Guided Region Prefetching: A Cooperative Hardware/Software Approach

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

IBM Technical Contacts: John Keaty, Rob Bell (IBM Server Group) and Mootaz Elnozahy (ARL)

[†] Department of Computer Science
University of Massachusetts, Amherst

[§] Department of Computer Science
University of Texas, Austin

[‡] Department of Electrical Engineering and Computer Science
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 tolerate these latencies. Software prefetching typically provides better prefetch accuracy than hardware, but is limited by per-prefetch overheads and the compiler's limited prefetch scope. Hardware prefetching can be much more effective at hiding level-two cache miss latencies, but generates many useless prefetches and 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, a 23% gain over no prefetching, but SRP incurs 153% extra memory traffic—more than doubling bandwidth requirements. GRP achieves the same performance but with one fifth the traffic increase of SRP, a 31% 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.

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 [40]. However, the hundreds of cycles that result from physical memory accesses cannot be tolerated, and as such cause 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 missed in the level-two cache [26].

There have been a large number of prefetching schemes proposed using both software and hardware techniques. 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 [26]. Hardware-only schemes prefetch both spatial regions [8, 9, 22, 32, 36] and pointer chains [12, 21, 33]. While these schemes can hide much of the main memory access time, they can also consume substantial amounts of memory bandwidth. This additional memory traffic need not degrade performance for a uniprocessor, but it increases power consumption, and will likely degrade performance in a multiprocessor environment. Off-chip bandwidth will be the dominant limiter of scalability for future chip multiprocessors (CMPs) [20], so prefetch schemes that consume bandwidth inefficiently will not be practical. Some schemes throttle prefetching when the accuracy drops below a threshold, but they miss opportunities for issuing prefetches [15].

	Speedup	traffic increase	Performance gap from perfect L2
No prefetching	1	1	33.37
Stride prefetching	1.153	1.09	23.16
SRP	1.231	2.53	17.94
GRP	1.226	1.31	18.32

Table 1: Summary of prefetching performance gains and traffic increases

In this paper, we propose a cooperative hardware-software prefetch framework called Guided Region Prefetching (GRP). The hardware engine generates prefetches triggered by L2 cache misses when the missing load contains a hint. A sophisticated compiler analysis produces a rich set of 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, yet it need not struggle to deduce future application references by performing complex pattern matching on prior accesses.

Using previously proposed techniques, GRP's 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 the compiler support, this prefetching hardware is effective at improving performance, but consumes copious amounts of bandwidth. GRP's compiler analyses inform 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; *pointer*: prefetch by following the pointer in the load's cache line; *recursive*: prefetch this pointer data structure recursively. The compiler also generates indirect prefetching instructions which trigger prefetching a set of references using an indirection array.

With this cooperative hardware/software interface, the memory system is able to obtain the high performance of previously proposed scheduled region prefetching (SRP) [26] on most of the SPEC2000 benchmarks, and a speedup of over 10% on two others. Table 1 shows a summary of the GRP results using the geometric mean.

Without prefetching, the mean performance across the benchmark suite is 33% lower than would be obtained by a perfect level-two cache, as shown in the rightmost column, a significant drop in performance. Following Sherwood et al. [36], stride prefetching provides an 15% speedup over no prefetching. SRP, which uses no compiler analysis, outperforms stride prefetching by 7%, and closes the gap between performance with SRP and performance with a perfect L2 cache to 18%. However, SRP consumes excessive memory bandwidth, a 153% increase over the no prefetching case. GRP, conversely, provides near-equal performance to SRP (a mean of 0.4% lower, with gains of greater than 10% for two benchmarks). GRP performance is actually consistently slightly higher than SRP on average, but incurs a large drop for one benchmark (mcf) which counteracts numerous smaller gains across the benchmark suite. More important, GRP provides equivalent performance to SRP but with substantially less traffic, an increase of only 31% over the no-prefetching case. This reduction in traffic saves power and also will provide performance gains in a multiprocessor setting.

We review related work in Section 2, showing how little of it attempts to find 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 necessary compiler analysis in detail. Section 5 evaluates the success of the GRP engine at bringing performance of most benchmarks close to a perfect L2 cache while reducing memory traffic. We also compare it with stride prefetching [36]. 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. Of those four, one simply requires more memory bandwidth, and three need more software/hardware assistance.

2. Related Work

In this section, we focus on the pertinent aspects of copious prior work in software and hardware data prefetching, along with the small number of previously proposed hybrid schemes.

Software prefetching relies on non-binding prefetch instructions which 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 which loads will cause cache misses at runtime 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) [5, 17, 31, 43].

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 [24]. For arrays, the compiler can compute the address directly and solves this problem fairly well [4, 31]. For greedy pointer prefetching [3, 28, 34], it is limited because it can only schedule a prefetch when it knows the effective address. "Jump" pointers identify a record n links ahead in the structure, but require much more sophisticated analysis [3, 28, 34]. Other approaches prefetch pointer arguments at call sites [27], and decouple prefetches from the main program using a separate thread context [11, 13, 29].

In contrast, the runtime behavior of a program's regular missing load instructions triggers hardware prefetching. Since the 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 [26]. However, unlike the compiler, the hardware has no direct knowledge of future memory references; the key challenge in hardware-based prefetching is coming up with 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 be extremely difficult to predict).

Because of the need to predict addresses, many hardware prefetchers exploit only spatial locality, prefetching one or more subsequent blocks on a cache miss [14, 22, 38]. More sophisticated schemes detect non-unit strided access patterns, such as Chen and Baer's reference prediction table (RPT) [8] and Palacharla and Kessler's strided stream buffers [32]. Other approaches prefetch pointer-based access patterns, as with correlation-based and Markov prefetching [1, 7, 21], or a broader class of prefetches, using dead block information [25] all of which predict cache blocks to prefetch based on repeated past access patterns. Another approach involves decoupling the data structure traversal from the computation, using specialized pointer-traversal hardware [33] or dedicated pre-execution hardware [2]. Researchers have also proposed memory-side prefetching to reduce latencies between prefetches [19, 39, 44].

Most pertinent to this work are two previous papers. First, predictor-directed stream buffering, proposed by Sherwood et al. [36], unifies strided stream buffers and Markov prefetching into a single, consistent hardware prefetching framework. We compare the GRP scheme to this scheme in Section 5, except without the Markov predictor which consumes too much state to be practical. Second, Cooksey et al. [12] recently proposed 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.

In the end, all hardware schemes are forced to trade coverage for accuracy (or vice versa), and either focus only on structured access patterns which can be predicted with high accuracy (forgoing coverage of less structured access patterns), or expend significant amounts of useless bandwidth 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, using 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 overly restrictive access patterns, or

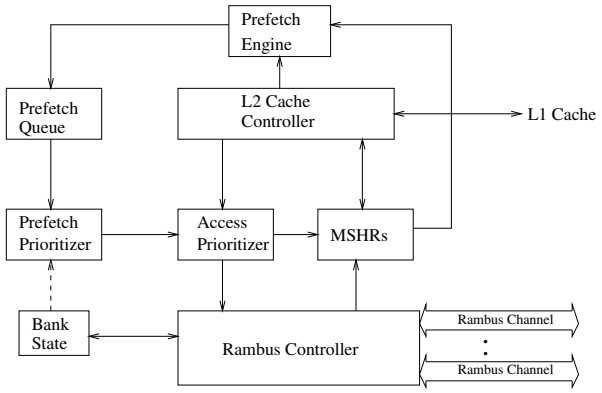

Figure 1: Prefetch Engine Organization

provided an interface that provides too much generality and complexity. 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 [9, 10] 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 [37]. Zhang and Torrellas [45] use the compiler to program bits in memory that identify contiguous spatially local regions and mark pointers for fast forwarding. Since these bits mark addresses and imply specific policies, their flexibility is limited. Karlsson et al. [23] use a technique called prefetching array (PA) to address the problems of short list traversal and unknown tree traversal path in greedy prefetching and jump-pointer prefetching. They insert a loop or a special instruction to trigger prefetches. Their technique shows less performance improvement for a lower bandwidth system. This problem is exactly the focus of our work. On the general side, fully programmable prefetch engines provide much flexibility but require significant memory system support and have not demonstrated that the required compiler support is realistic [41, 44, 39].

This paper describes GRP which combines the best of software and hardware prefetching in a scheme that is simple, yet effective. We convey sophisticated compiler analysis by associating a range of hints with loads, which an aggressive, simple, and general hardware prefetcher uses as necessary. Thus, we convey much of the information that the compiler knows, 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 issue an aggressive yet accurate stream of prefetches.

3. Hardware Prefetching Engine

Our hardware prefetching engine builds on the scheduled region prefetching design by Lin et al. [26]. We extend the original design in two dimensions. 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) [26] aggressively exploits spatial locality by attempting to prefetch large (4 KB) memory regions on each L2 cache miss. The two negative effects of aggressive prefetching—memory bus contention and cache pollution—are addressed directly by reducing the priority of prefetches in memory 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 1 shows the memory system with the SRP engine that forms our prefetching experimental baseline. The access prioritizer is the central component of our 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 prefetched. 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. 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 the next prefetch candidate block after the new miss block, and moves the entry to the head of the queue. In this work, we use a 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. Under LRU replacement, the controller puts prefetched data in the LRU position; it moves a block to the MRU position only if it is referenced directly by the CPU. As a result, useless prefetches in an n-way associative cache can displace at most one nth of the least 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 [26] shows this effect to be insignificant. The final prioritization optimization is to issue 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 [26]. However, it has two shortcomings which we address here. First, SRP does not provide any benefit for non-spatial reference patterns, such as those generated by pointer-based data structures and indirect array references. We add a pure hardware pointer prefetching mechanism to address this issue (see Section 3.2). We also add an indirect array scheme that works only with compiler hints (see Section 3.3). Second, SRP can produce copious amounts of excess memory traffic. Although this useless traffic has minimal performance impact thanks to SRP's prioritization techniques, it consumes energy, takes

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

Table 2: Compiler Hints for Representative References in Loops

bandwidth from useful prefetches, and may reduce its effectiveness in a multiprocessor environment. We thus use compiler hints for spatial and pointer accesses to gain both low bandwidth and accuracy. We describe the GRP hardware modifications and hints below in Section 3.3, and the compiler analysis itself in Section 4.

3.2 Prefetching for 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, our base pointer prefetching scheme simply greedily generates a prefetch for any fetched value that falls within the ranges of legitimate heap memory addresses. Our 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. Cooksey et al. [12] describe a similar but more efficient pointer test using bit masks, and apply it to prefetching in the more challenging IA32 environment.

Once the controller identifies a datum as a possible pointer value, it translates the virtual address to a physical address and forwards it 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 is useful when the prefetch address is near the end of its enclosing block). This mechanism thus does not chase recursive pointers.

3.3 GRP: Incorporating Compiler Prefetch Hints

This section describes our compiler hints and how the GRP hardware uses them to improve the precision of L2 spatial and pointer prefetching, reducing the potentially large amount of wasted bandwidth.

We introduce a communication channel between the compiler (or other software) and the prefetch engine. The compiler annotates load instructions with hints predicting whether particular spatial or pointer-based prefetches based on the load's address will be useful. (In this study, the compiler conveys the hints using the otherwise 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 four hints and shows typical representative code snippets for each. We summarize the changes to the hardware for each hint below, and then describe 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 *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 data for pointer values and generates prefetches only for these 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 result n levels into the recursive data structure. (We use $n = 6$ in our experiments).

GRP for Pointer References

GRP uses the same mechanism for pointer and recursive pointer hints. However, GRP applies the mechanism once 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 pinter 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.

Let pointer include single level and recursive pointers in the remainder of this discussion. 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 line in our configuration. Two-block fetching is sufficient considering structure alignment. The engine thus terminates after one level for pointers and six levels for recursive prefetching.[1] (Unlike Cooksey et al. [12], we do not extend the pointer or recursive pointer lookahead as prefetched blocks are referenced, but wait for another miss.)

GRP for Indirect Array References

A few of the benchmarks from the SPEC2000 suite 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. We introduce a specialized extension of GRP to target these patterns. A single *indirect prefetch* instruction conveys both a base address (`&a[0]`), an element size (`&sizeof(a[0])`), and an indirection 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

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

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 element size for index arrays is 4, which is a typical size on most systems. Since the compiler can detect the element size, it can easily pass it to the run-time if necessary.

This scheme is unique among all 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). In practice, this hint could be encoded as a store, with the effective address corresponding to one parameter and the store data to the other. 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.

4. Compiler Analysis Framework

This section describes the analysis that results in the four classes of hints (*spatial*, *indirect*, *pointer*, *recursive pointer*) used to 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 indicates 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 generates a 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 [30, 42]. Dependence testing first finds induction variables and then detects when the affine 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. This 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 2, 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 can determine the loop bounds and steps of the index variables, 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 loop levels. The compiler is conservative when reuse distance is unknown; we only mark a reference as spatial if its spatial reuse is in the innermost enclosing loop.

```
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 6: Algorithm generating spatial and non-spatial hints

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 3, *buf* is a heap array with type *T***. This compiler algorithm can find a wide set of spatial reuse. In addition to detecting the obvious spatial reuse of $buf[i][j]$ when j is an loop induction variable, the compiler can 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 deferences that show spatial locality, as illustrated in Figure 4, the compiler performs loop induction variable recognition on pointers to find constant increments of pointers. The type *T* in Figure 3 and Figure 4 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. The ensuing analysis on L2 cache misses shows that the cases in Figure 3, Figure 4, and regular spatially local array references, together cover almost all spatial reuses in C code.

4.3 Spatial Algorithm

Figure 6 summarizes the algorithm used for generating spatial hints for both arrays and spatial pointer accesses. The dependence testing requires affine subscription expressions[2]. 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. Our algorithm is intra-procedural and flow in-

[2]An expression is *affine* if it can be represented by a linear of expression of loop induction variables.

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

Figure 2: Sample Fortran code

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

Figure 3: Typical heap array in C

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

Figure 4: Typical induction pointer in C

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

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

Figure 5: Typical recursive pointer in C

sensitive. If a reference in a routine is not enclosed in a loop, it is not marked.

4.4 Indirect Analysis

The compiler also detects and marks indirect array accesses, such as $c(b(i), j)$ in Figure 2. 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 case, and generate an indirect prefetching instruction using the address of $b(i)$ and the base address of array c, as previously described.

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 to a field reference from a structure containing 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 say a pointer update is *recursive* if it updates itself in a loop with an object of the same data type. In Figure 5, 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 3 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 complete algorithm to generate pointer hints is shown in Figure 7. It is complementary to the spatial marking algorithm shown in Figure 6, which marks spatially local pointer accesses.

5. Experimental Evaluation

```
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 7: Algorithm generating pointer and recursive pointer hints

In this section, we compare the performance benefits of SRP, GRP, and unified stride prefetching for the SPEC CPU2000 benchmarks. We demonstrate that GRP provides a compelling balance between higher performance and increased memory traffic among the three prefetching techniques. We conclude with a discussion of the characteristics of the remaining three benchmarks for which GRP does not eliminate main memory as a bottleneck.

5.1 Experimental Methodology

The Scale compiler infrastructure inserts the prefetch hints. 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 use the 17 SPEC CPU2000 C and Fortran benchmarks for which the Scale infrastructure generates performance comparable to the commercial Alpha compiler. Table 3 lists these benchmarks, along with statistics on memory instructions and compiler hints. 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*. Column 6 lists

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

Table 3: Number of compiler hints for each benchmark

the fraction of static memory operations with hints. Note that the compiler can mark an instruction both *spatial* and *pointer*. Column 6 shows the static number of indirect prefetch instructions.

We simulated these binaries on a version of sim-alpha [16] with scheduled region prefetching (SRP) [26] added to the simulator. We added the GRP pointer prefetching mechanisms, and modified the simulator to accept compiler hints and schedule prefetches accordingly if the binaries contain the hints. The simulator is configured to simulate 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 1 and 16 cycles, respectively. Each cache contains 8 MSHRs. For SRP, the prefetching queue size is 32 and uses LIFO scheduling. The stride predictor [36] uses a 4-way history table with 1K entries. There are 8 entries in each of 8 streaming buffers sharing the history table.

We use the SimPoint [35] tool set to detect the initialization phase of the benchmarks, and use a starting simulation point beyond the initialization phase. All performance numbers use 200M instructions. Figure 8 shows the performance using IPC of the system with a realistic memory hierarchy versus performance with a perfect memory system (perfect L1) and a perfect L2 cache. We sort the benchmarks by the difference between a realistic memory hierarchy and one with a perfect L2 cache, which is a gap of 33% on average (geometric mean). Of these benchmarks, only *crafty* shows a negligible L2 miss rate (0.4%). Consequently, we do not include results for *crafty* in subsequent experiments.

5.2 Performance Comparison of Stride Prefetching, SRP, and GRP

In this section, we first present the effects of both pointer prefetching 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.

We applied pointer prefetching alone to all benchmarks. Of course, it does not improve performance on the Fortran benchmarks. Seven C benchmarks show a significant performance improvement, notably a 48.3% boost for *equake* and a 15.9% increase for *mcf* as shown in Figure 9 (higher IPC is better). 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 the loop which sequentially initializes a heap array. The array element is a structure and pointer prefetching happens to prefetch the elements accessed later. For *equake*, when combined with region prefetching (SRP+Pointer), the performance gain is about the same as applying region prefetching only. For *mcf*, the combination degrades the performance due to low prefetching accuracy. Pointer prefetching outperforms SRP only for *twolf* with 2%, and recursive pointer prefetching achieves an additional one-half percent. 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 SPEC2000 integer and floating point benchmarks respectively. In most cases and on average, SRP and GRP both perform better than stride prefetching. SRP narrows the gap from a perfect L2 to within 10% for 10 benchmarks. For *swim* and *bzip2*, GRP performs over 10% better than SRP due to its lower traffic or indirect prefetching. 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 some hidden localities outside of loops. Stride prefetching shows a better IPC than GRP and SRP in *mesa* and *art*. But the gap is small.

GRP applies indirect prefetching to *vpr* and *bzip2*. Although we detect indirect references among 11 benchmarks, indirect prefetching only shows significant gains in performance for *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 7.7% from 16.7%, with only 15% of the memory traffic of SRP.

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 reports the normalized memory traffic for the three prefetch schemes. SRP increases memory traffic from 2% to a *factor of* 25.6 times over no prefetching. GRP causes only 31.4% additional traffic, on average, versus an SRP increase of 153.4%, compared to no prefetching. GRP eliminates over 20% of the total memory traffic for nine of the sixteen benchmarks–compared to SRP–and over 50% of its memory traffic for five benchmarks. The traffic for stride prefetching is 17.3% less than GRP, but stride prefetching only achieves 68% of the performance improvement that GRP provides.

Table 4 shows prefetching accuracies and prefetching coverage for the three prefetching techniques we implemented. We use the percentage miss reduction as a measurement for prefetching coverage. On average, SRP provides the best coverage and the worst accuracy. Stride prefetching trades the lowest coverage with the highest accuracy. GRP has an accuracy that is closer to stride prefetching, but coverage closer to SRP.

Since the normalized traffic in Figure 12 does not reflect the absolute bandwidth consumption of each benchmark, we list the actual memory traffic of each benchmark in Table 4. On average, SRP

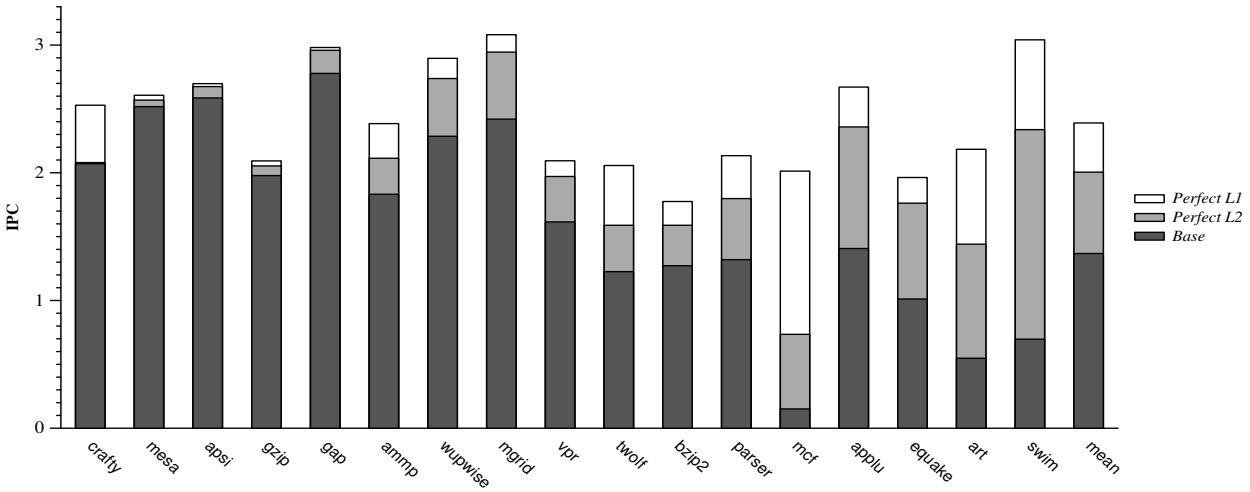

Figure 8: Processor performance

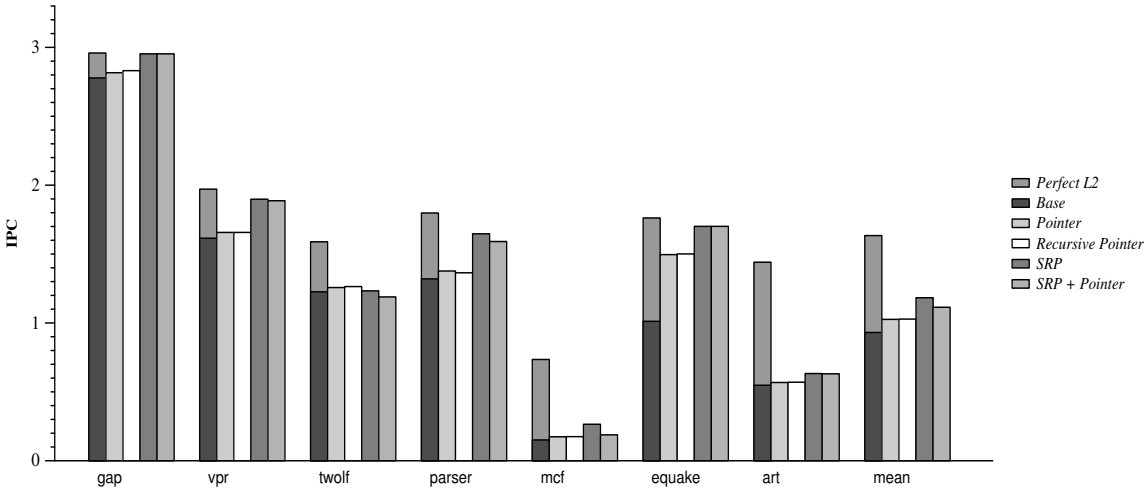

Figure 9: Performance gains from pointer prefetching

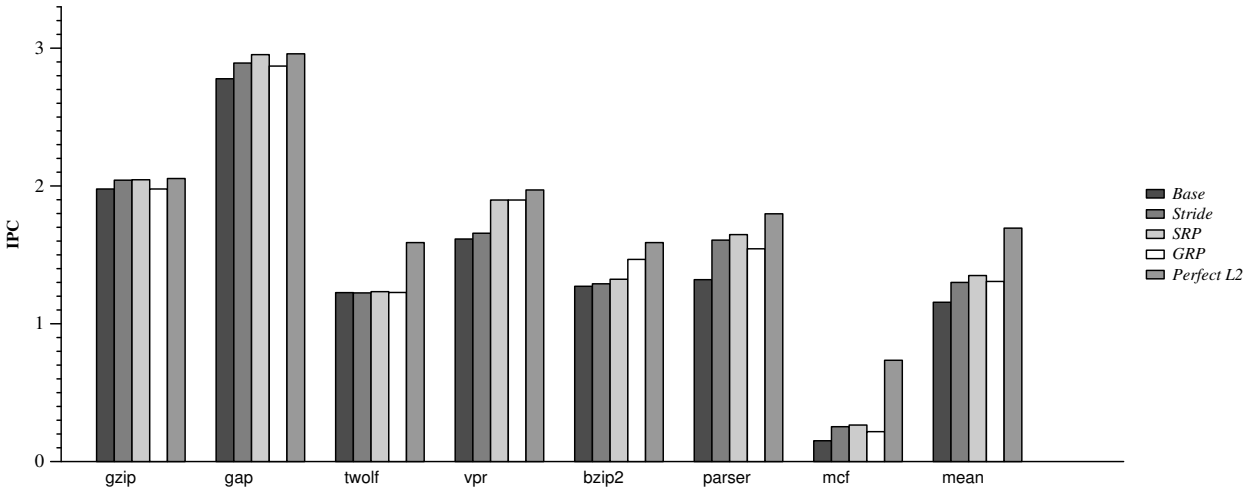

Figure 10: Performance gains from region prefetching and stride prefetching for integer benchmarks

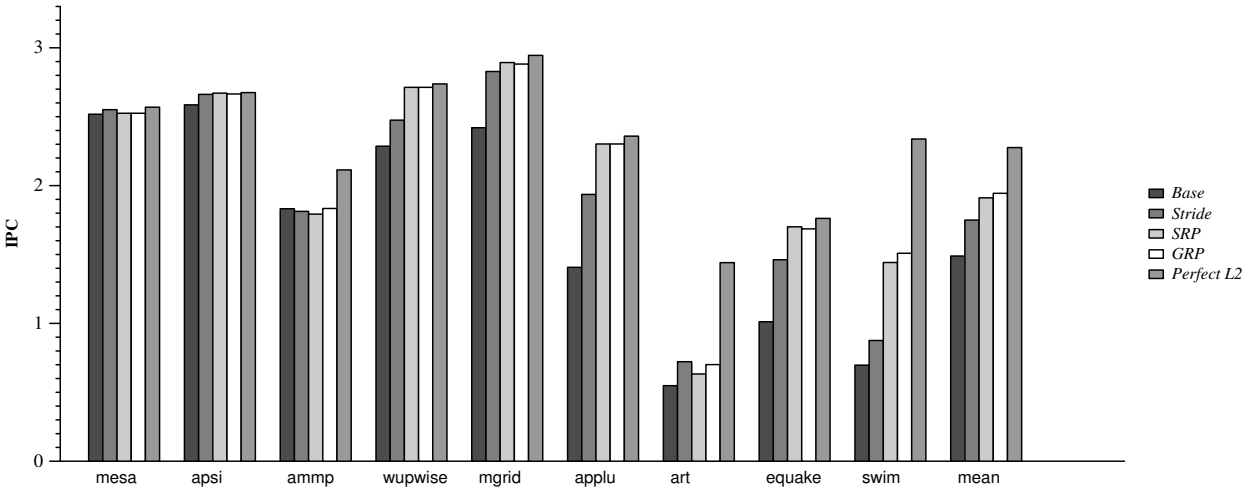

Figure 11: Performance gains from region prefetching and stride prefetching for floating-point benchmarks

Benchmark	Performance Gap (%)	miss characteristics	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	16.74	indirect array reference	66.58
300.twolf	22.40	linked list and random pointers	35.37

Table 5: Level 2 miss characteristics

consumes 86.57% more memory bandwidth over the no-prefetching system. GRP and stride prefetching report a 17.1% and 9.9% increase in memory requests, respectively.

5.4 Compiler Sensitivity

We implemented an aggressive compiler scheme and a conservative scheme to compare with the scheme discussed in Section 4. The aggressive marks a reference as *spatial* even its reuse distance is greater than L2 cache size. The conservative scheme marks a reference as *spatial* only when its reuse sits in the innermost loop. The aggressive scheme 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* by from 1% to 34%, and reduces performance by an average of 5% across the benchmark suite.

5.5 Remaining Issues

Six of the benchmarks show a gap of greater than 10% between SRP and a perfect L2. We list them in Table 5 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* under 10%, and improve *ammp* as well. *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 [6]. *art* is bandwidth bound. While the reduced GRP prefetches reduce traffic and increase performance over SRP by 10.7%, the performance gap is still large. Larger caches and wider channels improve *art* appreciably. Finally, *mcf* and *twolf* contain heavy traversals of short linked lists and tree data structures, making them unamenable to either the GRP pointer or spatially-based schemes.

6. Conclusions and Future Work

Main memory access latencies are a significant performance issue in modern systems. Purely compiler-based prefetching techniques have difficulty managing such large latencies. 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 goes up, this bandwidth will become increasingly precious as well.

This paper shows that a cooperative approach between compiler-based analysis and hardware-based aggressive prefetching provides similar benefits at a much lower bandwidth cost. Compiler techniques identify accesses that clearly possess spatial locality statically. 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 on accesses without locality.

We also extend the scheduled region prefetching engine to address pointer-based applications by aggressively prefetching any datum which 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 distinct objects. It remains to be seen whether this phenomenon will apply to the benchmarks that other researchers used to show the importance of greedy pointer hardware prefetching [12].

Finally, we note that with just the spatial and indirect hints, our 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-list or tree traversals (*mcf*, *twolf*) where memory-side prefetching may help. For the rest of

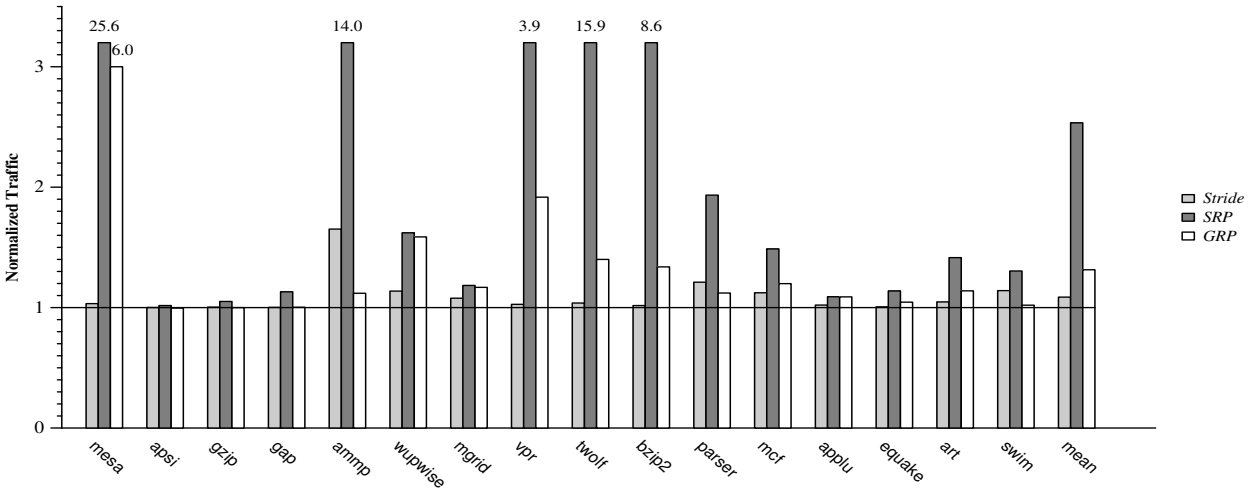

Figure 12: Normalized traffic

Benchmark	Base		Stride			SRP			GRP		
	miss rate	traffic	coverage	accuracy	traffic	coverage	accuracy	traffic	coverage	accuracy	traffic
mesa	9.3	50K	68.8	95.6	52K	29.0	1.4	1301K	25.8	5.8	303K
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	1356K	8.0	85.7	1380K	28.6	5.3	11624K	46.0	48.1	1817K
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
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	39.3	5309K	45.3	81.4	5834K	61.1	52.3	9857K	51.4	67.9	6218K

Table 4: Prefetching accuracy, coverage and memory traffic

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.

7. 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] 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.

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

[5] 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.

[6] 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.

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

[8] 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.

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

[10] 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.

[11] 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.

[12] 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*, October 2002. To appear.

[13] D. Y. D. Kim. 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*, San Jose, CA, Oct. 2002.

[14] 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.

[15] 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.

[16] R. Desikan, D. Burger, and S. W. Keckler. Measuring experimental error in microprocessor simulation. In *Proceedings of the 28th International Symposium on Computer Architecture*, pages 266–277, June 2001.

[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*, Toulouse, France, Jan. 2000.

[24] 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.

[25] 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.

[26] 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*, 2001.

[27] 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*, 1995.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

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

[35] 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*, 2001.

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

[37] 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.

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

[39] 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.

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

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

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

[43] 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*, Berlin, Germany, June 2002.

[44] 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.

[45] 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.