Comparing and Combining Read Miss Clustering and Software Prefetching *

Vijay S. Pai† and Sarita V. Adve‡

†Electrical and Computer Engineering
Rice University
Houston, TX 77005
vijaypai@rice.edu

‡Computer Science
University of Illinois
Urbana-Champaign, IL 61801
sadve@cs.uiuc.edu

Abstract

A recent latency tolerance technique, read miss clustering*, restructures code to send demand miss references in parallel to the underlying memory system. An alternate, widely-used latency tolerance technique is software prefetching, which initiates data fetches ahead of expected demand miss references by a certain distance. Since both techniques seem to target the same types of latencies and use the same system resources, it is unclear which technique is superior or if both can be combined. This paper shows that these two techniques are actually mutually beneficial, each helping to overcome limitations of the other.*

We perform our study for uniprocessor and multiprocessor configurations, in simulation and on a real machine (Convex Exemplar). Compared to prefetching alone (the state-of-the-art implemented in systems today), the combination of the two techniques reduces execution time an average of 21% across all cases studied in simulation and an average of 16% for 5 out of 10 cases on the Exemplar. The combination sees execution time reductions relative to clustering alone averaging 15% for 6 out of 11 cases in simulation and 20% for 6 out of 10 cases on the Exemplar.

1 Introduction

Modern processors address long cache miss latencies with techniques to exploit instruction-level parallelism (ILP), such as out-of-order issue and non-blocking and overlapped reads. Previous work found that such ILP hardware techniques were often ineffective in targeting data memory stalls because code typically did not expose enough parallelism to exploit the features of the hardware [23]. Motivated by this observation, we have recently proposed code transformations that improve memory parallelism by clustering multiple independent read misses within the processor's out-of-order instruction window (called *read miss clustering*) [22]. An alternate, widely-used latency-tolerance technique is software-controlled non-binding prefetching. Prefetching helps tolerate latencies by initiating (often multiple overlapping) data fetches ahead of expected demand misses [4]. On the surface, both techniques seem to target the same types of latencies and use the same system resources; therefore, without further analysis, it is unclear which technique is superior or if both techniques can be used together. Since prefetching is already widely used, such an analysis is required for system designers to incorporate read miss clustering in real systems. This paper provides such an analysis. It shows that the two techniques are actually mutually beneficial, each helping to overcome the performance limitations of the other. This result is surprising because prefetching is widely believed to be an effective latency tolerance technique that can already exploit parallelism in the memory system (by sending multiple prefetches in parallel).

The read miss clustering transformation is based on a novel adaptation of unroll-and-jam [22]. Specifically, it extends unroll-and-jam by mapping memory parallelism in a modern ILP system to the previously-studied problem of floating-point pipelining [2, 3, 20]. The new transformation aims to cluster multiple expected read misses together within the same instruction window of an out-of-order processor, without sacrificing data locality. Read miss clustering was experimentally shown to improve latency tolerance and exploit hardware support for memory parallelism. However, this technique leaves at least some misses exposed as stalls, since it hides later read misses behind the stall time of earlier misses. Further, certain dependences in the code can prevent the needed code restructuring.

Software prefetching is a widely used latency-tolerance method implemented on many commercial systems. The most commonly known and implemented software prefetching algorithms apply software pipelining to the innermost loop for a given miss reference [17, 18, 19]. Such software pipelining creates a prologue, which prefetches data for the first iterations; a steady-state, which includes computation along with prefetches scheduled ahead by a

*This work is supported in part by an IBM Partnership award, Intel Corporation, the National Science Foundation under Grant No. CCR-9410457, CCR-9502500, CDA-9502791, and CDA-9617383, and the Texas Advanced Technology Program under Grant No. 003604-025. Sarita V. Adve is also supported by an Alfred P. Sloan Research Fellowship.

certain number of iterations termed the *prefetch distance*; and an epilogue, with only computation for the last iterations. However, the lack of computation in the prologue typically prevents that phase from overlapping the latencies of its prefetches, leading to *late* prefetches and exposed stall latencies. Further, the prefetch distances required for many loops are excessive, shrinking the prefetching steady-state. Some references are also inherently difficult to prefetch since their addresses are not known sufficiently in advance (e.g., linked list traversals). Software prefetching also adds instruction overhead for both the prefetches and their address-generation instructions. Other limitations of current prefetching algorithms are discussed in Section 3.

In this study, we show that read miss clustering and software prefetching can profitably combine to address limitations of either technique alone. Prefetching can help to tolerate latencies remaining after miss clustering. Clustering can help prefetching by reducing the stalls caused by prologue late prefetches, by decreasing the needed prefetch distance and increasing the steady state length, and by extracting parallelism among demand misses for references that are hard to prefetch.

We evaluate these latency tolerance techniques both with a detailed simulator (RSIM) and on a real system (Convex Exemplar), applying miss clustering by hand in both cases and applying prefetching by hand for the simulation. We consider prefetching for both regular and irregular applications [15, 16, 17, 27]. For the applications and systems we study, clustering alone outperforms prefetching alone for most cases. This result, however, is sensitive to system trends, and there may be some applications where clustering is not applicable but prefetching is. More importantly, this paper finds that the combination of read miss clustering and prefetching yields better execution time benefits than either technique alone in most cases, and we expect this trend to continue. Specifically, for our systems and applications, compared to prefetching alone (the state-of-the-art implemented today), the combination of the two techniques reduces execution time an average of 21% across all cases studied in simulation and an average of 16% for 5 out of 10 cases on the Exemplar. The combination sees execution time reductions relative to clustering alone averaging 15% for 6 out of 11 cases in simulation and 20% for 6 out of 10 cases on the Exemplar.

2 Background

2.1 Read Miss Clustering

Instructions in an out-of-order processor's instruction window (reorder buffer) can issue and complete out-of-order. To maintain precise interrupts, however, instructions commit their results and retire from the window in-order after completion [30]. The only exception is for writes, which can use write-buffering to retire before completion.

Because of the growing gap in processor and memory speeds, external cache misses can take hundreds of processor cycles. However, current out-of-order processors typically have only 32–80 element instruction windows. Consider an outstanding read miss that reaches the head of the window. If all other instructions in the window are fast (e.g., typical computation and read hits) or can be buffered aside (e.g., writes), the independent instructions may not overlap enough latency to keep the processor busy throughout the cache miss. Since the later instructions wait to retire in-order, the instruction window will fill up and block the processor. Thus, this miss still incurs stall time despite such ILP features as out-of-order issue and non-blocking reads.

Suppose that independent misses from elsewhere in the application could be scheduled into the instruction window behind the outstanding read miss. Then, the later misses are hidden behind the stall time of the first miss. Thus, read miss latencies can typically be effectively overlapped only behind other read misses, and such overlap only occurs if read misses to multiple cache lines appear clustered within the same instruction window. This phenomenon is termed *read miss clustering*, or simply *clustering* [22].

Recent work discusses read miss clustering more thoroughly and proposes a compile-time algorithm for increasing read miss clustering for modern out-of-order processors [22]. The algorithm is based on a novel adaptation of unroll-and-jam, by which an outer-loop is unrolled and the resulting inner-loop copies are fused (jammed) together [2].

Figures 1(a) and 1(b) provide pseudocode for the base and clustered code, respectively. The pseudocode shows a traversal of a 2-D matrix with dimensions $I_o \times I_i$, before and after unroll-and-jam. (All pseudocode in this paper uses row-major notation.) The initial code is a row-wise traversal, optimized for spatial locality. Successive read misses are separated by as many iterations as there are words in a cache line, L (typically 4–16 for the 32–128 byte cache lines in modern microprocessors). Thus, if the instruction window holds only L or fewer iterations, no read miss clustering is available. A column-wise traversal would facilitate clustering by referencing different rows of the matrix in successive iterations, but would destroy locality if there were more rows in the matrix than available cache lines. Using unroll-and-jam for read miss clustering (Figure 1(b)) effectively stops the "column-wise" traversal as soon as the hardware resources for overlap are filled, allowing clustering without sacrificing locality. The above use of read miss clustering is graphically represented in Figures 2(a) and 2(b). These figures represent the traversal order of a matrix laid out in row-major order, with crosses for data elements and shaded blocks for cache lines.

Unroll-and-jam has been used previously to improve floating-point pipelining in the presence of recurrences (cy-

```
for(j=0;j<I_o;j++)
 for(i=0;i<I_i;i++)
  ···A[j,i]
```
(a) Original code

```
for(j=0;j<I_o;j+=N)
 for(i=0;i<I_i;i++)
  ···A[j,i]
  ···A[j+1,i]
  ......
  ···A[j+N-1,i]
```
(b) After clustering alone

```
for(j=0;j<I_o;j++)
 for(i=0;i<d;i+=L)
   PF(&A[j,i])

 for(i=0;i<I_i − d;i++)
  if(i mod L ≡ 0)
     PF(&A[j,i+d])

  ···A[j,i]

 for(;i<I_i;i++)
  ···A[j,i]
```
(c) After prefetching alone

```
for(j=0;j<I_o;j+=N)
 for(i=0;i<d';i+=L)
   PF(&A[j,i])
   ......
   PF(&A[j+N-1,i])
 for(i=0;i<I_i − d';i++)
  if(i mod L ≡ 0)
     PF(&A[j,i+d'])
     ......
     PF(&A[j+N-1,i+d'])
  ···A[j,i]
  ......
  ···A[j+N-1,i]
 for(;i<I_i;i++)
  ···A[j,i]
  ......
  ···A[j+N-1,i]
```
(d) After both

Figure 1. Pseudocode of a 2-D matrix traversal, (a) as originally generated, (b) after read miss clustering with unroll-and-jam (postlude not shown), (c) after software prefetching, and (d) after the combination of clustering and prefetching. All pseudocode uses row-major notation.

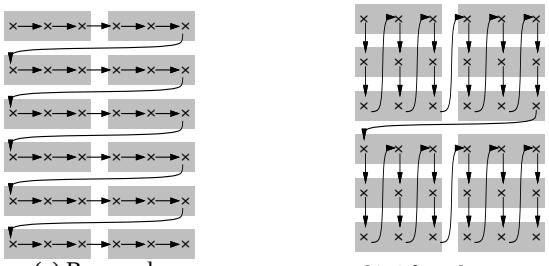
cles in the dependence graph) [3, 20]. We have mapped floating-point pipelining to memory parallelism and defined two classes of dependences that limit memory parallelism (address dependences and cache-line sharing dependences), incorporating instruction-window constraints both during and after the application of unroll-and-jam [22]. The analysis and transformation expose opportunities for clustering both regular and irregular memory references while maintaining locality.

2.2 Software Prefetching

The following discusses the best known and implemented software prefetching algorithm for regular references, as well as algorithms for irregular references.

2.2.1 Algorithm for Adding and Scheduling Prefetches

The best known software prefetching algorithm implemented in a compiler is the loop-based algorithm of Mowry et al. [17, 18, 19]. The *analysis* phase of the algorithm identifies the static references that can miss (leading references). Then, the *scheduling* phase uses loop peeling,



(a) Base code (b) After clustering

Figure 2. Impact of matrix traversal order on read miss clustering. Crosses represent elements, and shaded blocks represent cache lines. The matrix is shown in row-major order.

unrolling, and strip-mining to insert prefetches only for the dynamic instances of leading references that are expected to miss. The innermost loop for a miss reference is software pipelined to schedule a prefetch ahead of the demand access by a certain number of iterations, called the *prefetch distance*. The prefetch distance (d) is computed as $d = \min(L \times \lceil \frac{D}{CL} \rceil, I_i)$, where D is the expected miss latency in cycles, C estimates the shortest possible path through an iteration in cycles, L is the number of successive inner-loop iterations that share a cache line, and I_i represents the total number of inner-loop iterations (the upper limit on inner-loop software pipelining). The term $L \times \lceil \frac{D}{CL} \rceil$ represents the distance needed to completely overlap the prefetch latency.

Software pipelining produces a prologue, steady-state, and epilogue from the original inner loop. The prologue consists only of prefetches to cover the first d iterations. The steady-state includes both prefetches (scheduled according to the prefetch distance) and computation for $I_i - d$ iterations. Either strip-mining, unrolling, or a conditional test is used to insure that a prefetch is issued for a reference only once for every L iterations of the original loop. The epilogue includes only computation for the last d iterations of the original inner loop; no prefetches are issued since all the inner-loop iterations have already been prefetched in the prologue and steady-state. Figures 1(a) and 1(c) illustrate a matrix traversal before and after applying software prefetching, respectively.

2.2.2 Extensions for Irregular Memory References

The above algorithm handles only affine references, but newer prefetching algorithms have been developed to support two classes of irregular references. The first class supports references formed by indirection through an affine reference [17]. Such references require two prefetches: one for the affine reference and one for the indirect reference itself. Since the address calculation of the second prefetch uses the data of the first, the first prefetch must be scheduled before

the second according to the prefetch distance. This effectively doubles the needed prefetch distance.

More recent research has focused on linked data structures based on pointer-chasing (e.g., linked lists). Software prefetching techniques for linked data structures use *jump pointers*, naturally-occurring or artificially-created pointers to later elements in the traversal [16, 27]. A recent study also adds a *prefetch array* containing pointers to the first elements of a linked data structure, thus allowing a prefetching prologue [15]. However, the prefetch arrays themselves can cause new cache misses, and prefetch arrays were actually seen in that study to degrade performance on some bandwidth-limited systems.

Linked-data structure prefetching schemes are limited by the jump pointers used. For example, each node of a singly-linked list has a single naturally-occurring jump pointer to the next element in the traversal. Consequently, the prefetch distance is limited to 1 iteration for techniques that only use naturally-occurring jump pointers (e.g., *greedy prefetching* [16]). Further, even artificial jump pointers may not help for structures such as hash-tables, as these are typically dominated by traversals of very short lists (limiting the maximum prefetching distance) and also see little work per iteration. We consider two linked-data structure prefetching schemes: greedy prefetching [16] and prefetch arrays [15]. Since the linked-data structure application we study is dominated by accesses to hash tables with short lists, schemes based on longer artificial jump pointers are not applicable.

3 Limitations of Read Miss Clustering and Software Prefetching

This section discusses problems that limit read miss clustering and software prefetching, as well as limitations shared by both schemes.

3.1 Limitations of Read Miss Clustering

Incomplete latency hiding. A demand read miss that has not completed by the time it reaches the head of the instruction window will block retirement, incurring data memory stall time. Read miss clustering alone usually leaves some latencies exposed, since later misses are hidden behind the stall time of earlier misses.

Legality issues. Certain dependences can prevent the inner-loop fusion step required by unroll-and-jam [3, 7]. These constraints can limit the applicability of clustering.

3.2 Limitations of Software Prefetching

Prologue late prefetches. The prefetching prologue is meant to cover the references of the first few steady-state iterations. However, the prologue is unlikely to hide their latencies since it contains no computation. Thus, the data requested by some of these prefetches arrives after their demand references, leading to *late* prefetches and exposed latencies. Since the prologue is invoked each time the inner loop starts, prologue late prefetches arise on each outer-loop iteration of a loop nest[1]. Prefetching schemes that do not include a prologue would also see such exposed latencies before each steady-state.

Short steady-states. Because of the deficiencies in the prologue, effective prefetching depends on most inner-loop iterations fitting in the steady-state. Using the terms of Section 2.2, we see that the steady state has $\max(0, I_i - L \times \lceil \frac{D}{CL} \rceil)$ iterations. Indirect prefetching effectively doubles the needed prefetch distance, further shrinking the steady-state.

The above shows that a large steady-state requires an inner loop with a large number of iterations or a large amount of computation per iteration. Many loops do not meet these requirements. First, loops blocked for cache locality or fine-grained communication tend to have few iterations. Second, each loop iteration often includes little actual computation. Inner-loop unrolling also cannot help, since increases in C are offset by decreases in I_i.

Hard-to-prefetch references. Some references have addresses that are difficult to calculate sufficiently in advance, making them hard to prefetch. Examples include greedy prefetching of linked lists (which limits the available prefetch distance to one iteration) and references with unanalyzable address calculations.

Instruction overhead. Prefetch instructions and their address generation overhead increase dynamic instruction count and CPU computation time. As a result, the latency-tolerance benefits of prefetching do not always translate directly to overall execution time benefits.

Other limitations. We call a prefetch *unnecessary* if it already hits in the L1 or L2 cache, *early* if the prefetched line is evicted before its demand reference, and *damaging* if the prefetch evicts some other useful line from the cache. All of these classes of prefetches incur overhead and can hurt performance. Some prefetching schemes (e.g., prefetch arrays) can also introduce extraneous data fetches, increasing resource contention.

3.3 Limitations Shared by Both Techniques

Both read miss clustering and software prefetching can increase resource contention, increasing total system latencies [22, 23]. Both techniques can also introduce new conflict misses by increasing the number of active lines in the cache. Additionally, both techniques can increase the static code size and instruction-cache footprint of an application.

[1] Previous work by Saavedra et al. attempted to reduce the number of prologue invocations by merging prologues into earlier epilogues, but found no performance benefits because of increased cache conflicts [28].

These performance limitations may impede these latency-tolerance techniques in environments with low bandwidth or poor instruction memory.

4 Combining Clustering and Prefetching

Software prefetching and read miss clustering seem to target the same types of latencies, and both techniques require the same system resources (e.g., cache miss buffers and memory system bandwidth). However, their latency-tolerance methods are quite distinct, since prefetching pipelines inner loops to add new fetches ahead of their demand accesses and clustering restructures loop nests at an outer-loop level to extract parallelism among demand read misses. This section discusses how applying read miss clustering before software prefetching can address the performance limitations described in Section 3. Figure 1(d) shows the matrix traversal of Figure 1(a) after applying the combination of clustering followed by prefetching.

Incomplete latency hiding. Effective prefetching can tolerate all steady-state latencies, while read miss clustering alone leaves at least some miss latencies exposed. Thus, prefetching can potentially tolerate the steady-state latencies left behind after clustering.

Prologue late prefetches. Read miss clustering can reduce the effect of prologue late prefetches by reducing the number of times an inner-loop is started. Consider a 2-level loop nest with I_o outer loop iterations, such as the code in Figure 1(a). With prefetching alone, the inner loop would be started I_o times, with late prefetches on the first steady-state iteration each time. If unroll-and-jam with a degree of N is applied before prefetching, the outer loop will only have $\frac{I_o}{N}$ iterations, and the inner loop only starts $\frac{I_o}{N}$ times. This reduces the number of separate instances of prologue late prefetches (or other exposed latencies at the beginning of each steady-state) by a factor of N.

Short steady-states. Read miss clustering increases the computation in an inner-loop iteration (the C term of Section 2.2) without changing the number of inner-loop iterations (I_i) (as seen in Figures 1(a) and 1(b)). Since the prefetch distance (in iterations) is inversely proportional to C, clustering can reduce the prefetch distance and increase the length of the steady-state, increasing the effectiveness of prefetching. (This implies that d' in Figure 1(d) is smaller than d in Figure 1(c).)

Hard-to-prefetch references. The memory parallelism benefits of read miss clustering also apply to references prefetched with a prefetch distance insufficient to overlap their latency fully, or for unprefetched misses.

Instruction overhead. Read miss clustering through unroll-and-jam can exploit *scalar replacement*, which replaces redundant memory references with register operations and reduces the instruction count [2, 5, 7]. If the redundant references tended to hit in the cache (as seen in previous work [22]), scalar replacement can reduce both the unnecessary prefetches resulting from these references and their address-generation overhead.

Legality limitations of clustering. Unlike clustering, software prefetching has no legality constraints caused by dependences. Software prefetching can thus tolerate latencies in portions of an application where dependences prevent the use of clustering transformations.

Other limitations. The memory parallelism provided by read miss clustering can help to tolerate any latencies exposed by early or damaging prefetches, but may also increase early or damaging prefetches by keeping more lines active in the cache at once.

Trends. We expect D (the miss latency in cycles) to increase as processor clock speeds improve faster than memory latencies. The number of cycles per iteration, C, is likely to decrease with more aggressive processor architectures. Both hardware trends increase the prefetch distance. Additionally, software that more aggressively uses locality transformations such as tiling sees shorter inner loops with each inner loop initiated more times [1, 25, 32]. These hardware and software trends increase the impact of prologue late prefetches, short steady-states, and hard-to-prefetch references, all of which can be addressed by read miss clustering. On the other hand, we expect the impact of prefetching instruction overhead to be less important as processor speeds increase in the future. For data-intensive applications, this trend seems less substantial than the other latency-related trends.

5 Methodology

5.1 Evaluation Environments

We perform our experiments both in simulation and on current hardware, studying both multiprocessor and uniprocessor systems. The simulation results enable more detailed analysis, and we primarily focus on these results. We use the RSIM simulator to model an aggressive ILP-based uniprocessor and a CC-NUMA multiprocessor with the release consistency memory model and a directory-based cache coherence protocol [24]. Table 1 summarizes the base system parameters for our simulation study. The cache sizes are scaled according to the application input sizes, following the methodology of Woo et al. [33].

We also perform experiments on a Convex Exemplar SPP-2000 with 180 MHz HP PA-8000 processors [12, 13]. Each processor has 4-way issue, a 56-entry out-of-order instruction window, and a 1 MB single-level data cache with 32-byte lines and up to 10 simultaneous misses outstanding. The Exemplar supports a CC-NUMA configuration using SMP hypernodes of up to 16 processors. We perform our experiments within a hypernode, treating the machine

Processor parameters	
Clock rate	500 MHz
Fetch rate	4 instructions/cycle
Instruction window	64 instructions in-flight
Memory queue size	32
Outstanding branches	16
Functional units	2 ALUs, 2 FPUs, 2 address units
Instruction latencies (cycles)	1 (addr. gen., most ALU), 3 (most FPU), 7 (int. mult./div.), 16 (FP div.), 33 (FP sqrt.)
Memory hierarchy and network parameters	
L1 D-cache	16 KB, direct-mapped, 2 ports, 10 MSHRs, 64-byte line
L1 I-cache	16 KB, direct-mapped, 64-byte line
L2 cache	64 KB (for Erlebacher, FFT, and LU) or 1 MB (for Em3d, MST, and Ocean), 4-way associative, 1 port, 10 MSHRs, 64-byte line, pipelined
Memory banks	4-way, permutation interleaving
Bus	167 MHz, 256 bits, split transaction
Network	2D mesh, 250MHz, 64 bits, flit delay of 2 network cycles per hop

Table 1. Base simulated configuration.

as an SMP and avoiding any data-placement issues. Our explicitly parallel applications use the `pthreads` library.

5.2 Evaluation Workload

We perform our evaluations with six applications. Table 2 summarizes the experimental applications and data sets used on the simulated and real systems. We limit the number of processors for each application according to the scalability of the application and the limits of the system (16 in simulation, 8 on the Exemplar). Each application is compiled with the Sun SPARC SC4.2 compiler for the simulation and the Exemplar C compiler for the real machine, with the optimization level `-xO4` for the Sun compiler and `+O3` for the Exemplar. We add prefetching by hand for the simulated system, following the algorithms described in Section 2.2. Since the Exemplar C compiler supports prefetching, we use compiler-generated prefetching for the real machine [29]. The Exemplar compiler could prefetch all test programs except MST. We add read miss clustering by hand, following the algorithm for analysis and transformation reported in previous work [22]. Since prefetching has already been implemented in compilers, and miss clustering is based on a transformation that has been implemented in compilers (unroll-and-jam), we are confident that the combined technique can be effectively implemented in a real compilation system.

Four of our applications are regular. These applications are all array-based and only use indices that are affine functions of the controlling loop variables. Erlebacher is a shared-memory port of a program by Thomas Eidson at ICASE, and FFT, LU, and Ocean are from the SPLASH-2 application suite [33]. For better load balance, LU is modified slightly to use flags instead of barriers. We also apply an additional transformation which combines loop interchange and outer-loop prefetching to tolerate latencies in the postludes left behind by unroll-and-jam [21]. Although general outer-loop prefetching can increase early prefetches and conflict or capacity misses, the short inner-loops of the interchanged postludes make such problems unlikely.

We also consider two irregular applications, Em3d and MST. Em3d is a shared-memory port of a Split-C benchmark [10] and uses indirection through affine array references. MST is the minimal-spanning tree algorithm from the Olden benchmarks [26]. MST uses linked data structures and is dominated by linked-list traversals for lookups in a hash-table. In both of these applications, the inner loop copies being fused in unroll-and-jam have different lengths, so only the minimum length seen across the unrolled copies is fused. Then, each copy completes its remaining length separately. We do not run MST on the multiprocessor because of excessive synchronization overhead, and we do not run MST at all on the real machine because the Exemplar compiler does not prefetch its list traversals. For our simulation study, we prefetch MST with both greedy prefetching and prefetch arrays [15, 16]. We limit the prefetch array length to 4 because typical linked-list lengths for MST were found to be 2–4 [15]. Because the lists are so short, we also do not add artificial jump pointers to the list elements.

5.3 Evaluation Metrics

Our key metric is total execution time. For detailed analysis, we categorize simulated execution time as follows: data read miss stall, data read hit or write stall (usually seen only in the event of high resource contention), CPU (busy time and functional-unit stalls), synchronization, and instruction memory stall times. We count stall cycles as follows, similar to previous work (e.g., [16, 23]). For each cycle, we calculate the ratio of the instructions retired from the instruction window to the maximum retire rate and attribute this fraction of that cycle to busy time. The rest of the cycle is attributed as stall time to the first instruction that could not retire that cycle, or as instruction memory stall if no instruction is available in the window because of an I-cache stall. We also gain insights by counting late prefetches, considering only those for which a demand reference exposes data read miss stall time.

6 Experimental Results

Section 6.1 compares and combines clustering and prefetching in simulation, and Section 6.2 confirms the benefits of these techniques on a real system.

	Simulated system		Convex Exemplar	
Application	**Input Size**	**Procs.**	**Input Size**	**Procs.**
Em3d	32K nodes, deg. 20, 20% rem.	1,16	100K nodes, deg. 20, 20% rem.	1,8
Erlebacher	64x64x64 cube, block 8	1,16	128x128x128 cube, block 8	1,8
FFT	65536 points	1,16	4M points	1,8
LU	256x256 matrix, block 16	1,8	4224x4224 matrix, block 128	1,8
MST	1024 nodes	1	N/A	N/A
Ocean	258x258 grid	1,8	1026x1026 grid	1,8

Table 2. Application input sizes and number of processors for simulation and real system. MST is not included for the real machine because the Exemplar compiler adds no prefetches for it.

6.1 Simulation Results

Figures 3(a) and (b) shows the multiprocessor and uniprocessor execution times of the applications running on the base simulated system. The execution time bars show the original code (Base/noPF), the code after prefetching alone (Base/+PF), after clustering alone (Clust/noPF), and after the combination of the two (Clust/+PF). All execution-time bars are split as described in Section 5.3 and normalized to the execution time with neither prefetching nor clustering. (For MST, this chart only shows prefetch arrays. Greedy prefetching is described later.)

6.1.1 Comparing Clustering and Prefetching

We focus here on the first three bars of Figure 3 for each application and system configuration (i.e., base code, prefetching alone, and clustering alone). Section 6.1.2 discusses the fourth bar, which combines clustering and prefetching. (Previous work has already covered each scheme in isolation [22, 23].)

Comparing clustering alone to prefetching alone, we see that clustering gives comparable or better overall execution times than prefetching for all applications except the uniprocessor Ocean. In the multiprocessor, clustering reduces execution time an average of 20% (ranging 5–39%), while prefetching reduces execution time an average of 17% for 3 out of 5 applications (ranging 9–30%, with less than 5% degradation on the other 2 codes). In the uniprocessor, clustering reduces execution time an average of 30% (ranging 5–39%), while prefetching reduces execution time an average of 17% (ranging 1–35%).

To understand the differences between clustering and prefetching, we consider the individual components of execution time. Prefetching alone actually has a greater impact on data memory stall time for all applications except LU and MST. However, the instruction overhead of prefetching (discussed in Section 3) increases CPU time and offsets the greater memory stall time for all cases except the uniprocessor Ocean. This CPU overhead actually leads to slight performance degradations *relative to the base code* in the multiprocessor Em3d and Ocean. Additionally, clustering alone actually sees reductions in the CPU component of execution time for many of the applications because of the scalar replacement benefits of unroll-and-jam (discussed in Section 4). Both latency tolerance techniques have little negative impact on instruction memory stalls, since these loop-based applications still tend to hit in the I-cache.

In MST, prefetch arrays provide substantial benefits but also increase the needed working set and cause new misses. These new misses cannot be prefetched well because the index into the array is calculated through a hash function just before the traversal, and the arrays are too short to allow prefetching of the remaining elements. Additionally, the prefetch arrays always fetch several items of the list being traversed, even though a hash match might arise within the first 1 or 2 list entries. Thus, the remaining prefetches are useless and increase overhead without tolerating any latency. Other linked-data structure prefetching schemes such as greedy prefetching do not increase the needed working set. Figure 4 includes results with greedy prefetching as well, with GPF and PFA denoting greedy prefetching and prefetch arrays, respectively. Greedy prefetching suffers from hard-to-prefetch references, as the prefetch distance for its linked-list traversals is limited to 1 iteration, and each iteration has very little computation. On the other hand, clustering alone tolerates latencies more effectively by restructuring the demand references at an outer-loop level so that multiple lists are traversed in parallel. Prefetching schemes that use longer artificial jump pointers would be inapplicable, since the linked-lists in MST are very short[2].

6.1.2 Combination of Clustering and Prefetching

Overall results. The fourth bar of each application and system configuration in Figure 3 combines clustering and prefetching. Even though each scheme in isolation tolerates substantial latencies, the combination allows additional benefits in many cases. Except for the uniprocessor Em3d and MST, this combination either performs the best or sees less than 1% degradation from the best. Compared to clustering alone, the combination reduces execution time an av-

[2] *Root jumping* is a linked-data structure prefetching technique motivated by short linked lists in which prefetches are issued for a later list traversal in lockstep with the current list traversal. Because list lengths are not known in advance in MST, the ideal prefetch distance cannot be determined. In [27], only the very next list traversal is prefetched (in lockstep with the current traversal), thereby limiting parallelism.

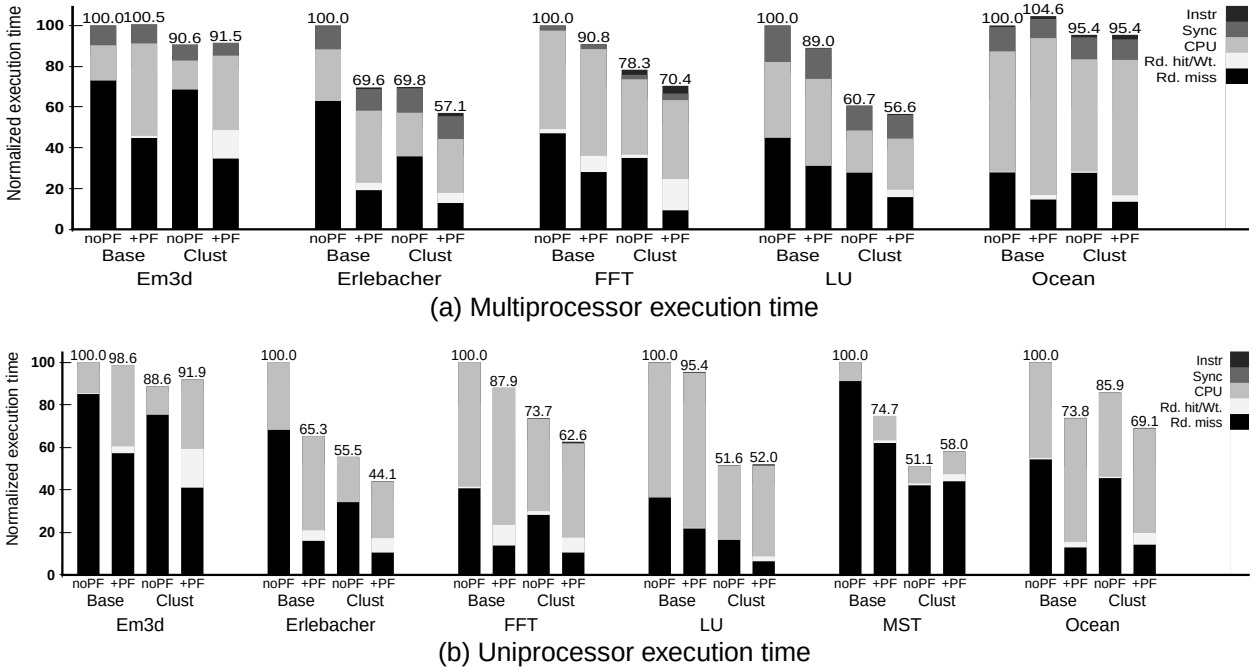


Figure 3. Execution times on base simulated system with software prefetching, clustering, and their combination. All times are shown normalized to the execution time with neither technique.

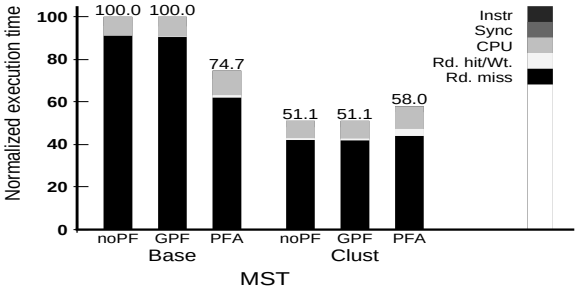


Figure 4. Greedy prefetching and prefetch arrays for MST on base simulated uniprocessor. Execution times are normalized to the unclustered code without prefetching.

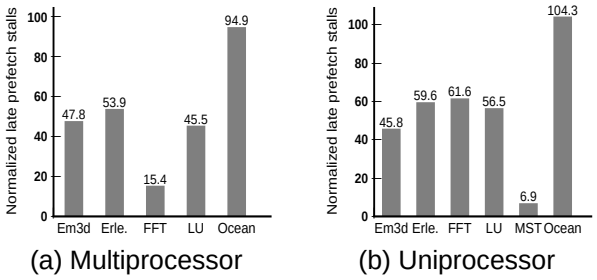


Figure 5. Number of late prefetch stalls after clustered prefetching represented as a percentage of the late prefetch stalls seen with prefetching alone.

erage of 12% for 3 out of 5 applications in the multiprocessor (ranging 7–18%, with a 1% degradation in 1 code and no impact on another) and 18% for 3 of 6 uniprocessor codes (ranging 15–21%, with less than 5% degradation in 2 codes and 14% degradation in MST). Compared to prefetching alone, the combination reduces execution time an average of 18% in the multiprocessor (ranging 9–36%) and 24% in the uniprocessor (ranging 6–49%). We compute these averages conservatively by comparing clustered prefetching against the best of either the base code or the code with one optimization alone (for cases where one optimization degrades performance).

Figure 5 shows the impact of clustering on the number of late prefetches that lead to stalls. All bars show the number of late prefetch stalls for clustered prefetching as a percentage of the number for prefetching alone. Figures 5(a) and 5(b) show multiprocessor and uniprocessor systems, respectively. Read miss clustering reduces the number of late prefetch stalls by an average of 49% on the multiprocessor and 44% on the uniprocessor, with dramatic improvements in all cases except Ocean.

Compared to clustering alone, some of the improvements seen with clustered prefetching are negated by the CPU overhead of prefetching. Additionally, an increase in read hit and write time (from increased contention) also degrades the performance of some applications. On the other hand, the scalar replacement benefits of clustering sometimes re-

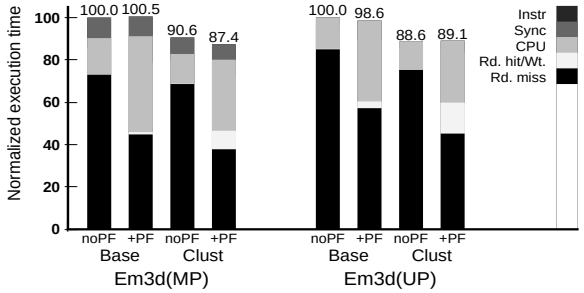


Figure 6. Execution times of Em3d on base system when less clustering is used in combination with prefetching.

duce the CPU overhead of prefetching and the number of unnecessary prefetches for some applications.

Application-specific details. In Em3d, clustering improves prefetching by overlapping prologue late prefetches and by increasing the otherwise small steady-state sizes (caused by short loops and indirect prefetching), reducing late prefetch stalls over 50%. Prefetching tolerates read miss latencies remaining after clustering alone. However, clustered prefetching sees stalls from exposed read hits. A closer examination shows register spilling when clustering and prefetching are applied together, though neither one alone causes spills. These spills tend to hit in the cache, but increase contention for cache ports. Previous work on unroll-and-jam suggests limiting the unrolling degree based on register pressure [7]. An unroll-and-jam algorithm suitable for combining with prefetching may profit from heuristics that consider the additional register pressure caused by prefetching. To demonstrate the potential effectiveness of using less clustering in combination with prefetching, Figure 6 shows the performance of Em3d with a smaller degree of unroll-and-jam in the Clust/+PF run (4, instead of the 6 used in Clust/noPF). The resulting reduction in register spills and read hit stall time improves total execution time and allows benefits for clustered prefetching in the multiprocessor.

Erlebacher, FFT, and LU all have important phases blocked for cache locality and/or load balance: the fine-grained wavefront pipeline in Erlebacher, the transpose in FFT, and the entire code of LU. None of these blocked portions achieves a steady-state with prefetching alone. Clustering actually enables a steady state for the transpose of FFT, reducing the number of late prefetch stalls by 85% for the multiprocessor. Clustering also overlaps prologue late prefetches in all three applications. Erlebacher also sees some benefits from scalar replacement of references that tend to cause unnecessary prefetches, while scalar replacement in LU substantially reduces the total instruction count. In Erlebacher and FFT, prefetching helps to tolerate steady-state latencies left behind by clustering in other phases of the application. These three applications benefit significantly from clustered prefetching, with substantial benefits relative to prefetching alone in all cases and relative to clustering alone in all but the uniprocessor LU. (The incremental latency tolerance of prefetching in the uniprocessor LU is not sufficient to offset its CPU overhead, leading to a slight degradation relative to clustering alone.)

For MST, prefetch arrays improve the unclustered version, but degrade the performance of the clustered code. In particular, clustering leaves less available bandwidth for the extra fetches added by prefetch arrays, exposing the negative effects of this scheme's increased working set and new misses. Greedy prefetching avoids such degradations, but also provides no benefits over clustering alone because of its limited prefetch distance. Thus, clustering alone provides the best performance in MST.

Ocean sees an increase in conflict misses from clustering. Combined with prefetching, this causes additional contention-related stalls and early prefetches. Additionally, these conflicts also turn some unnecessary prefetches into necessary ones, increasing the number of late prefetch stalls in the uniprocessor and causing clustered prefetching to see more data memory stall time than prefetching alone. However, scalar replacement provides some benefits in Ocean by reducing unnecessary prefetches and the CPU overhead of prefetching. The net effect is that clustered prefetching provides the best overall execution time.

6.1.3 Sensitivity to system parameters

Processor speeds and external memory latencies diverge further for processors in the gigahertz frequency range. To model this trend, we also performed experiments that model 1 GHz processors without changing any absolute memory hierarchy times (in ns or MHz).

The results in Figure 7 show behavior qualitatively similar to Figure 3. (Em3d is shown with reduced clustering in combination with prefetching. MST is still shown with prefetch arrays; greedy prefetching again had negligible impact.) As expected, CPU overhead becomes less important, reducing prefetching overhead in both the base and clustered versions. Prefetching alone now outperforms clustering alone in the multiprocessor Erlebacher and the uniprocessor Ocean, but clustered prefetching remains the best for all but the uniprocessor MST. Clustered prefetching reduces execution time relative to prefetching alone an average of 21% across all cases, and reduces execution time relative to clustering alone an average of 13% in all but the uniprocessor MST. These results are similar to the base configuration, and we expect this trend to continue based on the discussion in Section 4.

6.2 Results on Real Machine

Table 3 gives the impact of prefetching, clustering, and their combination for multiprocessor and uniprocessor ap-

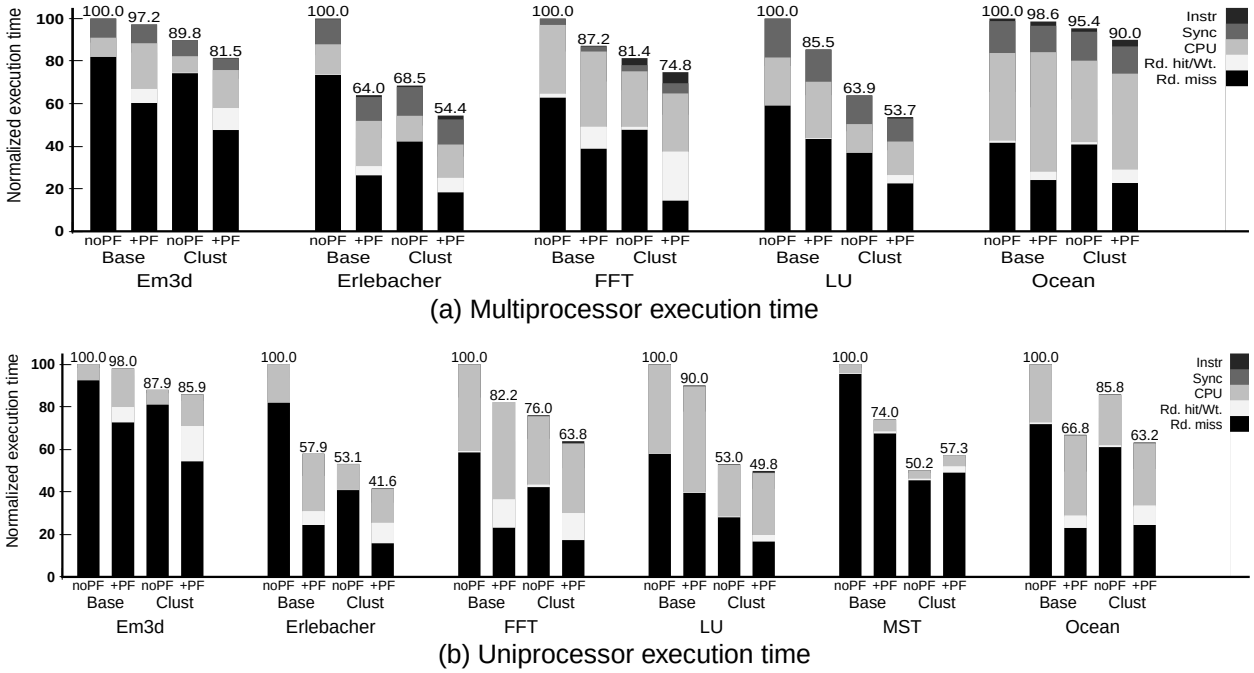


Figure 7. Execution times on simulated system with faster processors.

plications run on the Convex Exemplar. Total execution times are normalized to the code with the same number of processors with neither prefetching nor clustering. (Detailed categorizations are not available on the real machine.) Here, clustering alone outperforms prefetching alone for 6 out of 10 applications and systems shown. Clustered prefetching performs better than both clustering alone and prefetching alone for 5 out of 10 cases (multiprocessor and uniprocessor Erlebacher, multiprocessor and uniprocessor FFT, and uniprocessor Ocean). Compared to prefetching alone, clustered prefetching reduces execution time an average of 14% for 2 out of 5 multiprocessor codes (ranging 11–16%) and 19% for 3 out of 5 uniprocessor codes (ranging 3–29%). Compared to clustering alone, clustered prefetching reduces execution time an average of 21% for 3 out of 5 multiprocessor codes (ranging 11–40%) in the multiprocessor and 20% for 3 out of 5 uniprocessor codes (ranging 4–40%). As in Section 6.1.2, these averages are computed conservatively by comparing against the base code when a code is degraded by its sole optimization. Additionally, we do not count LU since both prefetching and clustered prefetching are greatly degraded here. All the observed degradations are discussed below.

The multiprocessor version of Ocean sees the best performance with prefetching alone, with 8% degradation from clustered prefetching. The simulation results of Section 6.1 had suggested an increase in conflicts with clustering or clustered prefetching. The HP PA-8000 may further exacerbate these problems, as its cache is direct-mapped and must serialize conflicting references [29].

Em3d sees only minor performance differences between prefetching alone, clustering alone, and clustered prefetching. Em3d sees an anomalous benefit from prefetching in the unclustered version due to a detail of the Exemplar C compiler. We observe from assembly-code analysis that this compiler does not generate a prologue or epilogue for prefetched loops. The lack of a prologue may not affect performance, since those prefetches would likely be late (Section 2.2). Without an epilogue, however, the algorithm always fetches extra data beyond the end of the inner loop. In Em3d, these items are actually used in later outer-loop iterations. With clustered prefetching, most of these prefetches become unnecessary and simply add instruction overhead, since misses from later iterations of the original outer-loop would have already been fetched by the restructured code.

Prefetching does not always benefit by omitting the epilogue. In particular, extraneous fetches might not touch useful data and could increase conflict and capacity misses (especially for tiled codes), as well as coherence traffic and apparent data sharing for fine-grained shared-memory multiprocessor codes. Such problems arise in LU, where prefetching dramatically degrades performance. Clustered prefetching can mitigate these negative effects by reducing the prefetch distance, thereby limiting the extraneous data fetched, and by overlapping the resulting capacity or communication misses. Nevertheless, clustering alone significantly outperforms clustered prefetching and prefetching alone for LU. We consider the Em3d and LU results more

	Multiprocessor system				Uniprocessor system			
Application	Base	Base+PF	Clust	Clust+PF	Base	Base+PF	Clust	Clust+PF
Em3d	100.0	89.5	90.8	92.8	100.0	88.5	87.1	90.2
Erlebacher	100.0	76.9	78.5	68.2	100.0	77.5	65.7	54.8
FFT	100.0	88.3	83.4	74.3	100.0	90.5	71.1	68.5
LU	100.0	241.5	77.3	118.6	100.0	219.8	76.3	100.4
Ocean	100.0	56.2	102.9	60.9	100.0	49.3	78.5	47.7

Table 3. Execution times on Convex Exemplar with software prefetching, clustering, and their combination. All times are shown normalized to the execution time with the same number of processors and neither prefetching nor clustering.

indicative of specific compiler issues rather than underlying problems in clustered prefetching.

7 Related Work

Section 2 discusses the previous work that our study uses most directly. Previous prefetching techniques have also provided some memory parallelism among prefetches, but their main focus has been on fetching sufficiently in advance. Roth and Sohi discuss parallelism among prefetches, but they use this largely to facilitate fetching ahead and make no attempt to fully utilize the resources for parallelism [27]. In contrast, read miss clustering restructures the code aiming to fully utilize the miss buffers of the cache, increasing the parallelism achieved by prefetching.

Chen and Baer studied hardware prefetching combined with local instruction scheduling to improve the effectiveness of nonblocking reads [8]. Because the code transformations studied were limited to the basic-block level, their benefits stemmed primarily from overlapping misses with independent computation.

Saavedra et al. observed that tiled codes were more difficult to prefetch [28]. They suggest optimizations to improve cache performance for tiled codes, but do not target the short steady-states. As a result, their analysis generally favors prefetching alone over the combination of prefetching and tiling. We show that using unroll-and-jam for read miss clustering can help to lengthen steady-states, allowing us to maintain the bandwidth benefits of tiling while also improving prefetching effectiveness.

Two works on unroll-and-jam have particular relevance. Carr has considered prefetches and cache misses while calculating the heuristics used when applying unroll-and-jam for scalar replacement or locality [5]. However, that work did not seek to improve prefetching, but instead assumed that prefetching was effective given enough hardware resources. Carr et al. have used unroll-and-jam to improve software pipelining, without considering cache misses [6]. That study would reduce floating-point stalls in the software pipelining prologue and steady-state by improving floating-point parallelism, but would not lengthen the steady-state.

Our previous work suggested negative interactions between clustering and prefetching [23]. However, that work achieved clustering through loop interchange, which can increase resource contention by eliminating inner-loop spatial locality. In contrast, our current study achieves clustering through unroll-and-jam and shows how this technique can actually improve prefetching.

Finally, this work has focused on software latency tolerance techniques. Hardware techniques such as hardware prefetching or multithreading also provide latency tolerance [9, 11, 14, 31]. The interaction of read miss clustering with such hardware techniques remains an open question.

8 Conclusions and Future Work

This work compares and combines two latency-hiding techniques, read miss clustering and software prefetching. For the applications and systems we study, clustering alone outperforms prefetching alone for most cases. This result, however, is sensitive to system trends, and there may be some applications where clustering is not applicable but prefetching is. More interestingly, the combination of read miss clustering and prefetching yields better execution time benefits than either technique alone in most cases, and we expect this trend to continue since each technique can address limitations in the other.

Cases where clustered prefetching falls short can be attributed to a small number of application-dependent causes that are not fundamental limitations of the technique itself (e.g., cache conflicts in Ocean, register pressure in Em3d, compiler anomalies for Em3d and LU on the Exemplar, or insufficient bandwidth for MST with prefetch arrays). Compiler or hardware solutions that target these problems can provide further benefits for clustered prefetching. Other interesting candidates for clustered prefetching research include other data-intensive application domains with tile-structured loop nests, such as some multimedia codes.

Acknowledgments

We thank Vikram Adve, Keith Cooper, Chen Ding, Ken Kennedy, John Mellor-Crummey, Partha Ranganathan, and Willy Zwaenepoel for valuable comments on this work.

References

[1] W. Abu-Sufah, D. J. Kuck, and D. H. Lawrie. On the Performance Enhancement of Paging Systems Through Program Analysis and Transformations. *IEEE Trans. on Computers*, C-30(5):341–356, May 1981.

[2] F. E. Allen and J. Cocke. A Catalogue of Optimizing Transformations. In R. Rustin, editor, *Design and Optimization of Compilers*, pages 1–30. Prentice-Hall, 1972.

[3] D. Callahan, J. Cocke, and K. Kennedy. Estimating Interlock and Improving Balance for Pipelined Machines. *Journal of Parallel and Distributed Computing*, 5(4):334–358, Aug. 1988.

[4] D. Callahan, K. Kennedy, and A. Porterfield. Software Prefetching. In *Proc. of the 4th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, Apr. 1991.

[5] S. Carr. Combining Optimization for Cache and Instruction-Level Parallelism. In *Proc. of the Conf. on Parallel Architectures and Compilation Techniques*, pages 238–247, Oct. 1996.

[6] S. Carr, C. Ding, and P. Sweany. Improving Software Pipelining with Unroll-and-Jam. In *Proceedings of 29th Hawaii International Conference on System Sciences*, Jan. 1996.

[7] S. Carr and K. Kennedy. Improving the Ratio of Memory Operations to Floating-Point Operations in Loops. *ACM Trans. on Programming Languages and Systems*, 16(6):1768–1810, Nov. 1994.

[8] T.-F. Chen and J.-L. Baer. Reducing Memory Latency via Non-blocking and Prefetching Caches. In *Proc. of the 5th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 51–61, Oct. 1992.

[9] T.-F. Chen and J.-L. Baer. A Performance Study of Hardware and Software Data Prefetching Schemes. In *Proc. of the 21st Annual Int'l Symp. on Computer Architecture*, pages 223–232, Apr. 1994.

[10] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proc. of Supercomputing*, pages 262–273, Nov. 1993.

[11] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W.-D. Weber. Comparative Evaluation of Latency Reducing and Tolerating Techniques. In *Proc. of the 18th Annual Int'l Symp. on Computer Architecture*, pages 254–263, May 1991.

[12] Hewlett-Packard Company. *Exemplar Architecture (S and X-Class Servers)*, Jan. 1997.

[13] D. Hunt. Advanced Features of the 64-bit PA-8000. In *Proceedings of IEEE Compcon*, pages 123–128, Mar. 1995.

[14] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proc. of the 17th Annual Int'l Symp. on Computer Architecture*, pages 364–373, May 1990.

[15] M. Karlsson, F. Dahlgren, and P. Stenström. A Prefetching Technique for Irregular Accesses to Linked Data Structures. In *Proc. of the 6th Int'l Symp. on High Performance Computer Architecture*, pages 206–217, Jan. 2000.

[16] C.-K. Luk and T. C. Mowry. Compiler-Based Prefetching for Recursive Data Structures. In *Proc. of the 7th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, Oct. 1996.

[17] T. Mowry. *Tolerating Latency through Software-controlled Data Prefetching*. PhD thesis, Stanford University, 1994.

[18] T. Mowry and A. Gupta. Tolerating Latency Through Software-Controlled Prefetching. *Journal on Parallel and Distributed Computing*, pages 87–106, June 1991.

[19] T. C. Mowry, M. S. Lam, and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proc. of the 5th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, Oct. 1992.

[20] A. Nicolau. Loop Quantization or Unwinding Done Right. In *Proc. of the 1st Int'l Conf. on Supercomputing*, pages 294–308, June 1987.

[21] V. S. Pai. *Exploiting Instruction-Level Parallelism for Memory System Performance*. PhD thesis, Department of Electrical and Computer Engineering, Rice University, Aug. 2000.

[22] V. S. Pai and S. Adve. Code Transformations to Improve Memory Parallelism. In *Proc. of the 32nd Annual Int'l Symposium on Microarchitecture*, pages 147–155, Nov. 1999.

[23] V. S. Pai, P. Ranganathan, H. Abdel-Shafi, and S. Adve. The Impact of Exploiting Instruction-Level Parallelism on Shared-Memory Multiprocessors. *IEEE Trans. on Computers*, 48(2):218–226, Feb. 1999.

[24] V. S. Pai, P. Ranganathan, and S. V. Adve. *RSIM Reference Manual, Version 1.0*. Electrical and Computer Engineering Department, Rice University, Aug. 1997. Technical Report 9705.

[25] A. K. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. PhD thesis, Rice University, Apr. 1989.

[26] A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren. Supporting dynamic data structures on distributed-memory machines. *ACM Trans. on Programming Languages and Systems*, 17(2):233–263, Mar. 1995.

[27] A. Roth and G. S. Sohi. Effective Jump-Pointer Prefetching for Linked Data Structures. In *Proc. of the 26th Annual Int'l Symp. on Computer Architecture*, pages 111–121, May 1999.

[28] R. H. Saavedra, W. Mao, D. Park, J. Chame, and S. Moon. The Combined Effectiveness of Unimodular Transformations, Tiling, and Software Prefetching. In *Proc. of the 10th Intl. Parallel Processing Symp.*, pages 39–45, Apr. 1996.

[29] V. Santhanam, E. H. Gornish, and W.-C. Hsu. Data Prefetching on the HP PA-8000. In *Proc. of the 24th Annual Int'l Symp. on Computer Architecture*, pages 264–273, June 1997.

[30] J. E. Smith and A. R. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Trans. on Computers*, C-37(5):562–573, May 1988.

[31] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proc. of the 23rd Annual Int'l Symp. on Computer Architecture*, pages 191–202, May 1996.

[32] M. E. Wolf and M. S. Lam. A Data Locality Optimizing Algorithm. In *Proc. of the Conf. on Programming Language Design and Implementation*, pages 30–44, June 1991.

[33] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. of the 22nd Annual Int'l Symp. on Computer Architecture*, pages 24–36, June 1995.