious.

This paper shows that a cooperative approach between compiler-based analysis and hardware-based aggressive prefetching provides benefits comparable to aggressive hardware prefetching with much lower traffic. Compiler techniques identify accesses that clearly possess spatial locality. Rather than use this information to attempt to schedule software prefetches—with the resulting complications of providing timely prefetches while minimizing instruction overhead—our system simply passes this access-pattern information to a hardware prefetching engine. The engine then generates prefetches at the L2 cache with low overhead. Compared to pure hardware prefetching, the compiler analysis saves bandwidth by avoiding useless prefetches to addresses with little locality.

We also extend the hardware prefetching engine to address pointer-based applications by aggressively prefetching any datum that appears to be a pointer. As with spatial locality, we see significant traffic benefits from having the compiler indicate pointer and recursive-pointer loads. However, for the SPEC2000 benchmarks, the aggressive spatial locality analysis subsumes the pointer prefetches for most benchmarks, due to spatially local layouts of pointer-connected objects. Even *Sphinx*, which we chose for its sparse irregular pointer behavior, benefits very little from pointer prefetching. It still remains to be seen whether this phenomenon will dominate the benchmarks that other researchers have used to show the importance of greedy pointer hardware prefetching [14].

With solely the spatial and indirect hints, the GRP compiler/hardware prefetch framework eliminates most L2-related stalls across the SPEC2000 suite, with comparatively modest increases in traffic. The remaining three benchmarks that are limited by L2 memory system performance are either bandwidth bound (*art*) or contain many irregular linked-lists and/or tree traversals (*mcf*, *twolf*) where memory-side prefetching may help. For the rest of the SPEC2000 suite, however, the GRP approach eliminates physical memory accesses as a performance bottleneck while making significantly more efficient use of the system bandwidth than similarly aggressive prefetch engines.

Acknowledgments

This work is supported by NSF ITR grant CCR-0085792, NSF grants CCR-0105503, ACI-0203895, and CCR-9985109, NSF Research Instrumentation grant EIA-9985991, the Defense Advanced Research Projects Agency under contracts F30602-98-1-0101 and F33615-01-C-1892, grants from the Intel Research Council, an IBM University Partnership award and other gifts from IBM, and two Alfred P. Sloan Research Fellowships.

References

[1] T. Alexander and G. Kedem. Distributed predictive cache design for high performance memory system. In *Second International Symposium on High Performance Computer Architecture*, Feb. 1996.

[2] M. Annavaram, J. M. Patel, and E. S. Davidson. Data prefetching by dependence graph precomputation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 52–61, 2001.

[3] Architecture and Language Implementation Group, University of Massachusetts, Amherst. Scale compiler infrastructure. In *http://ali-www.cs.umass.edu/Scale*.

[4] D. Burger and T. M. Austin. The simplescalar tool set version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin, June 1997.

[5] B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compiler Techniques*, pages 280–291, Barcelona, Spain, Sept. 2001.

[6] B. Cahoon and K. S. McKinley. Simple and effective array prefetching for Java. In *ACM Java Grande*, pages 86–95, Seattle, WA, Nov. 2002.

[7] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, Santa Clara, CA, Apr. 1991.

[8] S. Carr, K. S. McKinley, and C. Tseng. Compiler optimizations for improving data locality. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 252–262, San Jose, CA, Oct. 1994.

[9] M. Charney and A. Reeves. Generalized correlation-based hardware prefetching. Technical Report EE_CEG_95-1, Cornell University, Feb. 1995.

[10] T. Chen and J. Baer. Reducing memory latency via non-blocking and prefetching caches. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 51–61, Boston, MA, Oct. 1992.

[11] T. Chen and J. Baer. Effective hardware based data prefetching. *IEEE Transactions on Computers*, 44(5):609–623, May 1995.

[12] T.-F. Chen. An effective programmable prefetch engine for high-performance processors. In *Proceedings of the 29th International Symposium on Microarchitecture*, Ann Arbor, Michigan, Nov. 1995.

[13] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *Proceedings of the 28th International Symposium on Computer Architecture*, pages 14–25, June 2001.

[14] R. Cooksey, S. Jordan, and D. Grunwald. A stateless, content-directed data prefetching mechanism. In *Proceedings of the Tenth Annual International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 279–290, San Jose, CA, October 2002.

[15] F. Dahlgren, M. Dubois, and P. Stenstrom. Fixed and adaptive sequential prefetching in shared-memory multiprocessors. In *Proceedings of the 1993 International Conference on Parallel Processing*, pages 56–63, St Charles, IL, 1993.

[16] F. Dahlgren and P. Stenstrom. Effectiveness of hardware-based stride and sequential prefetching in shared-memory multiprocessors. In *First International Symposium on High Performance Computer Architecture*, pages 68–77, Raleigh, NC, Jan. 1995.

[17] S. Ghosh, M. Martonosi, and S. Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 228–239, San Jose, CA, Oct. 1998.

[18] E. H. Gornish and A. V. Veidenbaum. An integrated hardware/software scheme for shared-memory multiprocessors. In *Proceedings of the 1994 International Conference on Parallel Processing*, pages 281–284, St Charles, IL, 1994.

[19] C. J. Hughes and S. Adve. Memory-side prefetching for linked data structures. Technical Report UIUCDCS-R-2001-2221, University of Illinios, Urbana Champagne, May 2001.

[20] J. Huh, D. Burger, and S. W. Keckler. Exploring the design space of future CMPs. In *Proceedings of the 2001 International Conference on Parallel A rchitectures and Compilation Techniques*, pages 199–210, Sep 2001.

[21] D. Joseph and D. Grunwald. Prefetching using Markov predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 252–263, 1997.

[22] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 364–373, Seattle, WA, June 1990.

[23] M. Karlsson, F. Dahlgren, and P. Sternstrom. A prefetching technique for irregular accesses to linked data structures. In *Sixth International Symposium on High Performance Computer Architecture*, page 206, Toulouse, France, Jan. 2000.

[24] D. Kim and D. Yeung. Design and evaluation of compiler algorithms for pre-execution. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 159–170, San Jose, CA, Oct. 2002.

[25] A. C. Klaiber and H. M. Levy. An architecture for software-controlled data prefetching. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 43–53, Toronto, Canada, May 1991.

[26] A.-C. Lai, C. Fide, and B. Falsafi. Dead-block prediction and dead-block correlating prefetchers. In *Proceedings of the 28th International Symposium on Computer Architecture*, July 2001.

[27] K.-F. Lee, H.-W. Hon, and R. Reddy. An overview of the SPHINX speech recoginition system. In *IEEE Transactions on Acoustics, Speech and Signal Precessing*, volume 38(1), pages 35–44, 1990.

[28] W.-F. Lin, S. K. Reinhardt, and D. Burger. Reducing DRAM latencies with an integrated memory hierarchy design. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, pages 301–312, Jan 2001.

[29] M. H. Lipasti, W. J. Schmidt, S. R. Kunkel, and R. R. Roediger. SPAID: Software prefetching in pointer- and call-intensive environments. In *Proceedings of the 28th Annual IEEE/ACM International Symposium on Microachitecture*, pages 231–236, Nov. 1995.

[30] 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, Oct. 1996.

[31] C.-K. Luk. Tolerating memory latency through software-controlled pre-execution on simultaneous multithreading processors. In *Proceedings of the 28th International Symposium on Computer Architecture*, pages 40–51, June 2001.

[32] K. S. McKinley, S. Carr, and C. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[33] T. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, Boston, MA, Oct. 1992.

[34] S. Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of the 21th International Symposium on Computer Architecture*, pages 24–33, Chicago, IL, Apr. 1994.

[35] A. Roth, A. Moshovos, and G. S. Sohi. Dependence based prefetching for linked data structures. In *Proceeding of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 115–126, 1998.

[36] A. Roth and G. Sohi. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the 26th International Symposium on Computer Architecture*, pages 111–121, Atlanta, GA, May 1999.

[37] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, Barcelona, Spain, Sept. 2001.

[38] T. Sherwood, S. Sair, and B. Calder. Predictor-directed stream buffers. In *Proceedings of the 33rd International Symposium on Microarchitecture*, pages 42–53, Monterey, California, Dec. 2000.

[39] J. Skeppstedt and M. Dubois. Hybrid compiler/hardware prefetching for multiprocessors using low-overhead cache miss traps. In *Proceedings of the 1997 International Conference on Parallel Processing*, pages 298–307, Bloomington, IL, 1997.

[40] A. J. Smith. Cache memories. *Computing Surveys*, 14(3):473–530, Sept. 1982.

[41] Y. Solihin, J. Lee, and J. Torrellas. Using a user-level memory thread for correlation prefetching. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 171–182, May 2002.

[42] S. T. Srinivasan and A. R. Lebeck. Load latency tolerance in dynamically scheduled processors. *Journal of Instruction Level Parallelism*, 1:1–24, 1999.

[43] S. P. Vanderwiel and D. J. Lijia. A compiler-assisted data prefetch controller. In *Proceedings of International Conference on Computer Design*, Austin, TX, 1999.

[44] M. E. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, Toronto, Canada, June 1991.

[45] Y. Wu. Efficient discovery of regular stride patterns in irregular porgrams and its use in compiler prefetching. In *Proceedings of the SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 210–221, Berlin, Germany, June 2002.

[46] C.-L. Yang and A. R. Lebeck. Push vs. pull: Data movement for linked data structures. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, pages 176–186, May 2000.

[47] Z. Zhang and T. Torrellas. Speeding up irregular applications in shared memory multiprocessors: Memory binding and group prefetching. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 1–19, Santa Margherita Ligure, Italy, June 1995.