Journal of Instruction-Level Parallelism 2 (2000) 1-16 Submitted 2/00; published 5/00

Code Transformations to Improve Memory Parallelism *

Vijay S. Pai VIJAYPAI@ECE.RICE.EDU
Electrical and Computer Engineering – MS 366, Rice University
Houston, TX 77005 USA

Sarita Adve SADVE@CS.UIUC.EDU
Computer Science, University of Illinois
Urbana-Champaign, IL 61801 USA

Abstract

Current microprocessors incorporate techniques to exploit instruction-level parallelism (ILP). However, previous work has shown that these ILP techniques are less effective in removing memory stall time than CPU time, making the memory system a greater bottleneck in ILP-based systems than in previous-generation systems. These deficiencies arise largely because applications present limited opportunities for an out-of-order issue processor to overlap multiple read misses, the dominant source of memory stalls.

This work proposes code transformations to increase parallelism in the memory system by overlapping multiple read misses within the same instruction window, while preserving cache locality. We present an analysis and transformation framework suitable for compiler implementation. Our simulation experiments show execution time reductions averaging 20% in a multiprocessor and 30% in a uniprocessor. A substantial part of these reductions comes from increases in memory parallelism. We see similar benefits on a Convex Exemplar.

Keywords: compiler transformations, out-of-order issue, memory parallelism, latency tolerance, unroll-and-jam

1. Introduction

Current commodity microprocessors improve performance through aggressive techniques to exploit high levels of instruction-level parallelism (ILP). These techniques include multiple instruction issue, out-of-order (dynamic) issue, non-blocking reads, and speculative execution.

Our previous work characterized the effectiveness of ILP processors in a shared-memory multiprocessor [2]. Although ILP techniques successfully and consistently reduced the CPU component of execution time, their impact on the memory (read) stall component was lower and more application-dependent, making read stall time a larger bottleneck in ILP-based multiprocessors than in previous-generation systems. In particular, current and future read miss latencies are too long to overlap with other instruction types. Thus, an ILP processor needs to overlap multiple read misses with each other to hide a significant portion of their latencies. An out-of-order processor can only overlap those reads held together within its instruction window. Independent read misses must therefore be *clustered* together within a single instruction window to effectively hide their latencies (*read miss clustering*). The applications in our study typically did not exhibit much read miss clustering, leading to poor parallelism in the memory system.

∗. 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 Adve is also supported by an Alfred P. Sloan Research Fellowship. Vijay S. Pai was also supported by a Fannie and John Hertz Foundation Fellowship. This paper extends the authors' paper of the same title published in MICRO-32 (Copyright IEEE, November 1999) [1] by including results with more applications and experiments on a real machine.

This paper presents code transformations to improve memory parallelism for systems with out-of-order processors, while preserving cache locality. We exploit code transformations already known and implemented in compilers for other purposes, providing the analysis needed to relate them to memory parallelism. The key transformation we use is unroll-and-jam, which was originally proposed for improving floating-point pipelining and for scalar replacement [3, 4, 5, 6]. We develop an analysis that maps the memory parallelism problem to floating-point pipelining.

We evaluate the clustering transformations applied by hand to a latency-detection microbenchmark and five scientific applications running on simulated and real uniprocessor and multiprocessor systems. These transformations reduce data memory latency stall time by over 80% for the latency-detection microbenchmark. For the scientific applications, the transformations reduce execution time by 5–39% (averaging 20%) in the simulated multiprocessor and 11–49% (averaging 30%) in the simulated uniprocessor. A substantial part of these execution-time reductions arise from improving memory parallelism, particularly as memory stall time becomes more significant. We confirm the benefits of our transformations on a real system (Convex Exemplar), where they reduce application execution time by 9–38% for 6 out of 7 applications.

An alternative latency tolerating technique is software prefetching, which has been shown to be effective for systems built with simple processors [7]. However, prefetching can be less effective in ILP systems because of increased late prefetches and resource contention [2]. We only consider read miss clustering in this work; our ongoing work indicates ways in which clustering transformations can also improve the effectiveness of prefetching [8].

2. Motivation for Read Miss Clustering

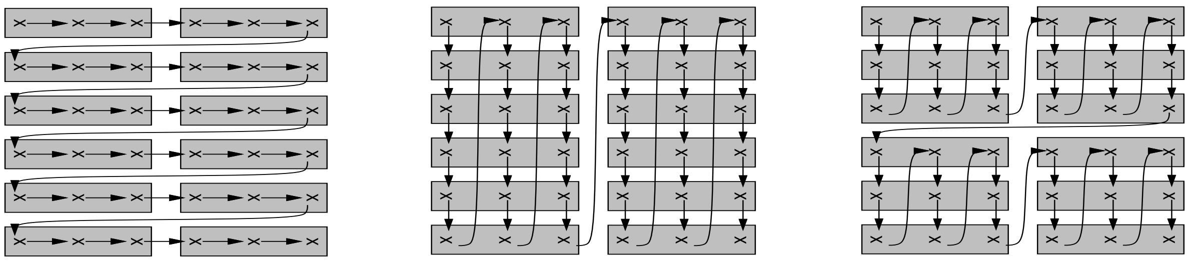
This section discusses the need for read miss clustering, the sources of poor clustering, and code transformations to improve clustering.

2.1 Latency Tolerance in ILP Processors

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 [9]. 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. We refer to this phenomenon as *read miss clustering*, or simply *clustering*.



(a) Exploits locality (b) Exploits miss clustering (c) Exploits both

Figure 1: Impact of matrix traversal order on miss clustering. Crosses represent matrix elements, and shaded blocks represent cache lines. The matrix is shown in row-major order.

```
for(···j++)
 for(···i++)
  ···A[j,i]
```
(a) Base code

```
for(···i++)
 for(···j++)
  ···A[j,i]
```
(b) Interchange

```
for(···jj+=N)
 for(···i++)
  for(j=jj;j<jj+N;j++)
   ···A[j,i]
```
(c) Strip-mine and interchange

```
for(···j+=N)
 for(···i++){

  ···A[j,i]
  ···A[j+1,i]
  ······
  ···A[j+N-1,i]
```
(d) Unroll-and-jam

Figure 2: Pseudocode for Figure 1 matrix traversals (row-major notation).

2.2 Increasing Read Miss Clustering

To understand the sources of poor read miss clustering in typical code, we consider a loop nest traversing a 2-D matrix. Figure 1 graphically represents three different matrix traversals. The matrix is shown in row-major order, with crosses for data elements and shaded blocks for cache lines. Figure 2 relates these matrix traversals to code generation, with pseudocode shown in row-major notation.

Figures 1(a) and 2(a) show a matrix traversal optimized for spatial locality, following much compiler research. In this row-wise traversal, L successive loop iterations access each cache line, where L is the number of data elements per cache line. While this traversal maximizes spatial locality, it minimizes clustering. For example, an instruction window that holds L or fewer iterations never holds read misses to multiple cache lines, preventing clustering. This problem is exacerbated by larger cache lines or larger loop bodies.

Read miss clustering can be maximized by a column-wise traversal, since successive iterations held in the instruction window access different cache lines. Figures 1(b) and 2(b) show such a column-wise traversal, obtained by applying loop interchange to the code in Figure 2(a). Each cache line is now accessed on multiple successive outer-loop iterations. However, the traversal passes through every row before revisiting an older cache line. If there are more rows than cache lines, this traversal could lose all cache locality, potentially overwhelming any performance benefits from clustering.

The above example suggests a tradeoff between spatial locality (favored by current code-generation schemes) and miss clustering. We seek a solution that achieves the benefits of clustering while preserving spatial locality. A column-wise traversal can maximize clustering; however, it must stop before losing locality. In particular, the column-wise traversal can stop as soon as the miss clustering resources are fully utilized. For example, a processor that allows ten simultaneous cache misses sees the maximum memory parallelism when ten independent miss references are clustered. The traversal could then continue in a row-wise fashion to preserve locality. Figure 1(c) shows a matrix traversal that exploits clustering and locality in this way. Figure 2(c) expresses this traversal by applying strip-mine and interchange to Figure 2(a).

Since the column-wise traversal length (N) of Figure 2(c) is based on the hardware resources for overlap (typically 3–12 today), the strip size is small, and the innermost loop can be fully unrolled. Figure 2(d) shows the result of that unrolling. Now, the code reflects the transformation of unroll-and-jam applied to Figure 2(a). This transformation unrolls an outer loop and fuses (jams) the resulting inner loop copies into a single inner loop. Previous work has used unroll-and-jam for scalar replacement (replacing array memory operations with register accesses), better floating-point pipelining, or cache locality [3, 4, 5, 6, 10]. Using unroll-and-jam for read miss clustering requires different heuristics, and may help even when the previously studied benefits are unavailable.

We prefer to use unroll-and-jam instead of strip-mine and interchange for two reasons. First, unroll-and-jam allows us to exploit additional benefits from scalar replacement. Second, unroll-and-jam does not change the inner-loop iteration count. The shorter inner loops of strip-mining can negatively impact techniques that target inner loops, such as dynamic branch prediction. By increasing inner-loop computation without changing the iteration count, unroll-and-jam can also help software prefetching [8].

Unroll-and-jam creates an N-way unrolled steady-state, followed by an untransformed postlude of left-over iterations. To enable clustering in the postlude, we simply interchange the postlude when possible. This should not degrade locality, since the postlude originally has fewer outer-loop iterations than the unroll-and-jam degree.

3. Analysis and Transformation Framework

This section provides a formal framework to apply memory parallelism transformations in a compiler.

3.1 Dependences that Limit Memory Parallelism

We first describe a dependence framework to represent limitations to memory parallelism. As in other domains, dependences here indicate reasons why one operation will not issue in parallel with another. However, these dependences are not ordinary data dependences, since memory operations can be serialized for different reasons. We build this framework to gauge performance potential, not to specify legality. Thus, we optimistically estimate memory parallelism and specify dependences only when their presence is known. The transformation stages must then use more conventional (and conservative) dependence analysis for legality. For simplicity, we only consider memory parallelism dependences that are either loop-independent or carried on the innermost loop. We can then exploit previous work with the same simplification [4].

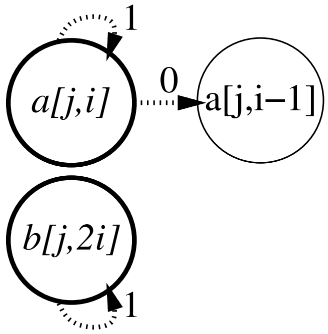
Since we focus on parallelism among read misses, we first require locality analysis to determine which static references can miss in the external cache (*leading references*), and which leading references are known to exhibit spatial locality across successive iterations of the innermost loop (*inner-loop self-spatial locality*). Known locality analysis techniques can provide the needed information [11]. Currently, we do not consider cache conflicts in our analysis and transformations.

We use the above information to identify limitations to read miss parallelism. We focus on three kinds of limitations, which we call *cache-line dependences*, *address dependences*, and *window constraints*.

Cache-line dependences. If a read miss is outstanding, then another reference to the same cache line simply coalesces with the outstanding miss, adding no read miss parallelism. Thus, we say that there is a *cache-line dependence* from memory operation `A` to `B` if `A` is a leading reference and a miss on `A` brings in the data of B. The cache-line dependence is a new resource dependence class, extending input dependences to support multi-word cache lines.

The following code illustrates cache-line dependences. In all examples, leading references known to have inner-loop self-spatial locality will be italicized, while other leading references will be boldfaced. The accompanying graph shows static memory references as nodes and dependences as edges. Each edge is marked with the inner-loop dependence distance, the minimum number of inner-loop iterations separating the dependent operations specified.

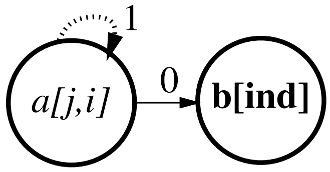
```
for(···j++)
 for(···i++)
  b[j,2*i] = b[j,2*i] + a[j,i] + a[j,i-1]
```



Note that there are no cache-line dependences from one leading reference to another; such a dependence would make the second node a non-leading reference. Additionally, any leading reference with inner-loop self-spatial locality has a cache-line dependence onto itself. That dependence has distance 1 for any stride, since the address of the miss reference will be closer to the instance 1 iteration later than to an instance farther away.

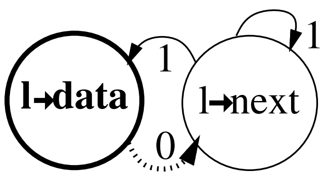
Address dependences. There is an address dependence from memory operation A to B if the result of A is used in computing the address of B, serializing B behind A. Address dependences typically arise for irregular accesses, such as indirect addressing or pointer-chasing. The following code segments show address dependences. The graphs show address dependences as solid lines and cache-line dependences as dotted lines. The first example shows the indirect addressing typical of sparse-matrix applications.

```
for(···j++)
 for(···i++){
  ind = a[j,i]
  sum[j] = sum[j] + b[ind]
```



The above shows one leading reference that exhibits cache-line dependences, connected through an address dependence to another leading reference. The following code shows address dependences from pointer dereferencing.

```
for(···i++){
 l = list[i]
 for(···l=l→next)
  sum[i] += l→data
```



The above assumes that the data and next fields always lie on the same cache line and that separate instances of l are not known to share cache lines. Even though l→next is a non-leading reference, it is important since a dependence flows from this node to the leading reference.

Window constraints. Even without other dependences, read miss parallelism is limited to the number of independent read misses in the loop iterations simultaneously held in the instruction window. We do not include these resource limitations in our dependence graphs, since they can change at each stage of transformation. We will, however, consider these constraints in our transformations.

Control-flow and memory consistency requirements may also restrict read miss parallelism. We do not consider these constraints, since their performance impact can be mitigated through well-known static or dynamic techniques such as speculation. However, these dependences may still affect the legality of any code transformations.

Of the three dependence classes that we consider (cache-line, address, and window), only address dependences are true data-flow dependences. Window constraints can be eliminated through careful loop-body scheduling, possibly enhanced by inner-loop unrolling. Such scheduling would aim to cluster together misses spread over a long loop body. Loop-carried cache line dependences can be made loop-invariant through inner-loop unrolling by a multiple of L, where L iterations share each cache line. Then, no cache line is shared across unrolled loop iterations. However, the inner-loop unrolling degree may need to go as high as $N \times L$ to provide clustered misses to N separate cache lines. This can be excessive, particularly with long cache lines. We therefore leave these loop-carried cache-line dependences in place and seek to extract read miss parallelism with less code expansion through outer-loop unroll-and-jam.

We will address memory parallelism limitations in loop nests by first resolving recurrences (cycles in the dependence graph), and then handling window constraints. A loop nest may suffer from one or both problems, and recurrence resolution may create new window constraints.

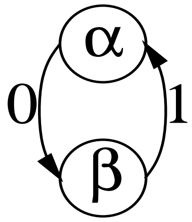
3.2 Resolving Memory Parallelism Recurrences

Unroll-and-jam has previously been used to improve floating-point pipelining in the presence of inner-loop floating-point recurrences [4, 6]. We seek to use unroll-and-jam to target loop nests with memory-parallelism recurrences, which arise for such common access patterns as self-spatial or pointer-chasing leading references. We map memory parallelism to floating-point pipelining, exposing several key similarities and differences between these problems. This section thus shows how to automate the process described in Section 2.2, which used unroll-and-jam to increase miss clustering without degrading locality.

3.2.1 Background on Floating-point Pipelining

Consider an inner loop that carries a floating-point recurrence (a cycle of true dependences). The operations of later iterations can stall for the results of earlier iterations, preventing maximum pipeline throughput. Further, inner-loop unrolling and scheduling cannot help, as later inner-loop iterations are also in the cycle. The following pseudocode has an inner-loop recurrence between statements α and β. The graph shows floating-point true dependences and dependence distances.

```
for(···j++)
 for(···i++){
  α: b[j,i] = a[j,i-1] + c[i]
  β: a[j,i] = b[j,i] + d[i]
```



The above recurrence has two floating-point operations, and needs 1 iteration for a complete cycle (the sum of the dependence distances). Thus, the system must serialize 2 floating-point operations (the number in the recurrence) to complete 1 iteration (the length of the cycle), regardless of the pipelining supported. Callahan et al. described floating-point recurrences as follows [4][1]:

- l_p : number of stages in the floating-point pipeline
- ρ : ratio of the number of nodes (static floating-point operations) in the inner-loop recurrence (R) to the number of iterations to traverse the cycle (τ)

1. Their notation was slightly different, with $\#R$, $\tau(R)$, $\rho(R)$, and f_L instead of R, τ, ρ, and f, respectively.

- f : static count of floating-point operations in an innermost loop iteration

Since R floating point operations must be serialized in τ iterations, the recurrence requires at least ρ pipeline latencies ($= l_p \times \rho$ pipeline stages) per iteration. Without dependences, each iteration would require only the time of f pipeline stages. Thus, the recurrence limits pipeline utilization to $\frac{f}{l_p \times \rho}$. Unroll-and-jam introduces independent copies of the recurrence from separate outer loop iterations, increasing f without affecting ρ [4]. To fill the pipeline, unroll-and-jam must be applied until $f \geq l_p \times \rho$. (The maximum ρ should be used for a loop with multiple recurrences, since each recurrence limits pipeline utilization.)

Certain dependences can prevent unroll-and-jam, but they are not directly related to the recurrences targeted. Previous work more thoroughly discusses legality and the choice of outer loops to unroll for deeper nests [4, 5, 6].

3.2.2 MAPPING TO MEMORY PARALLELISM

Above, unroll-and-jam used only the number of pipeline stages, not the latency. The number of pipeline stages simply represents the number of floating-point operations that can be processed in parallel. Thus, we can map this algorithm to memory parallelism: the goal is to fully utilize the miss clustering resources, not to schedule for some specific miss latencies. Here, l_p corresponds to the maximum number of simultaneous outstanding misses supported by the processor. The rest of the mapping is more difficult, as not all memory operations utilize the resources for miss parallelism — only those instances of leading references that miss at run-time do. This difference affects ρ and f.

Characterizing recurrences (ρ). We refer to recurrences with only cache-line dependences as *cache-line recurrences* and recurrences with at least one address dependence as *address recurrences*. Recurrences with no leading miss references are irrelevant here and can be ignored, since they do not impact read miss parallelism.

As discussed in Section 3.2.1, ρ is computed from two values: R and τ. We count only leading references in R, as only these nodes can lead to serialization for a miss. We count τ as in Section 3.2.1, since this specifies the number of iterations over which all the miss instances in a recurrence are serialized. Our discussion has focused on tolerating only read miss latencies, since write latencies are typically hidden through write buffering. However, our algorithm must count both read and write miss references in R and f. This is because cache resources that determine the maximum number of outstanding misses (e.g., miss status holding registers) are typically shared between reads and writes[2]. Nevertheless, we will not apply unroll-and-jam on an outer loop if it only adds write misses, since write-buffering is typically sufficient to hide their latencies.

Counting memory parallelism candidates (f). For floating-point pipelines, the f parameter counts the static instructions in an innermost loop iteration. We cannot use this same definition here for two key reasons, described below.

Dynamic inner-loop unrolling. An out-of-order instruction window of W instructions dynamically unrolls a loop body of i instructions by $\lceil \frac{W}{i} \rceil$. (For simplicity, we assume no outer-loop unrolling, although this could arise if the inner loop had fewer than $\lceil \frac{W}{i} \rceil$ iterations.) Such unrolling exposes no additional steady-state parallelism for loops with address recurrences, since these are analogous to the recurrences of floating-point pipelining. However, this unrolling can actually break cache-line recurrences. In particular, if L_m successive iterations share a cache line for leading reference m, dynamic inner-loop unrolling creates $\lceil \frac{W}{iL_m} \rceil$ independent misses from the original recurrence. Leading references outside recurrences can also contribute multiple outstanding misses ($L_m = 1$, since no cache-line sharing is known). Thus, we define

2. This is not true of no-write-allocate caches. However, we are most concerned with lower levels of the cache hierarchy that incur long latency misses. Such caches are usually write-allocate.

C_m, the number of copies of m that can contribute overlapped misses:

$$C_m = \begin{cases} \lceil \frac{W}{iL_m} \rceil & \text{loop with no address recurrences} \\ 1 & \text{otherwise} \end{cases} \tag{1}$$

Miss patterns. A simple count of leading references can overestimate memory parallelism, since not all leading reference instances miss in the cache. To determine which leading reference instances miss together, we must know the miss patterns (sequences of hits and misses) for the different leading references and their correlation with each other. Such measures can be difficult to determine in general. In this work, we make some simple assumptions, described below.

We split the leading references into two types: regular (arrays indexed with affine functions of the loop indices) and irregular (all others). For regular references, we assume that at least some passes through the inner loop experience misses on each cache line accessed, and that different regular leading references experience misses together. These assumptions lead to maximum estimated parallelism for regular leading references.

For irregular accesses, the miss pattern is not typically analyzable. We assume no correlation, either among instances of the same reference or across multiple references. Thus, we only need to know the overall miss rate, P_m, for each reference m. P_m can be measured through cache simulation or profiling. These assumptions allow more aggressive transformation than the more common assumption of no locality for irregulars.

We can now estimate the f parameter, accounting for both dynamic inner-loop unrolling and miss patterns:

$$f = f_{reg} + f_{irreg} \tag{2}$$

$$f_{reg} = \sum_{m \in RLR} C_m \tag{3}$$

$$f_{irreg} = \lceil \sum_{m \in ILR} P_m \times C_m \rceil \tag{4}$$

We split f into regular and irregular components, with RLR and ILR the sets of regular and irregular leading references respectively. The terms C_m in Equation 3 and $P_m \times C_m$ in Equation 4 give the maximum expected number of overlapped misses to separate cache lines contributed by leading reference m. We round up f_{irreg} to insure that some resources are held for irregular references when they are present.

The floating-point pipelining algorithm applied unroll-and-jam until $f \geq l_p \times \rho$. We should be more conservative for memory parallelism, as the cache can see extra contention when the resources for outstanding misses (MSHRs) fill up. Thus, we aim to apply unroll-and-jam as much as possible while maintaining $f \leq l_p \times \rho$ (using the maximum ρ for the loop).

After applying unroll-and-jam, we must recompute f for two reasons. First, unroll-and-jam can introduce new leading references and increase the iteration size. On the other hand, some leading reference copies may become non-leading references because of scalar replacement or group locality. For similar reasons, we must repeat the locality and dependence analysis passes.

Since f varies as described above, we may need to attempt unroll-and-jam multiple times with different unrolling degrees to reach our desired f. We can limit the number of invocations by choosing a maximum unrolling degree U based on the resources for memory parallelism, code expansion, register pressure, and potential for cache conflicts. If we unroll only one outer loop, we can choose the unrolling degree by binary search, using at most $\lceil \log_2 U \rceil$ passes [5]. Generalized searching for unrolling multiple outer loops can follow the strategies described in previous work [5]. We also refer to previous work for legality issues [4, 5, 6]. We add only that we prefer not to unroll-and-jam loops that only expose additional write miss references, since buffering can hide write latencies.

To revisit the motivating example of Section 2.2, note that the matrix traversal of Figure 2(a) has a cache-line recurrence with $\rho = 1$. Since modern caches typically have lines of length 32 to 128 bytes, L_m typically ranges from 4 to 16 for stride-1 double-word accesses. With current instruction window sizes ranging from 32 to 80, $\lceil \frac{W}{iL_m} \rceil$ is thus most likely to be 1 for a loop body with a moderate amount of computation. Thus, $f = f_{reg} = 1$ for the initial version of this loop. This example has no scalar replacement opportunities, so each recurrence copy created by unroll-and-jam contributes a leading reference to the calculation of f. Assuming U is chosen to be at least l_p, the search algorithm will find that unroll-and-jam by l_p leads to $f = l_p \times \rho$. Since $\rho = 1$ in this example, the final version of this loop will see l_p static leading references in an iteration.

3.3 Resolving Window Constraints

We now address memory parallelism limitations from window constraints. These can arise for loops with or without recurrences. Further, recurrence resolution can actually create new window constraints, since unroll-and-jam can spread its candidates for read miss parallelism over a span of instructions larger than a single instruction window. We proceed in two stages: first using loop unrolling to resolve any inter-iteration window constraints, then using local instruction scheduling to resolve intra-iteration constraints.

As discussed in Section 3.2.2, an instruction window of W instructions dynamically unrolls an inner-loop body of i instructions by $\lceil \frac{W}{i} \rceil$. Inter-iteration window constraints arise when the independent read misses in $\lceil \frac{W}{i} \rceil$ iterations do not fill the resources for memory parallelism (typically because of large loop bodies). Since any recurrences have already been resolved, we can now use inner-loop unrolling to better expose independent misses to the instruction scheduler. We can directly count the maximum expected number of independent misses in $\lceil \frac{W}{i} \rceil$ iterations, using the miss rate P_m to weight the irregular leading references. We then unroll until the resources for memory parallelism are filled, recomputing the exposed independent miss count after each invocation of unrolling.

Now we resolve any intra-iteration window constraints stemming from loop bodies larger than a single instruction window (possibly because of unroll-and-jam or inner-loop unrolling). In such cases, the instruction scheduler should pack independent miss references in the loop body close to each other. The technique of balanced scheduling can provide some of these benefits [12, 13], but may also miss some opportunities since it does not explicitly consider window size. Nevertheless, this heuristic worked well for the code sequences we examined. More appropriate local scheduling algorithms remain the subject of future research.

4. Experimental Methodology

4.1 Evaluation Environments

We perform most of our experiments using RSIM, the Rice Simulator for ILP Multiprocessors [14]. We focus on simulation results for detailed analysis and to model future-generation systems. We model both an ILP uniprocessor and an ILP-based CC-NUMA multiprocessor with release consistency. Table 1 summarizes the base simulated configuration. The cache sizes are scaled based on application input sizes according to the methodology of Woo et al. [15] [3]. The memory banks use permutation-based interleaving on a cache-line granularity to support a variety of strides [16]. The simulated system latencies without contention are 1 cycle for L1 hits, 10 cycles for L2 hits, 85 cycles for local memory, 180–260 cycles for remote memory, and 210–310 cycles for cache-to-cache transfers.

3. Realistic application inputs would typically require an impractical amount of simulation time. However, smaller input sizes may lead to overly optimistic views of cache performance. Thus, this methodology suggests scaling the cache sizes down according to the simulated working sets, and is widely used in multiprocessor architectural studies.

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 unit count	2 ALUs, 2 FPUs, 2 address units
Functional unit 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, LU, and Mp3d) 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.

We also confirm the benefits of our transformations on a Convex Exemplar SPP-2000 system with 180 MHz HP PA-8000 processors [17, 18]. 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 system's memory banks support skewed interleaving with a cache-line granularity [19]. Although the Exemplar supports a CC-NUMA configuration with SMP hypernodes, we perform our tests within a hypernode. Thus, we use the Exemplar as an SMP and do not consider data placement issues. We use the `pthreads` library for our explicitly parallel applications.

4.2 Evaluation Workload

We evaluate our clustering transformations using a latency-detection microbenchmark and five scientific applications. Table 2 summarizes the evaluation workload for the simulated and real systems. The number of processors used for the multiprocessor experiments is based on application scalability, with a limit of 16 on the simulated system and 8 on the real machine. Each code is compiled with the Sun SPARC SC4.2 compiler for the simulation and the Exemplar C compiler for the real machine, using the `-xO4` optimization level for the Sun compiler and `+O3` optimization level for the Exemplar compiler[4]. We incorporate miss clustering transformations by hand, following the algorithms presented in this paper.

Latbench is based on the `lat_mem_rd` kernel of `lmbench` [20]. `lat_mem_rd` sees inner-loop address recurrences from pointer-chasing. Latbench wraps this loop in an outer loop that iterates over different pointer chains, with no locality in or across chains. The pseudocode, given below, shows code added for Latbench in sans-serif.

```
for (j=0;j<N;j++){
  p = A[j];
  for(i=0;i<I;i++)
   p = p→next // serialized misses
  USE(p)      // keeps p live
```

4. We have not used the software prefetching supported by the Exemplar compiler, since prefetching and clustering may have complex interactions. Our ongoing work analyzes these interactions [8].

	Simulated system		Convex Exemplar	
Microbenchmark	**Input Size**	**Procs.**	**Input Size**	**Procs.**
Latbench	6.4M data size	1	40M data size	1
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
Mp3d	100K particles	1,8	2M particles	1
MST	1024 nodes	1	1024 nodes	1
Ocean	258x258 grid	1,8	1026x1026 grid	1,8

Table 2: Data set sizes and number of processors for experiments on simulated and real systems.

Latbench is clustered with unroll-and-jam. As in `lat_mem_rd`, looping overhead is minimized by unrolling the innermost loops to include 1000 pointer dereferences in each loop body, for both the base and clustered versions.

Erlebacher is a shared-memory port of a program by Thomas Eidson at the Institute for Computer Applications in Science and Engineering (ICASE). FFT, LU, and Ocean are from SPLASH-2 [15]. For better load balance, LU is modified slightly to use flags instead of barriers. These four regular codes see only cache-line recurrences. Each is clustered with unroll-and-jam and postlude interchanging.

Em3d is a shared-memory adaptation of a Split-C application [21], and is clustered using unroll-and-jam. This code has both cache-line and address dependences, but only cache-line recurrences. MST is the minimal-spanning tree algorithm from the Olden benchmarks [22]. MST uses recursive data structures and is dominated by linked-list traversals for lookups in a hash-table. The dominant loop nests in both applications have variable inner-loop lengths, so only the minimum length seen in the unrolled copies is fused (jammed). Each copy completes its remaining length separately. We assumed that the outer loops were explicitly identified as parallel to enable transformation despite the pointer references. We do not run a multiprocessor version of MST because of excessive synchronization overhead.

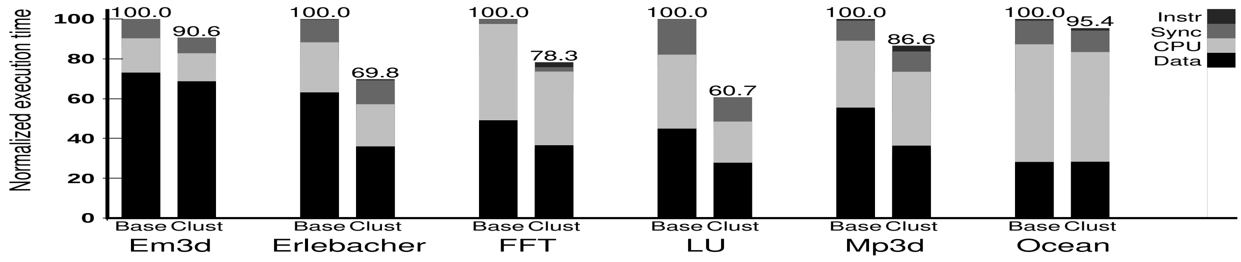
Mp3d is an irregular, asynchronous, communication-intensive SPLASH code [23]. To eliminate false-sharing, key data structures were padded to a multiple of the cache line size. To reduce true-sharing and improve locality, the data elements were sorted by position in the modeled physical world [24]. Mp3d has no recurrences, but sees poor miss clustering because of large loop bodies. Thus, inner-loop unrolling and aggressive scheduling can provide clustering here, as discussed in Section 3.3. We assumed that the dominant `move` loop was explicitly marked parallel. Despite these transformations, Mp3d does not see substantial multiprocessor speedup on the Convex Exemplar. As a result, it is only run as a uniprocessor code on the real machine.

5. Experimental Results

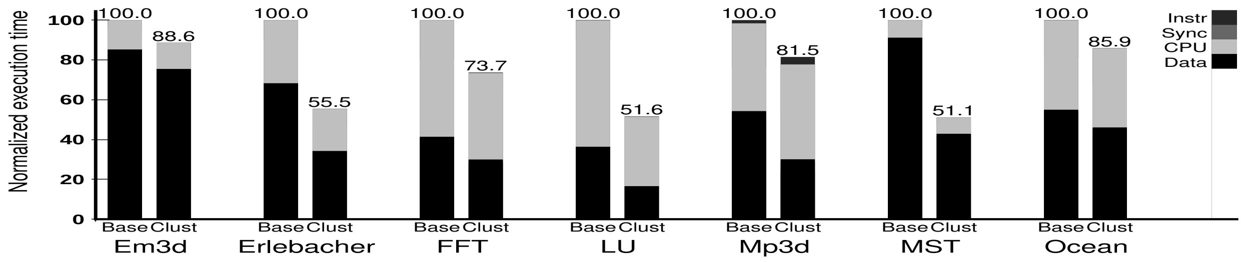
5.1 Performance of Latbench

The base Latbench of Section 4.2 indicates an average read miss stall time of 171 ns on the simulated system (identical to `lat_mem_rd`). Clustering drops the average stall time caused by each read miss to 32 ns, a speedup of 5.34X. On the Convex Exemplar, clustering reduces the average stall time for each miss from 502 ns to 87 ns, providing a speedup of 5.77X.

These results indicate the potential gains from memory parallelism transformations, but also indicate some bottlenecks, since the speedups are less than 10 (the number of simultaneous misses supported by each processor). Our more detailed statistics for the simulated system show that clustering increases con-



(a) Multiprocessor execution time



(b) Uniprocessor execution time

Figure 3: Impact of clustering transformations on application execution time.

tention, increasing average *total* latency from 171 ns to 316 ns (with total latencies measured from address generation to completion). Further, bus and memory bank utilization for the simulated system both exceed 85% after clustering. Thus, a further increase in speedup would require greater bandwidth at both the bus and the memory. (Total latencies and utilizations are not directly measurable for the real machine.)

5.2 Simulated Application Performance

Figure 3 shows the impact of the clustering transformations on application execution time for the base simulated system. Figure 3(a) shows multiprocessor experiments, while Figure 3(b) shows uniprocessor experiments. The execution time of each application is shown both before and after clustering (Base/Clust), normalized to the given application and system size without clustering. For analysis, execution time is categorized into data memory stall, CPU (which includes busy time and functional unit stall time), synchronization stall, and instruction memory stall times, following the conventions of previous work (e.g., [2, 25]). In particular, we calculate the ratio of the instructions retired from the instruction window in each cycle to the maximum retire rate of the processor. We attribute that fraction of the cycle to busy time. The rest of the cycle is attributed as stall time to the first instruction that could not be retired that cycle. Since writes can retire before completing and read hits are fast, nearly all data memory stalls stem from reads that miss in the L2 cache. That part of the miss latency that incurs stall time is called *exposed* miss latency.

Overall, the clustering transformations studied provide from 5–39% reduction in multiprocessor execution time for these applications, averaging 20%. The multiprocessor benefits in Erlebacher and Mp3d come almost entirely from reducing the memory stall time. (Mp3d sees some degradation in CPU time because of no scalar replacement or pipeline improvement and slightly worse return-address prediction.) Em3d, FFT, and LU see benefits split between memory stall time and CPU time; unroll-and-jam helps the CPU component through better functional unit utilization (in all three) and through scalar replacement (in FFT and LU). By speeding up the data producers in LU, the clustering transformations also reduce the synchronization time for data consumers. Our more detailed statistics show that the L2 miss count is nearly unchanged in all applications, indicating that locality is preserved and that scalar replacement primarily affects cache hits.

% execution time reduced	Em3d	Erlebacher	FFT	LU	Mp3d	MST	Ocean
multiprocessor	9.2	21.4	16.6	22.7	N/A	N/A	-2.9
uniprocessor	13.0	34.3	28.9	23.8	21.7	38.1	21.6

Table 3: Impact of clustering transformation on Convex Exemplar execution time.

The multiprocessor version of Ocean sees the smallest overall benefits from the clustering transformations, and those benefits are almost entirely the result of scalar replacement on CPU time. The potential benefits of clustering transformations on Ocean are limited because the base version of this application already sees some memory parallelism. Additionally, clustering increases conflict misses in this application. All applications except Ocean see more multiprocessor execution time reduction from the newly exposed benefits in read miss clustering than the previously studied benefits in CPU time.

The uniprocessor sees slightly larger overall benefits from the clustering transformations, ranging from 11–49% (average 30%). The speedup of data memory stalls is greater in the uniprocessor than in the multiprocessor, as the uniform latency and bandwidth characteristics of the uniprocessor better facilitate overlap. (For perfect overlap, misses of the *same latency* must be clustered together. Our algorithms do not consider this issue.) However, since the uniprocessor spends a substantially smaller fraction of time in data memory stalls than the multiprocessor for FFT and LU, the clustering transformations' benefits for these codes are predominantly in the CPU component. The only uniprocessor-specific application, MST, sees nearly all of its improvement in its dominant data memory stall component. The other applications respond to the clustering transformations qualitatively in the same way as in the multiprocessor system.

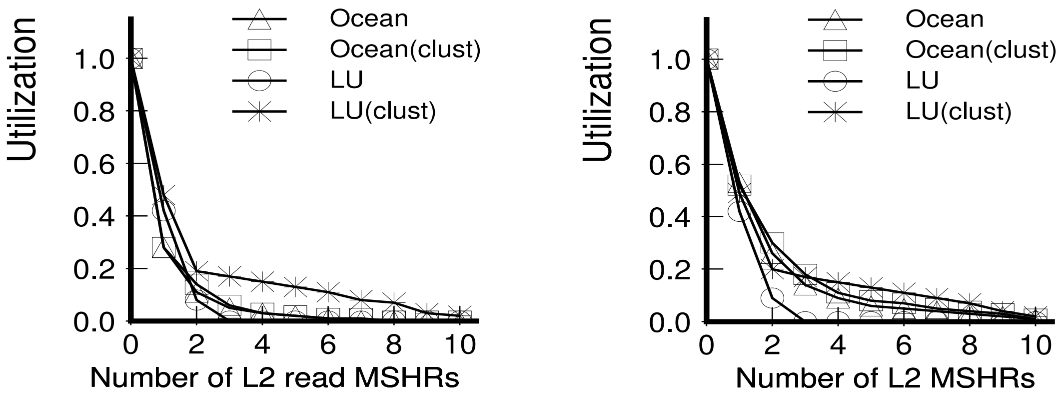
To represent the growing processor-memory speed gap, we also simulated a system with 1 GHz processors and all memory and interconnect parameters identical (in ns or MHz) to the base. The total execution time reductions are similar (5–36% in the multiprocessor, averaging 21%; 12–50% in the uniprocessor, averaging 33%). However, the larger fraction of memory stall time in these systems allows memory parallelism to provide more of the total benefits than in the base. Thus, targeting memory parallelism becomes more important for such potential future configurations.

All simulation experiments show few instruction memory stalls. Thus, the code added by our transformations does not significantly impact I-cache locality for these loop-intensive codes.

5.3 Exemplar Application Performance

This section repeats our experiments on a Convex Exemplar to confirm the benefits of clustering on a current system. There are numerous differences between the simulated system and the real machine, as described in Section 4.1. Table 3 shows that clustering also provides significant benefits in the real system. The multiprocessor and uniprocessor execution time reductions from clustering range from 9–38% for all but the multiprocessor version of Ocean, which sees a 3% degradation. (More detailed execution time breakdowns are not available on the real system.)

As in the simulation, the uniprocessor sees larger benefits than the multiprocessor. Although the SMP hypernode does not distinguish local and remote misses, there are still latency and bandwidth differences between ordinary and cache-to-cache misses. The multiprocessor configuration, however, sees lower benefits in the real machine than in simulation. This discrepancy could arise for two reasons. First, the memory banks and address busses of an SMP are shared by all the processors, potentially increasing multiprocessor contention relative to the simulated CC-NUMA system. Second, as discussed in Section 5.2, clustering adds new conflict misses to the multiprocessor version of Ocean. Such conflict misses would further increase the multiprocessor contention, causing a slight performance degradation. The clustering transformations provide similar uniprocessor execution time benefits in both the simulation and the real machine for all appli-



(a) Read miss parallelism

(b) Contention

Figure 4: Factors shaping memory parallelism (read L2 MSHR utilization) and contention (total L2 MSHR utilization).

cations except LU. The difference in LU may stem from the different interleaving policies of the simulated and real machines [16, 19].

5.4 Memory Parallelism and Contention

The MSHR utilization graphs of Figure 4 depict the sources of memory parallelism and contention for the simulated multiprocessor runs of Ocean and LU, the two extreme applications with regard to improvement from the transformations. (The corresponding data is not directly measurable on the real system.) Figure 4(a) indicates read miss parallelism, showing the fraction of total time for which at least N L2 MSHRs are occupied by read misses for each possible N on the X axis. The clustering transformations only slightly improve read miss parallelism for Ocean, since the stencil-type computations in Ocean give even the base version some clustering. In contrast, the transformations convert LU from a code that almost never had more than 1 outstanding read miss to one with 2 or more outstanding read misses 20% of the time and up to 9 outstanding read misses at times.

Figure 4(b) shows the total L2 MSHR utilization, including both reads and writes. This indicates contention, measuring how many requests use the memory system at once. In the unclustered version of the code, LU sees little additional contention from writes, while Ocean sees some write contention. By only targeting read misses, the clustering transformations do not further increase contention caused by writes. Any negative effects from increased read contention are largely offset by the performance benefits of read miss parallelism.

6. Conclusions and Future Work

This study finds that code transformations can improve memory parallelism in systems with out-of-order processors. We adapt compiler transformations known for other purposes to the new goal of memory parallelism. Our experimental results show substantial improvements in memory parallelism, thus hiding more memory stall time and reducing execution time significantly. As memory stalls become more important (e.g., multiprocessors or future systems with greater processor-memory speed gaps), more execution time reductions come from the transformations' newly exposed benefits in memory stall time than their previously studied benefits in CPU time.

We can extend this work in several ways. For example, we can seek to resolve memory-parallelism recurrences for unnested loops by fusing otherwise unrelated loops. We are also investigating the interactions of miss clustering with software prefetching, as their different approaches to latency tolerance allow each to provide distinct benefits.

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] V. S. Pai and S. Adve, "Code Transformations to Improve Memory Parallelism," in *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, pp. 147–155, November 1999.

[2] V. S. Pai, P. Ranganathan, H. Abdel-Shafi, and S. Adve, "The Impact of Exploiting Instruction-Level Parallelism on Shared-Memory Multiprocessors," *IEEE Transactions on Computers*, vol. 48, pp. 218–226, February 1999.

[3] F. E. Allen and J. Cocke, "A Catalogue of Optimizing Transformations," in *Design and Optimization of Compilers* (R. Rustin, ed.), pp. 1–30, Prentice-Hall, 1972.

[4] D. Callahan, J. Cocke, and K. Kennedy, "Estimating Interlock and Improving Balance for Pipelined Machines," *Journal of Parallel and Distributed Computing*, vol. 5, pp. 334–358, August 1988.

[5] S. Carr and K. Kennedy, "Improving the Ratio of Memory Operations to Floating-Point Operations in Loops," *ACM Transactions on Programming Languages and Systems*, vol. 16, pp. 1768–1810, November 1994.

[6] A. Nicolau, "Loop Quantization or Unwinding Done Right," in *Proceedings of the 1st International Conference on Supercomputing*, pp. 294–308, June 1987.

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

[8] V. S. Pai and S. Adve, "Improving Software Prefetching with Transformations to Increase Memory Parallelism," Tech. Rep. 9910, Department of Electrical and Computer Engineering, Rice University, November 1999.

[9] J. E. Smith and A. R. Pleszkun, "Implementing precise interrupts in pipelined processors," *IEEE Transactions on Computers*, vol. C-37, pp. 562–573, May 1988.

[10] S. Carr, "Combining Optimization for Cache and Instruction-Level Parallelism," in *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '96*, pp. 238–247, October 1996.

[11] M. E. Wolf and M. S. Lam, "A Data Locality Optimizing Algorithm," in *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pp. 30–44, June 1991.

[12] D. R. Kerns and S. J. Eggers, "Balanced Scheduling: Instruction Scheduling When Memory Latency is Uncertain," in *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pp. 278–289, June 1993.

[13] J. L. Lo and S. J. Eggers, "Improving Balanced Scheduling with Compiler Optimizations that Increase Instruction-Level Parallelism," in *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pp. 151–162, July 1995.

[14] V. S. Pai, P. Ranganathan, and S. V. Adve, "RSIM Reference Manual, Version 1.0," Tech. Rep. 9705, Department of Electrical and Computer Engineering, Rice University, August 1997.

[15] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 24–36, June 1995.

[16] G. S. Sohi, "High-Bandwidth Interleaved Memories for Vector Processors – A Simulation Study," *IEEE Transactions on Computers*, vol. 42, pp. 34–44, January 1993.

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

[18] D. Hunt, "Advanced Features of the 64-bit PA-8000," in *Proceedings of IEEE Compcon*, pp. 123–128, March 1995.

[19] D. T. Harper III and J. R. Jump, "Vector access performance in parallel memories using a skewed storage scheme," *IEEE Transactions on Computers*, vol. C-36, pp. 1440–1449, December 1987.

[20] L. McVoy and C. Staelin, "lmbench: Portable Tools for Performance Analysis," in *Proceedings of the 1996 USENIX Technical Conference*, pp. 279–295, January 1996.

[21] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick, "Parallel Programming in Split-C," in *Proceedings of Supercomputing '93*, pp. 262–273, November 1993.

[22] A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren, "Supporting dynamic data structures on distributed-memory machines," *ACM Transactions on Programming Languages and Systems*, vol. 17, pp. 233–263, March 1995.

[23] J. P. Singh, W.-D. Weber, and A. Gupta, "SPLASH: Stanford Parallel Applications for Shared-Memory," *Computer Architecture News*, vol. 20, pp. 5–44, March 1992.

[24] J. Mellor-Crummey, D. Whalley, and K. Kennedy, "Improving Memory Hierarchy Performance for Irregular Applications," in *Proceedings of the 13th ACM-SIGARCH International Conference on Supercomputing*, pp. 425–433, June 1999.

[25] C.-K. 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*, pp. 222–233, October 1996.