Journal of Instruction-Level Parallelism 1 (2004) XX-YY Submitted 5/03; published 6/04

The Efficacy of Software Prefetching and Locality Optimizations on Future Memory Systems*

Abdel-Hameed Badawy† ABSALAM@ENG.UMD.EDU
Aneesh Aggarwal† ANEESH@ENG.UMD.EDU
Donald Yeung† YEUNG@ENG.UMD.EDU
Chau-Wen Tseng‡ TSENG@CS.UMD.EDU
†*Electrical and Computer Engineering Dept.,*
‡*Computer Science Dept.,*
University of Maryland, College Park.

Abstract

Software prefetching and locality optimizations are techniques for overcoming the speed gap between processor and memory. In this paper, we provide a comprehensive summary of current software prefetching and locality optimization techniques, and evaluate the impact of memory trends on the effectiveness of these techniques for three types of applications: regular scientific codes, irregular scientific codes, and pointer-chasing codes. We find that for many applications, software prefetching outperforms locality optimizations when there is sufficient memory bandwidth, but locality optimizations outperform software prefetching under bandwidth-limited conditions. The break-even point (for 1 GHz processors) occurs at roughly 2.26 GBytes/sec on today's memory systems, and will increase on future memory systems. We also study the interactions between software prefetching and locality optimizations when applied in concert. Naively combining the techniques provides robustness to changes in memory bandwidth and latency, but does not yield additional performance gains. We propose and evaluate several algorithms to better integrate software prefetching and locality optimizations, including a modified tiling algorithm, padding for prefetching, and index prefetching. Finally, we investigate the interactions of stride-based hardware prefetching with our software techniques. We find that combining hardware and software prefetching yields similar performance to software prefetching alone, and that locality optimizations enable stride-based hardware prefetching for benchmarks that do not normally exhibit striding.

1. Introduction

Current microprocessors spend a large percentage of execution time on memory access stalls, even with large on-chip caches. Since processor speeds are growing at a greater rate than memory speeds, we expect memory access costs to become even more important in the future. Computer architects have been battling this *memory wall problem* [2] by designing ever larger and more sophisticated caches. Although caches are extremely effective, they are not the complete solution. Other techniques are required to fully address the memory wall problem.

Two promising approaches for improving memory performance are *software prefetching* and *locality optimizations*. The first executes explicit prefetch instructions to begin loading data from memory to cache. As long as prefetching begins early enough and the data is not evicted prior to its use, memory access latency can be completely hidden. However, as processor throughput improves due to memory latency tolerance, memory bandwidth use is increased since prefetching increases memory traffic. In comparison, locality optimizations use compiler or run-time transformations to change the computation order and/or data layout of a program to increase the probability it

©2004 AI Access Foundation and Morgan Kaufmann Publishers. All rights reserved.

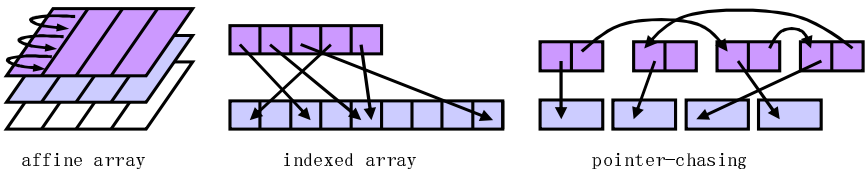

Figure 1: Examples of affine array, indexed array, and pointer-chasing memory access patterns.

accesses data already in cache. If successful, both average memory latency and bandwidth usage are reduced, since there will be fewer main memory accesses.

Both software prefetching and locality optimizations have been studied in isolation. In this paper, we examine how well each approach works for three types of data-intensive applications. Our evaluation uses a single unified environment to enable a meaningful comparison. A primary focus of our work is to compare the importance of *latency tolerance* provided by prefetching and *latency reduction* provided by locality optimizations on future high-performance memory systems. In addition, our work also investigates the interactions of software prefetching and locality optimizations when applied in concert. The contributions of this paper are as follows:

- We provide a comprehensive summary of current software prefetching and locality optimization techniques for three types of data-intensive applications: regular scientific codes, irregular scientific codes, and pointer-chasing codes.
- We compare the efficacy of software prefetching and locality optimizations for the three types of applications.
- We quantify the impact of bandwidth and latency scaling in future memory systems on the relative effectiveness of software prefetching and locality optimizations.
- We examine the performance of integrated software prefetching and locality optimizations, then propose and evaluate several enhancements to increase their combined effectiveness.
- We study the interactions of stride-based hardware prefetching with software prefetching and locality optimizations.

The rest of this paper is organized as follows. First, we describe three memory access patterns in Section 2. Then, we summarize the different techniques that have been previously developed for each access pattern. Section 3 discusses software prefetching techniques, and Section 4 discusses locality optimization techniques. Next, we present our experimental results in Section 5, and develop improved algorithms in Section 6. We also study the impact of stride-based hardware prefetchers on our results in Section 7. Finally, Section 8 discusses related work and Section 9 concludes the paper.

2. Memory Access Patterns

The types of software prefetching and locality optimizations which may be applied are dependent on the type of memory access pattern made by a program. We begin by presenting three important types of memory access patterns shown in Figure 1, using example codes in Figure 2.

2.1 Affine Array Accesses

The most basic memory access pattern is regular accesses with constant strides, such as references to elements in multidimensional arrays. The 2D Jacobi code in Figure 2 Part(A), commonly found

```
// Affine Array Accesses
// (2D Jacobi Kernel)
A(N,N,N),B(N,N,N)
do j=2,N-1
 do i=2,N-1
  A(i,j) = 0.25 *
   (B(i-1,j)+B(i+1,j)+
    B(i,j-1)+B(i,j+1))
```

Part(A)

```
// Indexed Array Accesses
// (Molecular Dynamics)
X(M),X2(M),index(N)
do t = 1, time
 do i = 1, N
  d = X1(index(i))-X2(index(i))
  force = d**(-7)-d**(-4)
  X1(index(i)) += force
  X2(index(i)) += -force
```

Part(B)

```
// Pointer-Based Structures
// (Linked List Traversal)
struct node {val, next} *ptr;
while (...) {
 ptr->next = malloc(node);
 ptr = ptr->next;
 ptr->val = ... ;
}
while (ptr->next)
 {ptr = ptr->next;⋯;}
```

Part(C)

Figure 2: Code examples for three classes of memory access patterns.

in finite-difference or multigrid solvers for systems of partial differential equations (PDEs), is an example code that performs very regular memory accesses. The code computes the value of a point in array A as the average of values of neighbors in all dimensions of array B.

Programs exhibiting these regular memory access patterns are also called *stencil* codes because they compute values based on applying a uniform *stencil* pattern repeatedly to each point of array A to compute the desired result. Array references result in memory accesses with constant strides if array subscripts are affine (i.e., linear combinations of loop index variables with constant coefficients and additive constants). These programs are also called *regular* codes because their memory access patterns are regular and well defined.

Affine array accesses are quite common in a variety of applications, including dense-matrix linear algebra and finite-difference PDE solvers as well as image processing and scans/joins in relational databases. These programs can usually exploit long cache lines to reduce memory access costs, but may suffer poor performance due to cache conflict and capacity misses arising from the large amounts of data accessed. An important feature of codes performing affine array accesses is that memory access patterns can be identified at compile time, assuming array dimension sizes are known. This allows both software prefetching and locality transformations to be determined precisely at compile time.

2.2 Indexed Array Accesses

Another memory access pattern is called indexed array accesses, because the main data array is accessed through a separate *index array* whose value is unknown at compile time. For example, consider the molecular dynamics code in Figure 2 Part(B), which calculates forces between pairs of atoms in a molecule. The index array *index* is accessed in an regular manner. In contrast, the two arrays $X1$ and $X2$ are indexed by the contents of the *index* array. The accesses are irregular due to the nature of the data stored in the index array, as shown in Figure 1. The cache performance of applications using indexed arrays can be poor since both spatial and temporal locality in such applications is typically low due to the irregularity of the access pattern.

Indexed array accesses arise in several scientific application domains where computational scientists attempt more complex simulations. In computational fluid dynamics, meshes for modeling large problems are sparse to reduce memory and computation requirements. In N-body solvers which arise in astrophysics and molecular dynamics, data structures are irregular because they model the positions of particles and their interactions. Index arrays are frequently used to store these more complicated relationships between data, since they are more efficient than pointers. Unfortunately, these irregular computations have poor temporal and spatial locality, and do not utilize processor caches efficiently. Unlike applications with affine accesses, compile-time transformations

```
                A(N,N,N), B(N,N,N)

                do j = 2, N-1
Prologue        do i = 2 , PD, step=4
  Loop             prefetch(&B[i][j])
                   prefetch(&B[i][j+1])
                   prefetch(&B[i][j-1])
                   prefetch(&A[i][j])

Unrolled        do i = 2, N-PD-1, step=4
  Loop             prefetch(&B[i+PD][j])
                   prefetch(&B[i+PD][j+1])
                   prefetch(&B[i+PD][j-1])
                   prefetch(&A[i+PD][j])

                   A(i, j) = 0.25*(B(i-1, j)+B(i+1, j)+B(i, j-1)+B(i, j+1))
                   A(i+1, j) = 0.25*(B(i, j)+B(i+2, j)+B(i+1, j-1)+B(i+1, j+1))
                   A(i+2, j) = 0.25*(B(i+1, j)+B(i+3, j)+B(i+2, j-1)+B(i+2, j+1))
                   A(i+3, j) = 0.25*(B(i+2, j)+B(i+4, j)+B(i+3, j-1)+B(i+3, j+1))

Epilogue        do i = N-PD, N-1
  Loop             A(i, j) = 0.25*(B(i-1, j)+B(i+1, j)+B(i, j-1)+B(i, j+1))
```

Figure 3: Example affine array prefetching for the 2D Jacobi kernel using Mowry's algorithm [3].

alone cannot improve locality because the values of the index array are not known at compile time. A combination of compile-time and run-time transformations are needed instead.

2.3 Pointer-Chasing Accesses

Finally, the third class of memory accesses are *pointer-chasing* codes that dynamically allocate memory and use pointer-based data structures such as linked lists, n-ary trees, and other graph structures. Figure 2 Part(C) shows an example of creating and traversing a singly-linked list. These programs are known as *pointer-chasing* codes, because accesses to additional parts of the data structure cannot take place until the pointer to the current portion of the data structure is resolved. This property forces associated memory references to be sequentialized, and is known as the *pointer-chasing problem*.

Pointer-chasing applications usually exist in programs solving complex problems where the amount and organization of data is unknown at compile time, requiring the use of pointers to manage both dynamic storage and linkage. They may also arise from high-level programming language constructs such as object-oriented programming. Because memory is allocated and accessed dynamically, the access pattern tends to be very irregular and lack locality, resulting in poor cache performance.

Indexed array accesses share characteristics to pointer-chasing codes, since the access to the index array must be resolved before and the data values can be accessed. However, indexed arrays have only one level of indirection, allowing many data elements and index array references to occur in parallel. In comparison, pointer accesses can have arbitrarily deep levels of indirection, sequentializing the entire pointer-chasing application.

Nodes in these pointer-chasing codes are usually dynamically allocated at run time, with their total number and connection pattern unknown at the time of compilation. To traverse the list, the pointers are dereferenced one after another in a serial manner. Because of the dynamic nature of data structure creation and modification, pointer-chasing codes commonly exhibit poor spatial and temporal locality and experience poor cache behavior. Pointer-chasing memory access patterns can be detected at compile time, but cannot be directly transformed by the compiler, since the pointer values are not known statically. Instead, cache-conscious run-time memory allocation can

be used to improve locality. In comparison, prefetching is not hindered by the fact that the memory locations are not known at compile-time, but is limited by the sequentialization of memory accesses.

3. Software Prefetching

Software prefetching relies on the programmer or compiler to insert explicit prefetch instructions into the application code for memory references that are likely to miss in the cache. At run time, the inserted prefetch instructions bring the data into the processor's cache in advance of its use, thus overlapping the cost of the memory access with useful work in the processor. Software prefetching has been shown to be effective in reducing memory stalls for both sequential and parallel applications, particularly for scientific programs making regular memory accesses [4, 5, 6, 3]. Recently, techniques have also been developed to apply prefetching to pointer-based data structures [7, 8, 9, 10, 11]. In this section of the paper, we present three software prefetching algorithms proposed previously in the literature for the three types of memory access patterns discussed in the previous section.

3.1 Affine Array Prefetching

To perform software prefetching for affine array references commonly found in scientific codes, we follow the well-known compiler algorithm proposed by Mowry [3]. Mowry's algorithm exploits the precise access pattern information available through static analysis of affine array references to insert prefetches for only the data needed by the processor, and no more. Consequently, the algorithm is quite effective and typically provides good performance gains.

Mowry's prefetch algorithm involves three steps. To illustrate, Figure 3 shows the 2D Jacobi kernel from Figure 2 Part(A) after all the steps have been applied. First, the affine array references within inner-most loops are identified as prefetching candidates. For each candidate, locality analysis is performed to determine which dynamic instances of the static memory reference will miss in the cache, hence requiring prefetching. To permit prefetching of only the missing dynamic instances, loop unrolling is performed to create multiple static instances of each memory reference within the loop body; the degree of loop unrolling is set to the number of back-to-back memory references Co-located in the same cache block (determined via locality analysis). By unrolling the loop this number of iterations, the dynamic instances that miss in the cache are isolated–the leading static memory reference in the unrolled loop always misses, while the remaining unrolled static memory references always hit. Hence, a single prefetch for the leading memory reference can be inserted into the unrolled loop, prefetching exactly the missing dynamic instances, thus avoiding unnecessary prefetches and reducing prefetch overhead.

Figure 3 illustrates the loop unrolling and prefetch insertion transformations for the 2D Jacobi kernel. In this loop, there are five affine references, four for the B array, and one for the A array. Assuming each reference accesses 8 bytes and assuming a 32-byte cache block, the loop should be unrolled by a factor of 4. After loop unrolling, four prefetches are inserted for the five leading memory references: $A(i,j)$, $B(i-1,j)$, $B(i+1,j)$, $B(i,j-1)$, and $B(i,j+1)$. Notice the $B(i-1,j)$ and $B(i+1,j)$ references lie on the same cache block, thus saving 1 prefetch.

The next step in Mowry's algorithm is to perform *prefetch scheduling*. Given the high memory latency of most modern memory systems, a single loop iteration normally contains insufficient work under which to hide the cost of memory accesses. To ensure that data arrive in time, the prefetches inserted in the first step of the algorithm must be initiated multiple iterations in advance.

The minimum number of loop iterations needed to fully overlap a memory access is known as the *prefetch distance*. Prefetch scheduling determines the prefetch distance and applies it to the inserted prefetches. Assuming the memory latency is l cycles and the work per loop iteration is w cycles, the prefetch distance, PD, is simply $\lceil \frac{l}{w} \rceil$. Figure 3 illustrates the indices of prefetched array elements contain a PD term, providing the early prefetch initiation required.

Finally, the last step is to "fix up" inefficiencies in prefetching created by prefetch scheduling. By modifying prefetches to initiate PD iterations in advance as illustrated in Figure 3, the first PD iterations do not get prefetched, and the last PD iterations prefetch past the end of each array. These inefficiencies can be addressed by performing loop peeling to handle the first and last PD iterations of the loop separately. The loop peeling transformation creates a *prologue loop* to execute PD prefetches for the first PD array elements, and an *epilogue loop* to execute the last PD iterations without prefetching. Figure 3 illustrates the prologue and epilogue loops created by loop peeling.

3.2 Indexed Array Prefetching

Indexed array accesses, of the form $A(B(i))$, are common in irregular scientific codes. We use the prefetch algorithm for indexed array accesses proposed in [12] by Mowry, which is an extension to his algorithm for affine arrays described in Section 3.1. The extensions stem from three major differences between indexed and affine array accesses. First, indexed array accesses contain two array references per prefetch candidate–one for the data array and one for the index array. Affine array accesses perform only a single array reference per prefetch candidate. Second, static analysis cannot determine locality information for consecutively accessed indexed array elements since their addresses depend on the index array values known only at runtime. Finally, before a data array reference can perform, the corresponding index array reference must complete since the index value is used to index into the data array. Hence, pairs of data and index array references are serialized.

Mowry's indexed array prefetching algorithm follows the same three steps for prefetching affine arrays: loop unrolling, prefetch scheduling, and loop peeling. Each step is modified to accommodate the added complexities of indexed array accesses compared to affine array accesses. To illustrate, Figure 4 shows a simplified molecular dynamics kernel after all the steps have been performed. First, loop unrolling is applied to isolate the missing dynamic instances, as described in Section 3.1. Unfortunately, cache-miss isolation via loop unrolling succeeds only for the index array accesses. Since the location of consecutively accessed data array elements depends on the index array values at runtime, static analysis fails to provide spatial locality information for the data array references, so the compiler must be conservative and assume all data array references lie on separate cache blocks. Consequently, while a single prefetch is sufficient for all instances of the index array in the unrolled loop body, a separate prefetch is necessary for every instance of the data array. Figure 4 illustrates the unrolled loop and the inserted prefetch code. The loop unrolling degree is two to limit the size of the example code.

Next, prefetch scheduling is performed. As in affine array prefetching, the computation of the prefetch distance uses the formula, $PD = \lceil \frac{l}{w} \rceil$. However, the adjustment of array indices for indexed arrays must take into consideration the serialization of data array and index array references. Since data array elements cannot be prefetched until the index array values they depend on are available, prefetches for index array elements should be initiated twice as early as data array elements. This ensures that an index array value is in cache when its corresponding data array prefetch is issued. Figure 4 illustrates the indices of prefetched data array elements contain the normal PD term, but

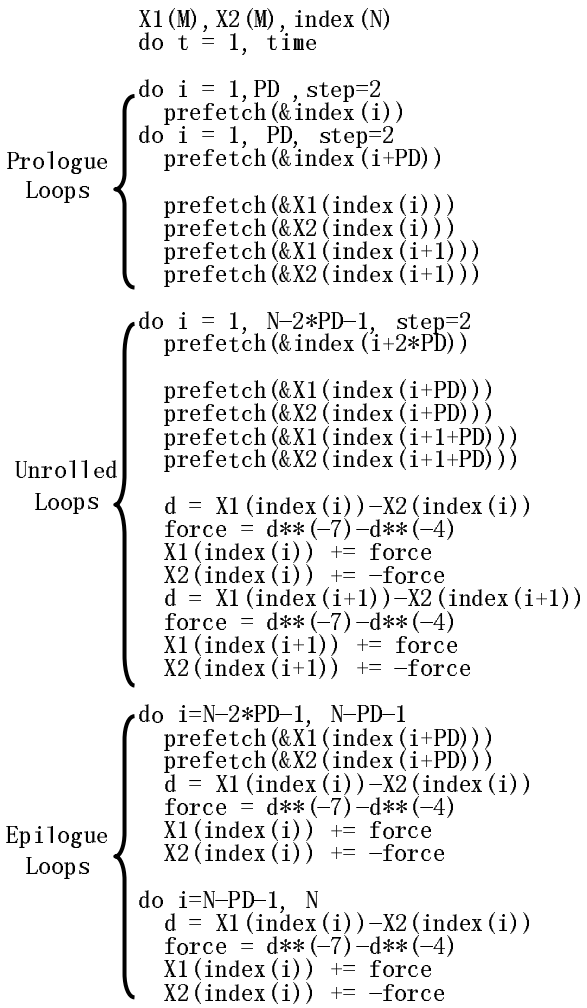

Figure 4: Example indexed array prefetching

prefetched index array elements contain a $2 * PD$ term to fetch them in advance of the data array prefetches.

Lastly, loop peeling is performed. As described in Section 3.1, prologue and epilogue loops are inserted to properly handle the first and last few iterations of the loop. For indexed array prefetching, two prologue loops and two epilogue loops are necessary because prefetch scheduling uses two prefetch distances, PD and $2 * PD$. The first prologue loop issues the index array prefetches for the first PD loop iterations. Then, the second prologue loop issues the index array prefetches for the next PD loop iterations as well as the data array prefetches for the first PD loop iterations. Similarly, the first epilogue loop executes the second-to-last PD loop iterations without index array prefetching. Finally, the last epilogue loop executes the last PD loop iterations without any prefetching. Figure 4 illustrates the modified loop peeling transformation.

3.3 Pointer-Chasing Prefetching

Prefetching for pointer-based data structures is challenging due to the memory serialization effects associated with traversing pointer structures. The memory operations performed for array traversal can issue in parallel because individual array elements can be referenced independently. At worst, pairs of array references are serialized in the case of indexed array traversal. But even in that case, separate indexed array references can perform in parallel. In contrast, the memory operations

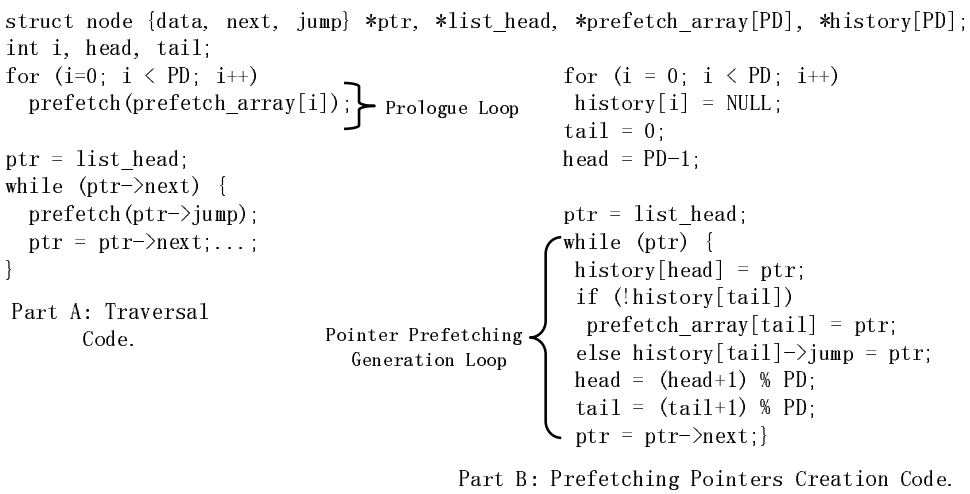

Figure 5: Example pointer prefetching using jump pointers and prefetch arrays [7].

performed for pointer traversal must dereference a series of pointers sequentially. The memory serialization in pointer chasing prevents conventional prefetching techniques from overlapping cache misses suffered along a pointer chain, thus limiting their effectiveness.

Jump pointer prefetching [8, 11] is a promising approach for addressing the pointer-chasing problem. In jump pointer prefetching, additional pointers are inserted into a dynamic data structure to connect non-consecutive link elements. These "jump pointers" allow prefetch instructions to name link elements further down the pointer chain (*i.e.* a prefetch distance, PD, away which is computed as in Sections 3.1 and 3.2) without sequentially traversing the intermediate links. Consequently, prefetch instructions can overlap the fetch of multiple link elements simultaneously by issuing prefetches through the memory addresses stored in the jump pointers. Figure 5 Part(A) illustrates a "while" loop that has been instrumented with jump pointer prefetching.

Jump pointer prefetching, however, cannot prefetch the first PD link nodes in a linked list because there are no jump pointers that point to these early nodes. To enable prefetching of early nodes, jump pointer prefetching can be extended with *prefetch arrays* [7]. In this technique, an array of prefetch pointers is added to every linked list to point to the first PD link nodes. Hence, prefetches can be issued through the memory addresses in the prefetch arrays before traversing each linked list to cover the early nodes, much like the prologue loops in affine array and indexed array prefetching prefetch the first PD array elements. Figure 5 Part(A) illustrates the addition of a prologue loop that performs prefetching through a prefetch array.

Before prefetching can commence, the prefetch pointers must be set. Figure 5 Part(B) shows an example of prefetch pointer initialization code which uses a *history pointer array* [8] to set the prefetch pointers. The history pointer array, called "history" in Figure 5, is a circular queue that records the last PD link nodes traversed by the initialization code. Whenever a new link node is traversed, it is added to the head of the circular queue and the head is incremented. At the same time, the tail of the circular queue is tested. If the tail is NULL, then the current node is one of the first PD link nodes in the list since PD link nodes must be encountered before the circular queue fills. In this case, we set one of the "prefetch_array" pointers to point to the node. Otherwise, the tail's jump pointer is set to point to the current link node. Since the circular queue has depth PD, all jump pointers are initialized to point PD link nodes *ahead*, thus providing the proper prefetch distance. Normally, the compiler or programmer ensures the prefetch pointer initialization code gets executed prior to prefetching, for example on the first traversal of a linked data structure. Furthermore, if the application modifies the linked data structure after the prefetch pointers have

```
// Tiled 3D Jacobi
A(N,N,N), B(N,N,N)
do kk=2,N-1,TK                    TIxTJxTK Tile
do jj=2,N-1,TJ
do ii=2,N-1,TI
  do k=kk,kk+TK-1                 Tiled Loops
  do j=jj,jj+TJ-1
  do i=ii,ii+TI-1
    A(i,j,k) = 0.16667 *
     (B(i-1, j, k) + B(i, j-1, k)+
     B(i+1, j, k) + B(i, j+1, k)+
     B(i, j, k-1) + B(i, j, k+1))
```

Part(A)

```
//Inspector-Executor for
//Molecular Dynamics
inspect_reorder(&E(2,N))
do t = 1, time
if (recalc)
 E(...) = ...
do i = 1, N
 d = X(E(1,i))-X(E(2,i))
 force = d**(-7)-d**(-4)
 Y(E(1,i)) += force
 Y(E(2,i)) += -force
```

Part(B)

```
// Pointer-Based Structures
// (Linked List Traversal)
 struct node{val, next} *ptr,*list;
while (...) {
 ptr->next = ccmalloc(node);
 pr = ptr->next;
 ptr->val = ... ;
}
while (ptr->next) {
 ptr = ptr->next;...;
}
```

Part(C)

Figure 6: Example Locality optimized affine array, indexed array and pointer-chasing access codes.

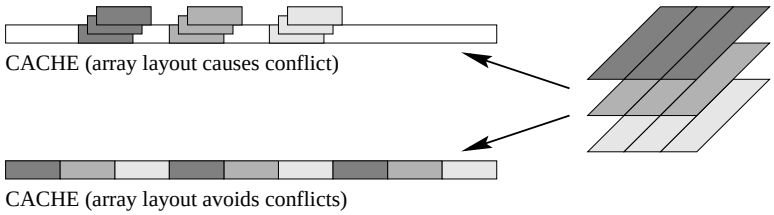

Figure 7: Example of conflict misses under two array layouts.

been initialized, it may be necessary to update the prefetch pointers either by re-executing the initialization code or by using other fix-up code.

4. Locality Optimizations

Software prefetching tries to hide memory latency while retaining the original program structure. Another alternative is to reduce memory accesses by changing the computation order and data layout of the program at compile and run time using *locality optimizations*. These optimizations try to improve *data locality*, the ability of an application to reuse data in the cache [13]. Reuse may be in the form of *temporal locality*, where the same cache line is accessed multiple times, or *spatial locality*, where nearby data is accessed together on the same cache line. Previous researchers have developed many locality optimizations. In this section, we consider optimizations for the three types of access patterns discussed in Section 2.

4.1 Tiling for Affine Accesses

In many way, programs with affine array accesses are the easiest for compilers to apply locality optimizations, since memory access patterns can be fully analyzed at compile time. One useful program transformation is *tiling* (blocking), which combines strip-mining with loop permutation to form small tiles of loop iterations which are executed together to exploit data locality [13]. Figure 6 Part(A) demonstrates how the 3D Jacobi code can be tiled. By rearranging the loop structure so that the innermost loops can fit in cache (due to fewer iterations), tiling allows reuse to be exploited on all the tiled dimensions so that data in cache can be accessed multiple times before it is flushed.

Tiling is very effective with linear algebra codes [14, 15, 16, 17, 18], and has been been extended to handle stencil codes used in iterative PDE solvers as well [19, 20, 13]. A major problem with tiling is that limited cache associativity may cause data in a tile to be mapped onto the same cache lines, even though there is sufficient space in the cache. Conflict misses will result, causing tile data to be evicted from cache before they may be reused [16]. This effect is shown in Figure 7.

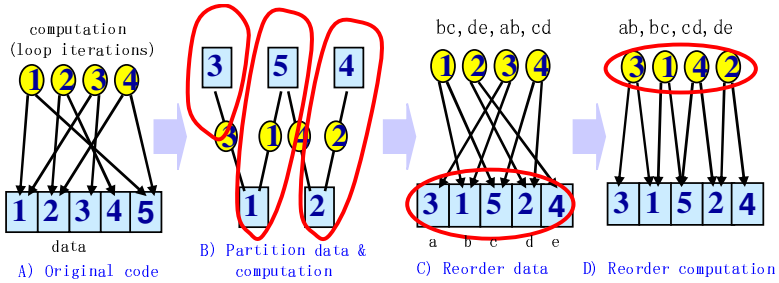

Figure 8: GPART algorithm for reordering data and computation in indexed array computations.

Previous research found *tile size selection* and *array padding* can be applied to avoid conflict misses in tiles [14, 17, 18]. Tile-size-selection algorithms carefully select tile dimensions tailored to individual array dimensions so that no conflicts occur. For 2D arrays, the Euclidean remainder algorithm may be used to quickly compute a sequence of non-conflicting tile dimensions through a simple recurrence [14, 18]. An alternative algorithm finds non-conflicting 2D tile using an greedy algorithm which expands tile dimensions while checking that no conflicts occur [16]. We can adapt this algorithm for finding non-conflicting 3D tiles by iteratively attempting to increase each tile dimension until none may be increased without introducing conflicts [19].

Tile size selection by itself may yield poor results since only small tiles may be able to avoid conflicts, particularly for array sizes near powers of two. One possible solution is to use padding to enable better tile sizes [21]. Padding increases the size of leading array dimensions, increasing the range of non-conflicting tile shapes. It has proven to be very useful for improving tiling for 2D linear algebra codes [18]. To combine padding with tile size selection for 2D arrays, we can test a small set of pads and choose the best choice. For 3D tiles, we would need to evaluate a much larger space of possible pads, so we extend the algorithm to stop searching for pad sizes when the predicted miss rate is within a small percentage of the predicted optimal [19].

4.2 Reordering for Indexed Accesses

Programs with index array accesses access data in an irregular manner, depending on the values in the index array. If data is accessed in an irregular manner, spatial locality is unlikely to be obtained if the data is larger than the cache. Fortunately, recent research has demonstrated run-time data and computation transformations can improve the locality of irregular computations [22, 23, 24, 25].

Because many irregular computations typically perform *reduction* (commutative and associative) operations such as SUM and MAX, loop iterations can be safely reordered to bring accesses to the same data closer together in time. Data layout can also be transformed so that data accesses are more likely to be to the same cache line. These compiler and run-time transformations can be automated using an inspector-executor approached developed for message-passing machines [26], where the compiler identifies index accesses and inserts calls to run-time libraries to analyze and reorder data and loop iterations. Figure 6 Part(B) illustrate how a code with indexed accesses may be optimized using a combination of compile and run-time transformations. The invocation of the inspector library routine rearranges both the index array and data arrays to bring memory accesses closer in time and space, resulting in better cache performance.

Figure 8 illustrates the process by which an inspector can reorder data and computation of an index array computation at run time to improve locality. In the figure, loop iterations are represented as circles and data elements as squares. Many indexed array codes compute values based on pairs of indexed values (e.g., endpoints of a mesh segment, forces between a pair of particles). Such computations may be viewed as an undirected graph, where each loop iteration

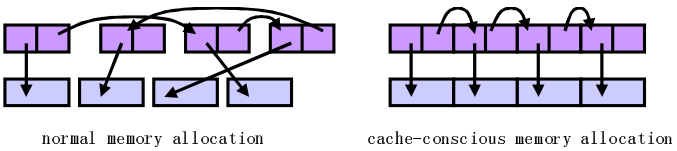

Figure 9: ccmalloc algorithm for improving memory allocation for pointer-chasing codes.

forms an edge connecting the pair of data elements accessed by the iteration. Each circle thus connects two squares.

In this representation, improving locality may be viewed as selecting an ordering of circles (loop iterations) and squares (data elements) so that nearby circles access identical or nearby squares. Several data and computation locality transformations exist to solve the problem of improving the locality of such graphs [22, 23, 25, 24]. Our evaluation uses a technique called gpart that relies on hierarchical clustering to improve locality [27, 28]. gpart works in three steps. First, the graph formed by the index array computation hierarchically partitioned into roughly cache-sized chunks. Second, the partitioning is then used to reorder the data to improve spatial locality. Finally, loop iterations are lexicographically sorted based on their data accesses to improve temporal locality. Experiments have shown gpart closely matches the performance of more sophisticated partitioning algorithms, with much less run-time overhead.

4.3 Memory Allocation For Pointers

Pointer-based programs frequently suffer from poor locality and are notoriously difficult to analyze and transform because of their reliance on pointers and dynamically allocated recursive data structures. Pointer-based applications are harder to optimize than indexed array codes, since pointer chasing forces link nodes to be traversed sequentially. Fortunately, researchers have developed *cache-conscious* heap allocation and transformation techniques to improve locality for pointer-based programs [29, 30, 31]. These techniques improve locality by assigning or changing the locations of dynamically allocated memory in ways designed to improve spatial locality. Examples of cache-conscious algorithms include run-time tree optimization routines that place parent nodes with child nodes for improved locality, and coloring when placing tree nodes to avoid conflict with the root.

Of particular interest is ccmalloc, a customized memory allocator which allocates memory in a location near to a user-specified address. ccmalloc is a heuristic that reserves space for future allocation requests when allocating new blocks of data [30]. Using this memory allocator, multiple members of a linked list are thus more likely to be in adjacent memory locations. Not only does this take advantage of hardware prefetching of long cache lines, but cache line utilization increases and fragmentation is reduced, decreasing the probability that useful cache lines will be evicted from cache.

Figure 6 Part(C) shows a simple list allocation code that uses ccmalloc to allocate list nodes on nearby cache blocks dynamically. The code modification is quite simple: replacing malloc with ccmalloc. Compiler analysis of pointer accesses can be used to determine when cache-conscious memory allocators should be used. For our experimental evaluation, we inserted calls to ccmalloc by hand in our pointer-chasing benchmark codes. ccmalloc for our evaluation works by taking a optional pointer argument during memory allocation, and allocating current and future nodes close to it whenever possible. To increase the probability of nearby placement, ccmalloc reserves space for future data blocks when allocating the first node [30]. Figure 9 displays an example of cache-conscious memory allocation to improve locality for pointer-chasing codes.

Note one problem with ccmalloc is that dynamic data structures which change after allocation may not benefit from this optimization. Frequent insert and delete operations after allocation will make logically contiguous nodes physically non-contiguous after a few deletions and/or insertions. Our pointer-chasing benchmarks did not perform many deletions to linked data structures.

5. Experimental Evaluation

This section evaluates the performance of software prefetching and locality optimizations, first independently, and then in concert. We describe our experimental methodology. Then, we compare software prefetching and locality optimizations under different memory bandwidths and latencies. Finally, we study their combination.

5.1 Methodology

Our experimental evaluation uses the SimpleScalar tool set [32] to model a 1GHz 4-way issue dynamically-scheduled processor. The simulator models all aspects of the processor including the instruction fetch unit, the branch predictor, register renaming, the functional unit pipelines, and the reorder buffer. To enable software prefetching, we added a prefetch instruction to the ISA of the processor model. In addition, our simulator also models the memory system in detail. We assume a split 8-Kbyte direct-mapped L1 cache with 32-byte cache blocks, and a unified 256-Kbyte 4-way set-associative L2 cache with 64-byte cache blocks. Although the caches are small, they match the more modest input data sets required for simulation.

Several of our experiments study sensitivity to memory latency and memory bandwidth. To facilitate these experiments, we modified the SimpleScalar simulator to accurately model bus contention across the L2-memory bus. Then, we varied the L2-memory latency from 80 to 640 cycles, and varied the L2-memory bus bandwidth between 1-64 Gbytes/sec (note that a bandwidth of 1 Gbyte/sec is equivalent to the processor loading one byte per cycle). The lower end of both the latency and bandwidth ranges captures the trends of existing memory systems which have latencies of around 100 cycles and bandwidths of around 2-3 GBytes/sec. The mid and high end of the latency and bandwidth ranges capture the characteristics of future architectures.

For all experiments, transfers across the L1-L2 bus incur a 7-cycle latency, and we assumed the L1-L2 bus has infinite bandwidth. We also assumed an unlimited number of MSHRs to maximize concurrency in the memory system, thus exposing memory bandwidth limitations. We don't expect the infinite L1-L2 bandwidth assumption to affect our results. Our benchmarks are memory intensive with very large working sets, so they tend to be bandwidth limited at the memory sub-system level. However, the unlimited MSHRs assumption is significant. Without sufficient MSHRs, prefetching performance would degrade since the number of outstanding prefetches would be limited. We optimistically assume enough MSHRs are provided in the caches to maximize the effectiveness of prefetching.

To drive our simulations, our experimental evaluation employs nine benchmarks, representing the three classes of data-intensive applications described in Section 2. Table 1 lists the benchmarks along with their problem sizes and memory access patterns.

The first three applications in Table 1 perform affine array accesses. MM multiplies two matrices, RB performs a 3D red-black successive-over-relaxation, and Jacobi performs a 3D Jacobi relaxation. Both Jacobi and RB are frequently found in multigrid PDE solvers, such as Mgrid from the SPEC/NAS benchmark suite. The next three applications perform indexed array ac-

Application	Problem Size	Access Pattern
RB	200x200x8 grid	Affine array
Jacobi	200x200x8 grid	Affine array
MM	200x200 matrices	Affine array
Irreg	14K node mesh	Indexed array
Moldyn	13K molecules	Indexed array
NBF	144K mols	Indexed array
Health	5 levels, 500 iters	Pointer-chasing
MST	1024 nodes	Pointer-chasing
EM3D	10K nodes	Pointer-chasing

Table 1: Benchmark summary.

Latency	RB	Jacobi	MM	Irreg	Moldyn	NBF	Health	MST	EM3D
80	12	8 , 36	24	8 , 20 , 20 , 40	1, 1, 2	2	31	3	2
160	24	16, 68	44	12, 40 , 40 , 80	2, 2, 3	4	62	3	3
320	48	28, 136	88	24, 80 , 80 , 160	4, 4, 5	8	124	3	6
640	96	56, 268	176	44, 160, 160, 319	7, 7, 9	16	247	3	11

Table 2: Prefetch distances for loops in all benchmarks versus latency in cycles.

cesses. Irreg is an iterative PDE solver for an irregular mesh, Moldyn is abstracted from the non-bonded force calculation in CHARMM, a key molecular dynamics application used at NIH to model macromolecular systems, and NBF (Non Bonded Force kernel), is a molecular dynamics simulation. NBF is taken from the GROMOS benchmark [33].

The last three applications perform pointer-chasing accesses. Health simulates the Columbian health care system, MST computes a minimum spanning tree, and EM3D simulates electromagnetic wave propagation through 3D objects. Health, MST, and EM3D are all from the olden benchmark suite [34].

For each application, we applied software prefetching and locality optimizations by hand, first in isolation, then in combination. We followed the algorithms described in Sections 3 and 4, applying the appropriate algorithm to each application given its memory access pattern. We then measured the performance of the optimized codes on our detailed architectural simulator.

As described in Section 3, software prefetching requires computing a prefetch distance, PD, to properly schedule prefetches. Recall that $PD = \lceil \frac{l}{w} \rceil$, where w is the work per loop iteration, and l is the memory latency (see Section 3.1). Hence, PD must be recomputed not only for every loop, but also for every memory latency setting. Table 2 reports the computed prefetch distances for the affine array, indexed array, and pointer-chasing benchmarks. For each benchmark, we list prefetch distances for four different memory latencies used in our experiments. In applications with multiple instrumented loops, the prefetch distance for each loop is listed separately.

Latency	RB	Jacobi	MM	Irreg	Moldyn	NBF	EM3D	Average
80	2.08	1.57	1.84	2.80	3.51	N/A	1.75	2.26
160	2.65	1.89	2.84	2.93	2.86	3.37	1.83	2.62
320	2.94	2.05	3.82	3.04	2.99	3.62	1.87	2.90
640	3.17	2.10	4.68	3.20	3.31	3.81	1.89	3.20

Table 3: Equi-performance bandwidth in Gbytes/sec versus memory latency in cycles. The last column reports the average per latency.

Finally, as described in Section 4, locality optimizations for indexed array benchmarks reorder computations using graph partitioning techniques to improve memory performance. For our indexed array benchmarks, we apply the RCB computation reordering algorithm.

5.2 Varying Memory Bandwidth

In this section, we evaluate the performance of software prefetching and locality optimizations under memory bandwidth scaling. Figure 10 plots execution time along the y-axis, and varies memory bandwidth from 1-64 Gbytes/sec along the x-axis, keeping memory latency fixed at 80 cycles. Each execution-time bar is broken down into memory stall, software overhead, and computation components. Groups of bars represent the original version of each program, and versions optimized with either software prefetching, locality optimization, or both. In this section, we focus only on applying the techniques in isolation. Later in Section 5.4, we will examine the techniques in combination.

For the affine array and indexed array benchmarks, both software prefetching and locality optimizations provide significant performance gains, improving performance by 45% on average over unoptimized codes. Comparing the techniques, we see two major differences. First, software prefetching incurs more overhead than locality optimizations. This overhead is due primarily to prefetch and related address computation instructions. In particular, loop unrolling is unable to isolate cache misses for index array accesses, as described in Section 3.2, contributing to increased runtime overhead for the indexed array benchmarks. Overall, locality optimizations exhibit very low overheads. Tiling incurs some overhead for the extra levels of loops, but this is minimal. For the indexed arrays benchmarks, the RCB algorithm has no measurable overhead since the runtime inspector is amortized over lots of computations [28].

Second, the relative effectiveness of software prefetching and locality optimizations to eliminate memory stalls depends on available memory bandwidth. At high memory bandwidths, software prefetching eliminates practically all memory stalls since the memory system can sustain the simultaneous memory requests necessary to hide all the memory latency. As memory bandwidth is reduced, memory requests must serialize, thus software prefetching loses its effectiveness. In contrast, locality optimizations reduce memory latency, and hence, memory traffic. This makes them highly effective at low bandwidths where reduced traffic pays off. However, locality optimizations cannot eliminate all memory stalls, so they achieve a lower maximum performance compared to software prefetching. Consequently, for all the array-based benchmarks except for NBF at 80 cycles of latency, software prefetching outperforms locality optimizations at high memory bandwidths, while locality optimizations outperform software prefetching at low memory bandwidths.

Table 3 reports the memory bandwidths at which software prefetching and locality optimizations achieve equal performance. Memory systems providing memory bandwidths higher than this *equi-performance bandwidth* favor software prefetching, while those providing lower memory bandwidths favor locality optimizations. For an 80-cycle memory latency corresponding to the data in Figure 10, Table 3 shows the average equi-performance bandwidth is 2.41 Gbytes/sec. (Note, NBF does not have an equi-performance bandwidth at 80 cycles of latency since locality optimization outperforms software prefetching at all memory bandwidths). Such a large equi-performance bandwidth underscores the importance of latency reduction techniques, and implies future memory systems must provide significant memory bandwidth before prefetching can outperform locality optimizations on these data-intensive applications.

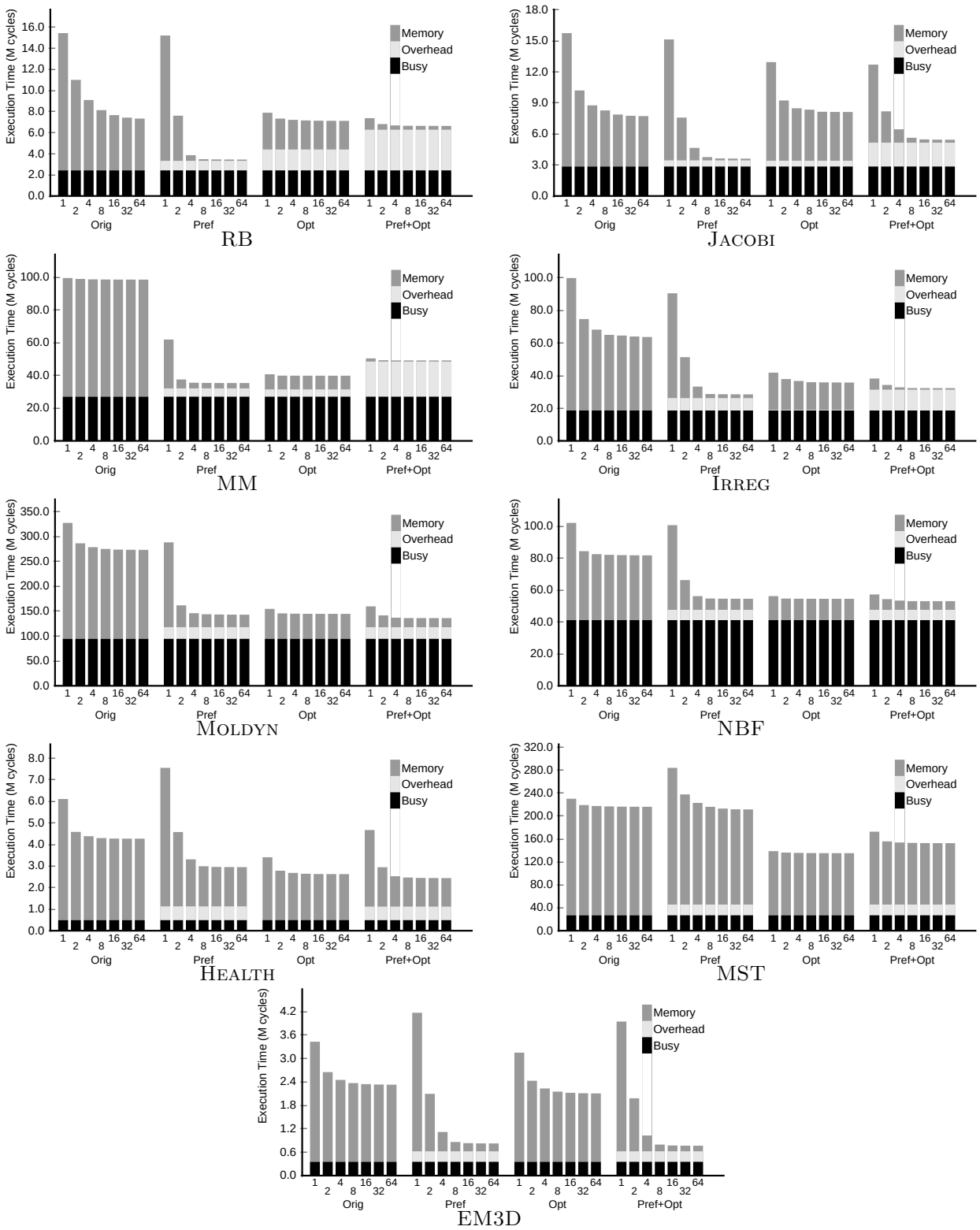

Figure 10: Execution time breakdown under memory bandwidth scaling at 80 cycles of memory latency with no optimizations (Orig), with software prefetching (Pref), with locality optimization (Opt), and with combined optimizations (Pref+Opt).

In the pointer-chasing benchmarks, locality optimization outperforms software prefetching at all memory bandwidths for Health and MST. This is due to three factors. First, pointer prefetching incurs high software overhead to create and manage jump pointers. Software overhead in Health and MST is 131% and 70%, respectively, of the busy component. In contrast, ccmalloc memory allocation incurs no measurable overhead. Second, the traversal loops in our pointer-chasing codes are short, particularly for MST, and do not provide sufficient work under which to hide memory latency. Hence, software prefetching cannot eliminate all memory stalls. Finally, pointer prefetching requires jump pointer storage that increases the cache miss rate and memory bandwidth consumption, making the optimized code even more data-intensive than the original code. As Figure 10 shows, software prefetching in Health and MST (as well as EM3D) performs worse than the original code at low memory bandwidths.

In EM3D, software prefetching achieves higher performance as compared to Health and MST; hence, its behavior resembles the array benchmarks rather than the other pointer-chasing benchmarks. This improved software prefetching performance is due to EM3D's primary data structure. Whereas Health and MST employ linked lists that are difficult to prefetch for the reasons explained above, EM3D uses an "array of lists" data structure that is more amenable to software prefetching. Specifically, EM3D simulates alternating electric and magnetic fields characteristic of electro-magnetic wave propagation via two arrays, one for "e-nodes" and another for "h-nodes" [35]. "Edge pointers" connect e-nodes to h-nodes and vice versa to form a bipartite graph, representing a mesh over which the electro-magnetic fields are simulated [36]. Like MST, the traversal code for each edge pointer is short, potentially limiting prefetching. However, because the edge pointers are rooted inside e-node and h-node array elements (and because these arrays are both very large), jump pointers can be created to name e-nodes and h-nodes well in advance of their traversal, resulting in effective memory latency tolerance. Like the array benchmarks, software prefetching achieves better performance than ccmalloc at high bandwidths, and worse performance than ccmalloc at low bandwidths. Consequently, an equi-performance bandwidth exists for EM3D as shown in Table 3, and is 1.7 Gbytes/sec at 80 cycles of latency.

5.3 Varying Memory Latency

Figures 11, 12, and 13 evaluate software prefetching and locality optimizations under memory latency scaling. Similar to Figure 10, we plot execution time versus memory bandwidth. In addition, we present results for 80, 160, 320, and 640-cycle memory latencies on separate lines in each graph. Each version of a program (original, prefetch, optimized, both) is displayed in a separate graph. Once again, we focus on applying the techniques in isolation, leaving a discussion of the combined techniques to Section 5.4.

Not surprisingly, execution time for all program versions increases as we scale memory latency. For the affine array and indexed array benchmarks, software prefetching effectively hides the increasing memory latencies given sufficient memory bandwidth. In contrast, locality optimizations suffer performance degradation as memory latencies grow; however, they still enjoy the benefit of reduced traffic at low memory bandwidths. As a result, software prefetching outperforms locality optimizations at high memory bandwidths, while locality optimizations outperform software prefetching at low memory bandwidths for all the memory latencies we simulated. Note, NBF at 80 cycles of latency is the one exception, as observed in the previous section, since locality optimization is slightly better than prefetching at all bandwidths. Table 3 shows equi-performance bandwidths generally increase with memory latency. Consequently, on future systems with high memory la-

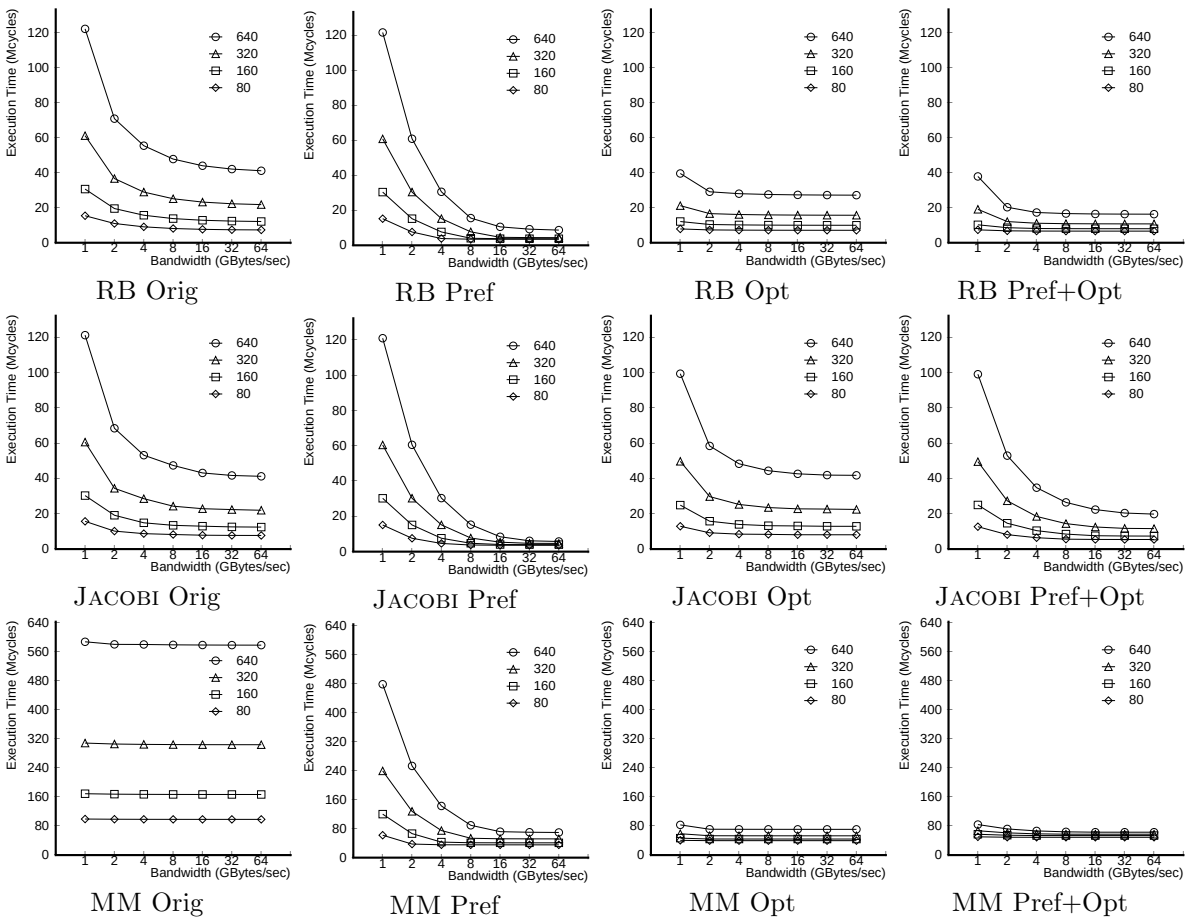

Figure 11: Execution time under both memory bandwidth and latency scaling for affine array benchmarks with no optimizations (Orig), with software prefetching (Pref), with locality optimization (Opt), and with combined optimizations (Pref+Opt).

tencies, greater memory bandwidth will be required before software prefetching demonstrates a performance advantage over locality optimizations.

In the pointer-chasing benchmarks, locality optimization outperforms software prefetching at all memory latencies and bandwidths for Health and MST. The same reasons given in Section 5.2 for the reduced effectiveness of software prefetching on pointer-based data structures explain locality optimization's performance advantage at higher memory latencies in these applications. Once again, EM3D is an exception. EM3D performance with memory latency scaling is similar to the affine array and indexed array benchmarks. The same reasons given in Section 5.2 apply. It is worth noting that since the performance of the three pointer-chasing benchmarks differs, there is still room for further research to study more pointer-chasing applications to better characterize their behavior when software prefetching and locality optimizations are applied.

5.4 Combined Techniques

This section evaluates software prefetching and locality optimizations in combination. We created combined versions of our benchmarks in the following manner. For the affine array benchmarks,

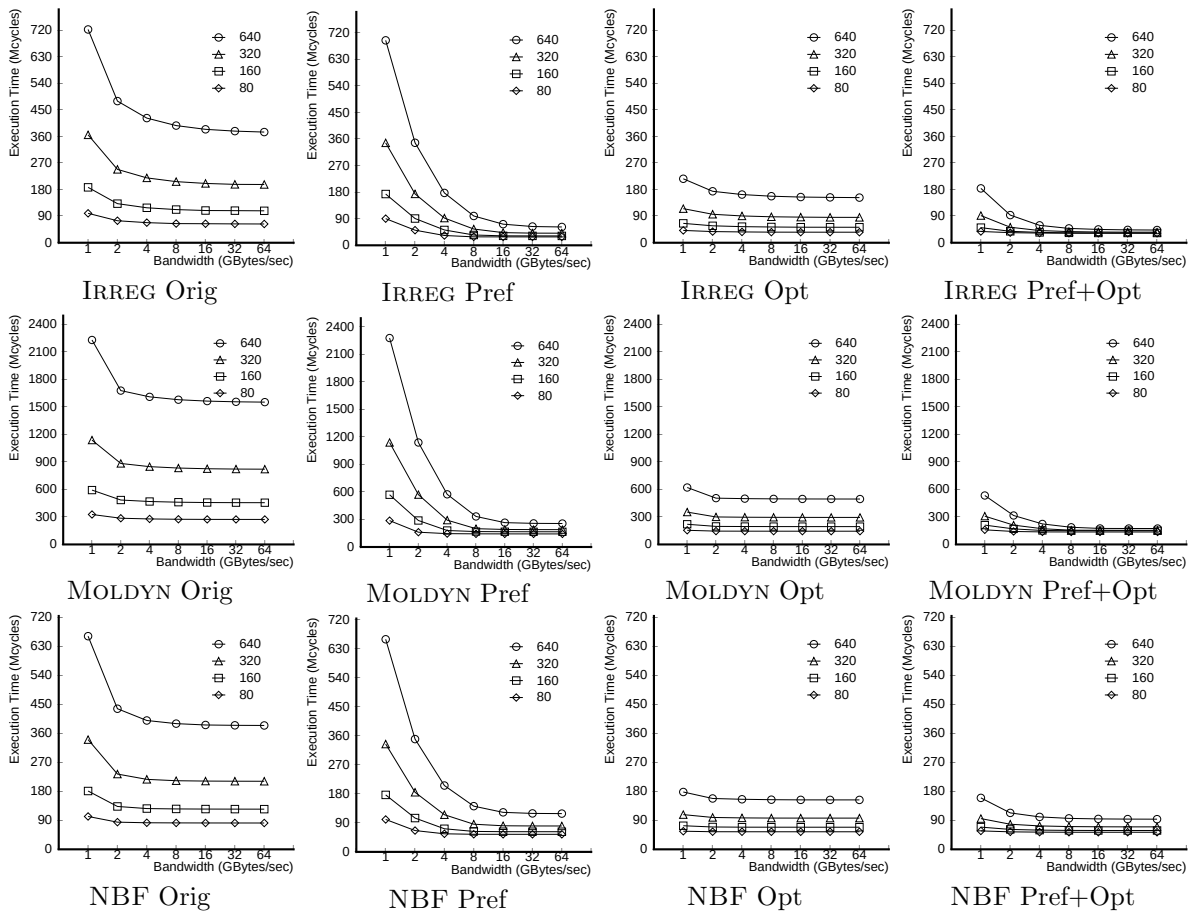

Figure 12: Execution time under both memory bandwidth and latency scaling for indexed array benchmarks with no optimizations (Orig), with software prefetching (Pref), with locality optimization (Opt), and with combined optimizations (Pref+Opt).

Latency	RB	Jacobi	MM	Irreg	Moldyn	NBF	Health	MST	EM3D
80	9	8 , 40	16	8 , 20 , 20 , 40	1, 1, 2	2	8	3	2
160	9	11, 80	28	12, 40 , 40 , 80	2, 2, 3	4	16	3	3
320	9	11, 156	52	24, 80 , 80 , 160	4, 4, 5	8	32	3	6
640	9	11, 308	104	44, 160, 160, 319	7, 7, 9	16	16	3	11

Table 4: Prefetch distances for the combined versions for all benchmarks versus latency in cycles.

we applied software prefetching to the innermost tiled loops. For the indexed array and pointer-chasing benchmarks, software prefetching and locality optimizations modify distinct parts of the code. Hence, for these programs, we simply merge the modified portions of the software prefetching and locality optimization program versions.

Table 4 reports the computed prefetch distances for the affine array, indexed array, and pointer-chasing benchmarks at different memory latencies. Note that for RB, the prefetch distance is fixed at 9 since the tile size is 9x10 whereas the computed prefetch distance is larger than 9. For our combined optimizations to function properly, we must limit the prefetch distance to the minimum of the prefetch distance and the length of the tile in order to prevent prefetching beyond the current

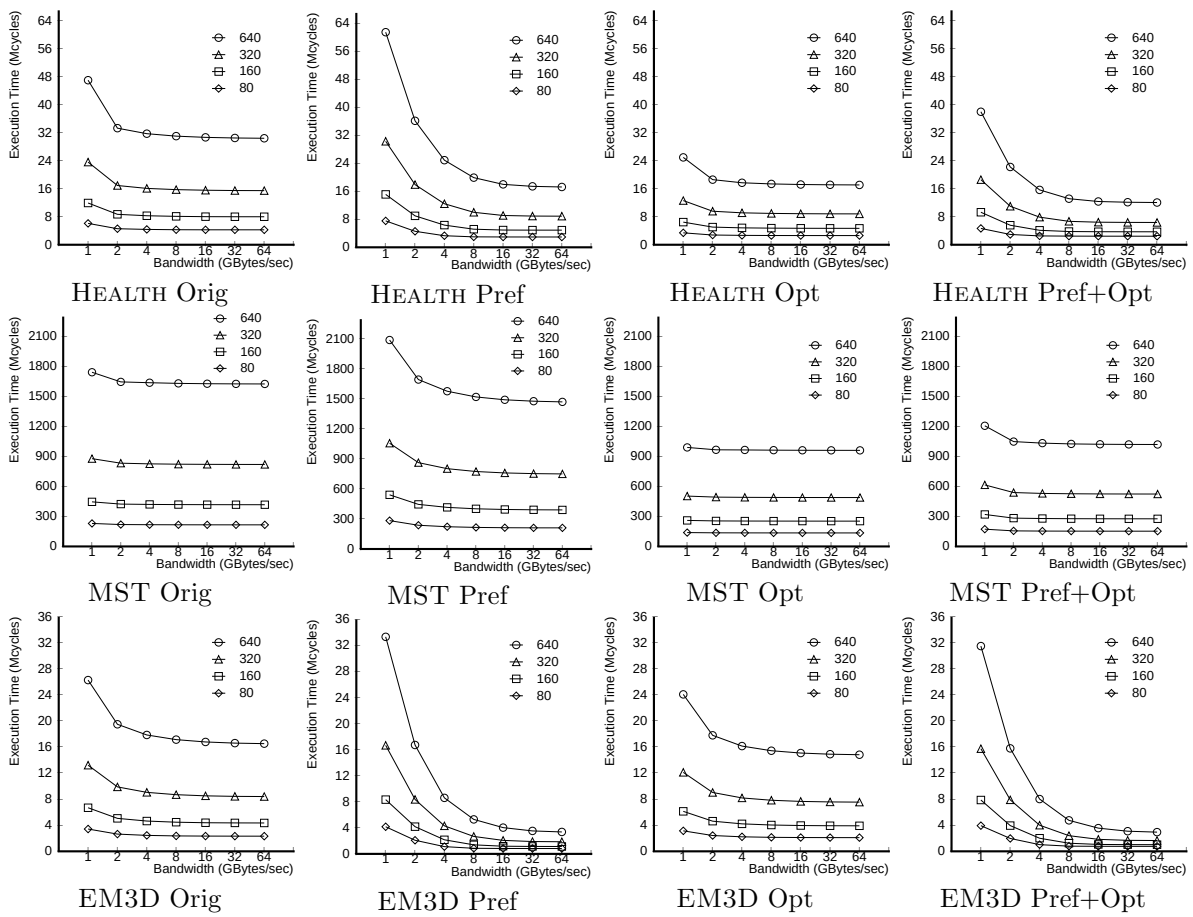

Figure 13: Execution time under both memory bandwidth and latency scaling for pointer-chasing benchmarks with no optimizations (Orig), with software prefetching (Pref), with locality optimization (Opt), and with combined optimizations (Pref+Opt).

computation tile. This effect also occurs in one of the loops from Jacobi for latencies higher than 80 cycles.

Results for the combined optimizations are reported in Figures 10, 11, 12, and 13 under "Pref+Opt." In Figure 14, we also summarize the average performance of each version of the program relative to memory bandwidth and latency. Performance is first normalized relative to the original program (with bandwidth of 1 Gbyte/sec and latency of 80 cycles), then averaged over all programs for each memory bandwidth or latency. Simulations show results vary depending on memory bandwidth and latency.

Software prefetching is very sensitive to available memory bandwidth. When bandwidth is very low, software prefetching increases overhead without reducing memory costs. The combined algorithm thus performs slightly worse than locality optimizations alone. In comparison, when memory latencies are very high, combining software prefetching and locality optimizations usually yields better performance than applying either one alone. As Figure 14 shows, combining is much better than prefetching, and only slightly better than locality optimizations.

Under certain conditions, combining techniques encounters high overhead compared to either technique alone. For the affine array benchmarks, tiling significantly reduces the number of iter-

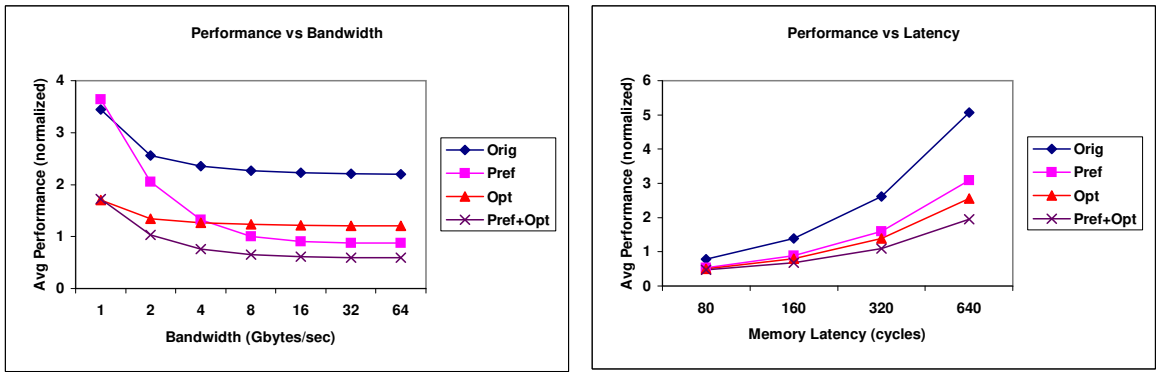

Figure 14: Comparing average performance for different versions of programs relative to memory bandwidth and latency. Performance is normalized relative to the original program with 1 Gbyte/sec bandwidth and 80 cycle latency.

ations in the innermost loop. When prefetching is applied to these short tiled loops, the software pipeline startup overhead incurred by prefetching becomes significant, reducing the amount of memory latency hidden. This effect is apparent in the high CPU overhead in the "Pref+Opt" versions of MM and Jacobi in Figure 10. Combining also inherits the overheads from both software prefetching and tiling, further reducing its performance relative to either techniques alone. For both indexed array and pointer-chasing benchmarks, software prefetching and locality optimizations modify different parts of the benchmark code. Software prefetching modifies the computation loops themselves while locality optimizations instrument the creation code semantics, or reorders data before entering the computation loop.

For pointer-chasing benchmarks, except for EM3D, combining always under-performs ccmalloc memory allocation alone at low memory bandwidths. The extra jump pointers and prefetch arrays required for pointer prefetching increase the demand for memory bandwidth, thus partially negating the reduced traffic benefits achieved by ccmalloc memory allocation in the combined version. The combined version also underperforms ccmalloc memory allocation at high memory bandwidths in MST. As described previously, software prefetching for the short list traversal loops in MST is ineffective; hence, combining software prefetching with ccmalloc memory allocation only adds overhead without reducing memory stalls. For EM3D, locality optimizations performs better than any other technique at low bandwidth. At higher bandwidth, combined is the best among all the techniques. As explained in Section 5.2, software prefetching is highly effective for EM3D. When combined with ccmalloc, the reduced memory traffic provided through better locality permits software prefetching to achieve even higher performance.

Finally, because combining exploits both latency tolerance and latency reduction, it is less sensitive to variations in memory bandwidth and latency than either technique in isolation. As shown in Figure 14, the combined version achieves the best performance on average for practically all memory bandwidths and latencies. Such robust performance is valuable when bandwidth and latency parameters on the target system are not available to the compiler, or when the compiler must produce a single optimized code for heterogeneous systems.

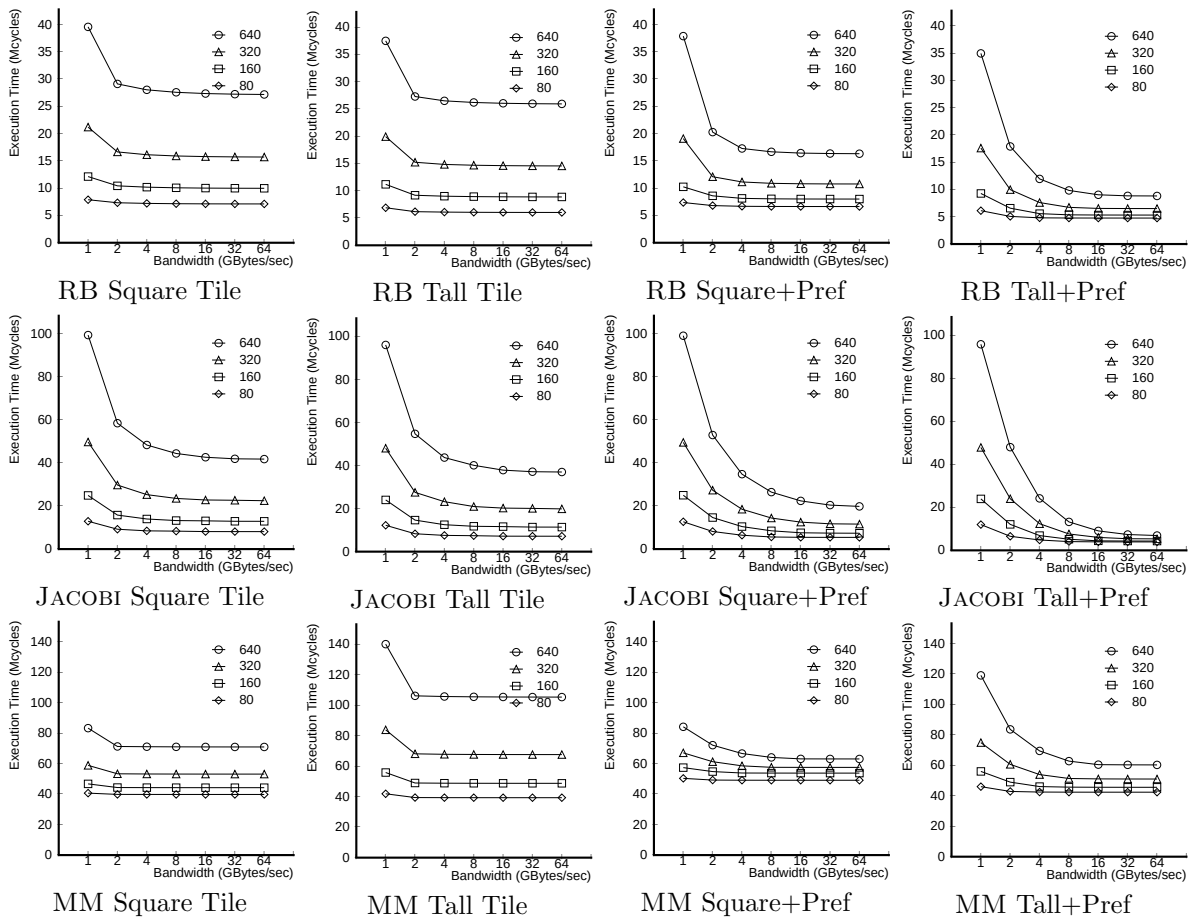

Figure 15: Comparing square tiles against tall tiles with and without prefetching.

6. Algorithm Enhancements

In addition to evaluating the effects of memory bandwidth and latency scaling on performance, our simulations also point out a number of ways to enhance both software prefetching and locality optimizations. This section presents several enhancements to better combine the two techniques and to enhance software prefetching for array codes in the presence of conflict misses. First, tiling is enhanced to combine more effectively with software prefetching. Then, padding which is normally used to reduce conflicts in tiling is applied to software prefetching to avoid prefetch thrashing. Finally, ccmalloc is used to reduce overhead in software prefetching for pointer-chasing codes.

6.1 Enhancing Tiling for Software Prefetching

One problem with combining tiling and software prefetching naively is the high startup overhead from prefetching short tiled loops, as described in Section 5.4. We can improve performance by modifying the original tiling algorithm. Our enhanced algorithm is identical to the one described in Section 4.1 except we select a larger innermost tile dimension size (*e.g. TI* in Figure 6). Our tiling heuristic uses the Euclidean GCD algorithm [14, 18] to generate a series of non-conflicting tile sizes. Although tiles with a square aspect ratio typically achieve the best cache utilization, we can bias the selection towards taller tiles with greater height to width aspect ratio. Such *tall*

Application	Square	Tall
RB	9×10	31×3
Jacobi	11×13	59×3
MM	33×23	83×9

Table 5: Tile sizes for square and tall-tile versions of the affine array benchmarks.

Latency	RB	Jacobi	MM
80	12	8 , 40	16
160	20	12, 80	28
320	31	24, 156	52
640	31	48, 308	104

Table 6: Prefetch distances for RB, Jacobi and MM with tall tiles versus latency in cycles.

tiles have more iterations in their innermost loop compared to square tiles, thus reducing startup overheads when used in combination with software prefetching. Note, however, we must take care not to choose tiles that are too tall. In the extreme case, having a tile size of $Y \times 1$ would result in mapping the problem back to the original computation order. Thus, extremely tall tiles negate the benefits that tiling provides.

Table 5 reports both square and tall tile sizes for the three affine array benchmarks. After choosing a new tall tile size and instrumenting tiling, we then instrument software prefetching to the innermost tiled loops. Table 6 shows the prefetch distances for the new tall tile loops. If the prefetch distance computed after the instrumentation with tall tiles exceeds the length of the tile, the prefetch distance is selected to be the minimum of the tile length and the computed prefetch distance, as discussed in Section 5.4.

Figure 15 presents the tall tile results with and without prefetching for MM, Jacobi, and RB, and compares them to the corresponding square tile results from Figure 11. Notice tall tiles and square tiles alone achieve similar performance. However, when combined with software prefetching, tall tiles significantly reduce the short-loop overheads suffered at high bandwidths when using square tiles, matching the performance of software prefetching alone from Figure 11. These simulation results demonstrate that tall tiles allow us to fully exploit the benefits of software prefetching and tiling simultaneously.

In addition, the combined tall tile and software prefetching techniques retain the robustness benefit described in Section 5.4. Figure 15 shows that at low bandwidth, the performance tracks tiling performance alone, while at high bandwidth, the performance tracks software prefetching performance alone. Hence, the enhanced combined technique shows more robustness to variations in the memory system parameters since the two techniques are more synergistic when the enhancement is applied.

6.2 Padding for Software Prefetching

While software prefetching can hide memory latency given sufficient memory bandwidth, conflict misses on prefetched data can degrade or even completely eliminate benefits. In our experiments, we found that prefetching for affine array codes may require array padding, particularly if the set associativity of the L2 cache is low. The problem is that for some applications and problematic data sizes, severe conflict misses may result, with all prefetched data being mapped to a small set of cache lines, as shown in the top of Figure 7. This problem is especially acute for affine accesses to arrays whose dimensions are near a multiple of the cache size since adjacent array elements will

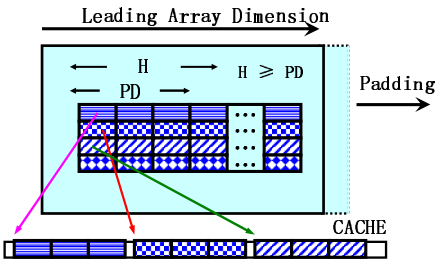

Figure 16: Illustrating the algorithm used for the choice of array padding.

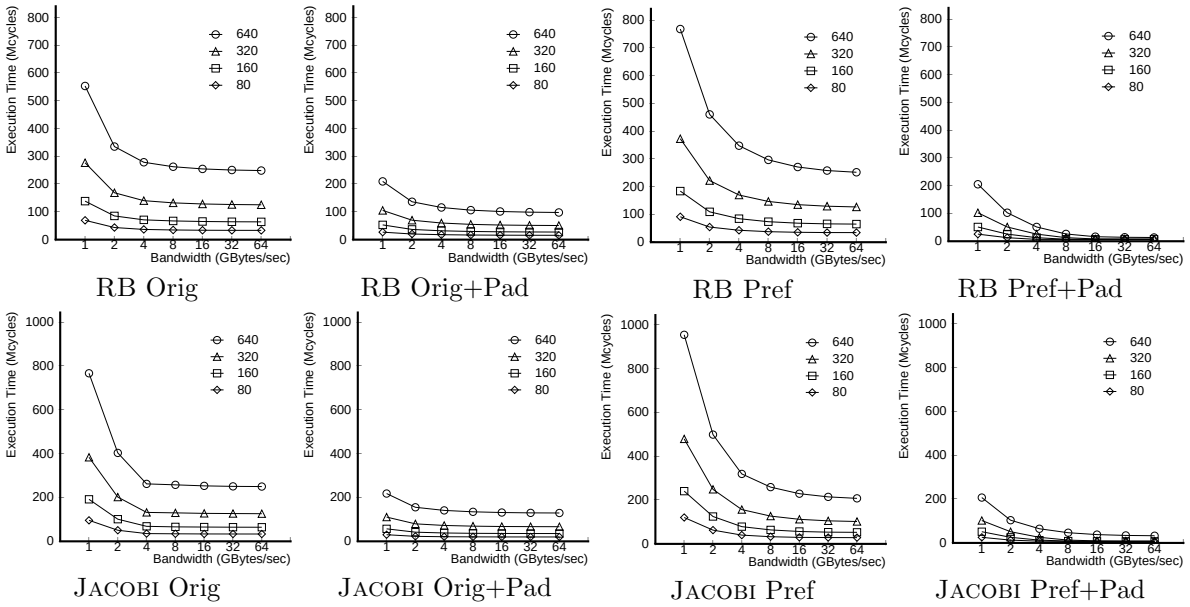

Figure 17: Execution results for padding with and without prefetching in Jacobi and RB.

conflict in cache. To alleviate the conflicts, array padding can be used to create a better mapping into the cache, as shown in the bottom half of Figure 7, allowing prefetched cache blocks to remain in cache until the processor references the data.

The appropriate amount of padding to apply to the array is computed as follows. First, the prefetch distance computed by the prefetching algorithm, as explained in Section 3 and as computed for our applications at the end of Section 5.1, is treated as the leading dimension of a tile. Second, the compiler uses the Euclidean GCD algorithm to determine whether cache conflicts will occur within a tile whose leading dimension is the prefetch distance, "PD." Padding is introduced incrementally to the leading array dimension until the Euclidean GCD algorithm gives a conflict-free tile size whose leading dimension is at least equal to or larger than "PD" [18, 19]. Figure 16 illustrates this algorithm, showing how the leading dimension of a tile "H" is related to the prefetch distance "PD," how padding is introduced along the leading array dimension, and how a group of PD prefetched array cache blocks do not conflict in the cache after padding.

For this padded problem size, prefetched cache blocks will stay in cache until they are referenced by the processor for the following reason. After issuing a prefetch, the processor will reference the prefetched cache block only after prefetch distance minus 1 ($PD - 1$) additional prefetches have been issued since each prefetch instruction prefetches data PD iterations in advance. So, there are at most PD prefetched cache blocks that have not been referenced by the processor at any one

moment in time. For the amount of padding computed, the Euclidean GCD algorithm ensures no conflicts will occur within a tile whose leading dimension is at least equal to the prefetch distance. Hence, we are guaranteed that there are no conflicts between the PD cache blocks that have been prefetched but not yet referenced by the processor.

Figure 17 presents experiments demonstrating the utility of combining array padding with prefetching. Versions of RB and Jacobi were created with and without both padding and prefetching. We used a 2-way set-associative L2 cache in these simulations for the purpose of illustration, since 4-way caches can eliminate conflicts in both benchmarks but this would not be the case for more complicated programs. We chose a problem size of 256×256×8. Such power-of-two problem sizes occur frequently in multigrid codes due to the need to use a series of meshes of increasing granularity. Based on the prefetch distance, our Euclidean algorithm chose to pad the array to 313×256×8 to eliminate conflicts.

The first two graphs in Figure 17 for RB and Jacobi show that padding alone applied to the original unoptimized code is capable of removing many of the conflict misses, and increases performance even with no prefetching applied. Figure 17 also shows that prefetching alone, in the graphs labeled "Pref," provides zero benefit due to the conflicts since conflicts evict prefetched cache lines before their use. In fact, performance degrades at low bandwidth due to fetching more data because of the conflicts. Once padding is applied, as shown in the graphs labeled "Pref+Pad," prefetching can improve performance beyond that achieved by padding alone. (Note, due to the change in the problem sizes for this particular enhancement, the results for Orig and Pref for both RB and Jacobi in Figure 17 are different from those in Figure 11).

6.3 ccmalloc and Prefetching

Software prefetching for pointer-chasing codes suffers high overhead to create and manage jump pointers, as described in Section 5.2. However, jump pointers may not be necessary when prefetching is combined with ccmalloc memory allocation. Since intelligent allocation places link nodes contiguously in memory, prefetch instructions can access future link nodes by simple indexing, just as for affine array accesses. Figure 9 shows the effect of ccmalloc on nodes linked together by pointers. From the right-hand part of the figure, it is intuitive that a compiler can insert prefetches for list nodes further down the list using the size of a node and the location of the first node. This approach, which we call *index prefetching* [1, 37], was originally proposed in [8]. With index prefetching, the jump pointers can be removed, thus eliminating all the overhead associated with jump pointer prefetching. To quantify this benefit, we created index prefetching versions for Health and MST, and show the results for these benchmarks in Figure 18. (We did not create an index prefetching version for EM3D, our third pointer-chasing benchmark, since it already achieves high performance with normal software prefetching as shown in Figures 10 and 13).

The upper portion of Figure 18 compares index prefetching to the original versions with prefetch arrays and ccmalloc allocation alone as well as in combination, assuming a memory latency of 80 cycles. The data shows index prefetching indeed eliminates most of the software overheads incurred by prefetch arrays, as we expected. As a result, index prefetching outperforms all other optimized versions at high memory bandwidths for both Health and MST. Index prefetching performs slightly worse than ccmalloc allocation alone at low bandwidth only, especially in MST, due to prefetching conditionally accessed link nodes, increasing memory bandwidth consumption.

While index prefetching reduces software overheads, it is not as effective in eliminating memory stalls as prefetch arrays for Health. In Health, many link nodes are deleted and re-inserted into

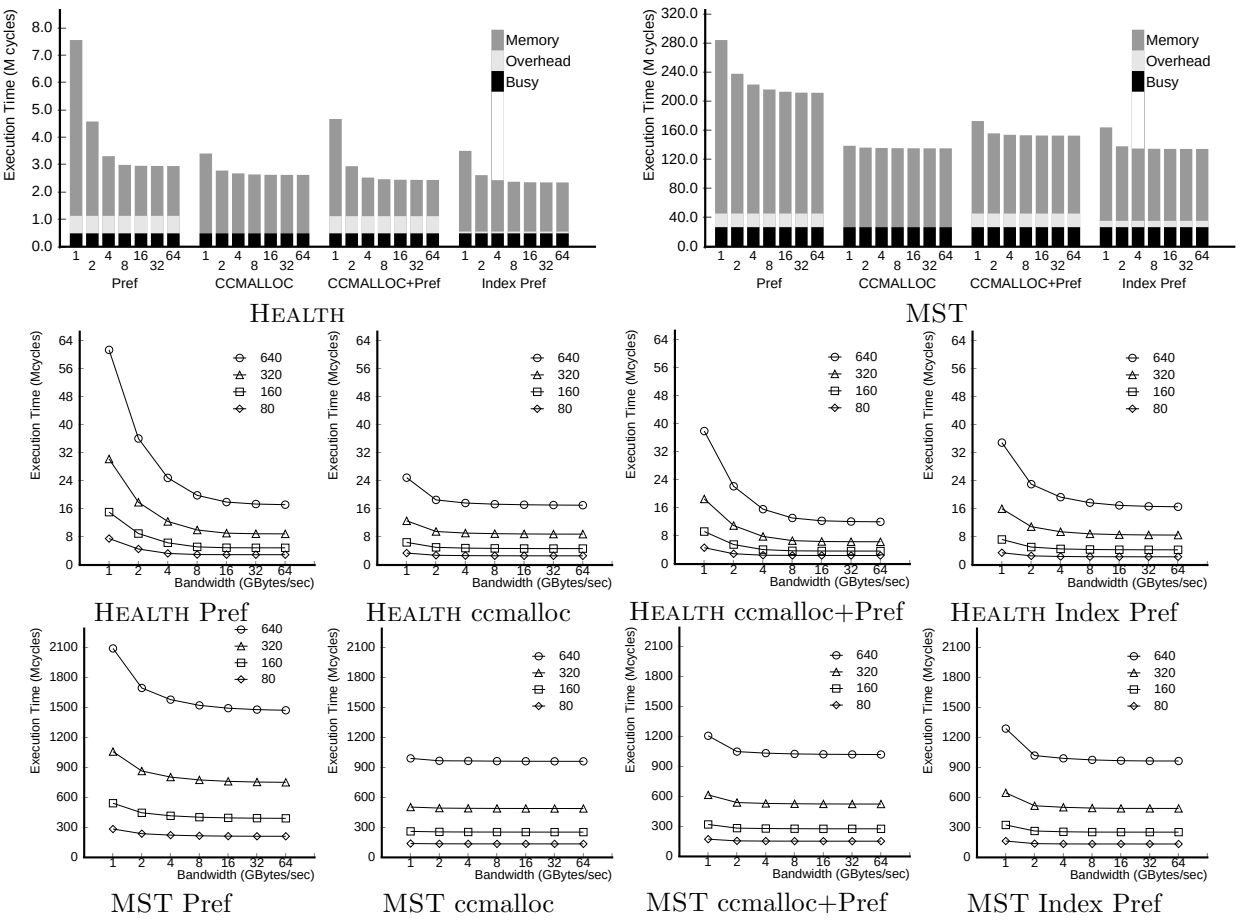

Figure 18: Comparing index prefetching (Index Pref) to prefetch arrays (Pref), ccmalloc memory allocation (ccmalloc), and combined optimizations (ccmalloc+Pref). Memory latency is fixed at 80 cycles in the top two graphs.

linked lists frequently. Contiguous allocation, and hence index prefetching, for such dynamic lists is useless since the layout of link nodes becomes random after a few delete and insert operations. As the upper portion of Figure 18 shows for Health, index prefetching hides less memory latency than prefetch arrays due to frequent delete and insert operations. At larger memory latencies, the increased memory stalls outweigh the reduced software overheads, so combining ccmalloc and prefetch arrays naively outperforms index prefetching.

7. Hardware Prefetching

While our research focuses on software techniques, there also exist several hardware techniques for addressing the memory bottleneck problem. *Hardware prefetching*, in particular, is a promising hardware technique that has received significant attention. In fact, research in this area is mature enough that many recent commercial microprocessors use simple prefetchers. Specifically, the Intel Pentium 4 [38] and IBM Power 5 [39] both employ *stride-based hardware prefetchers* [40, 41]. Given that hardware prefetching is becoming commonplace in high-end CPUs, an important question is

how do our software techniques compare against hardware prefetching? Furthermore, if applied in concert, how will our software techniques interact with hardware prefetching?

To address these questions, we augmented our simulator to model a stride-based hardware prefetcher, similar to the one used in [42]. Our stride prefetcher consists of a 256-entry stride table and 8 stream buffers. The stride table observes the post-L1 miss stream to detect strides on a per-load instruction basis. When a striding load is detected, the stride table allocates a stream buffer and commences prefetching using the observed stride. The processor checks both the L1 cache and stream buffers for each memory operation, and on a stream buffer hit, the prefetched block is moved into the L1 cache. To prevent thrashing, confidence estimation is used to guide stream buffer allocation. Each stride table entry contains a saturating counter that is incremented (decremented) whenever an L1 cache miss would have been correctly (incorrectly) predicted by the stride table. Each stream buffer also contains a saturating counter that is updated similarly. Stream buffer allocation occurs only when an inactive stream buffer is found, or when the allocating stride table entry has a higher confidence counter value than one of the active stream buffers.

Figure 19 shows the results of our hardware prefetching study in a format similar to Figure 10. As in Figure 10, we keep the memory latency fixed at 80 cycles, and vary the memory bandwidth between 1 and 64 GBytes/sec. Groups of bars represent versions optimized using different techniques. The bars labeled "Pref" employ software prefetching alone, and are identical to the corresponding bars in Figure 10. The bars labeled "Stride" and "Pref+Stride" employ stride-based hardware prefetching alone and in combination with software prefetching, respectively. Finally, the bars labeled "Pref+Opt" and "Stride+Opt" employ locality optimization in combination with software prefetching and stride-based hardware prefetching, respectively. Note, the "Pref+Opt" and "Stride+Opt" versions of the affine array benchmarks (*e.g.* RB, Jacobi, and MM) use the tall tile approach described in Section 6.1 since basic tiling does not perform well in combination with prefetching.

Comparing the "Stride" and "Pref" bars, we see hardware prefetching outperforms software prefetching slightly at high bandwidths for RB, Jacobi, MM, and MST. These benchmarks exhibit striding, so stride prefetching effectively removes the memory stalls. Also, hardware prefetching incurs zero runtime overhead since it does not consume processor resources to issue prefetches; hence, hardware prefetching enjoys a small performance advantage roughly equal to the runtime overhead incurred by software prefetching in these benchmarks. Hardware prefetching also outperforms software prefetching at low bandwidths for Health and EM3D. These benchmarks perform pointer chasing, so jump pointers and prefetch arrays must be instrumented to perform software prefetching. When memory bandwidth is scarce, fetching these extra pointers offsets the benefits of prefetching. Since hardware prefetching does not require extra pointers, it does not incur the additional memory traffic, allowing it to outperform software prefetching at low bandwidths.

While hardware prefetching holds a modest performance advantage in some cases, software prefetching outperforms hardware prefetching significantly in many other cases. In particular, Irreg, Moldyn, NBF, Health, and EM3D exhibit irregular access patterns that are non-striding. Stride prefetching fails to cover a significant portion of the cache misses in these benchmarks; hence, software prefetching achieves a significant gain over hardware prefetching, particularly at high bandwidths. In MM, Irreg, Moldyn, and MST, hardware prefetching frequently prefetches useless data, thus incurring higher memory traffic relative to software prefetching. This leads to a performance advantage for software prefetching at low bandwidths. Overall, our results indicate stride prefetching alone, as performed by hardware prefetchers in contemporary CPUs, cannot achieve the performance afforded by software prefetching on our benchmarks.

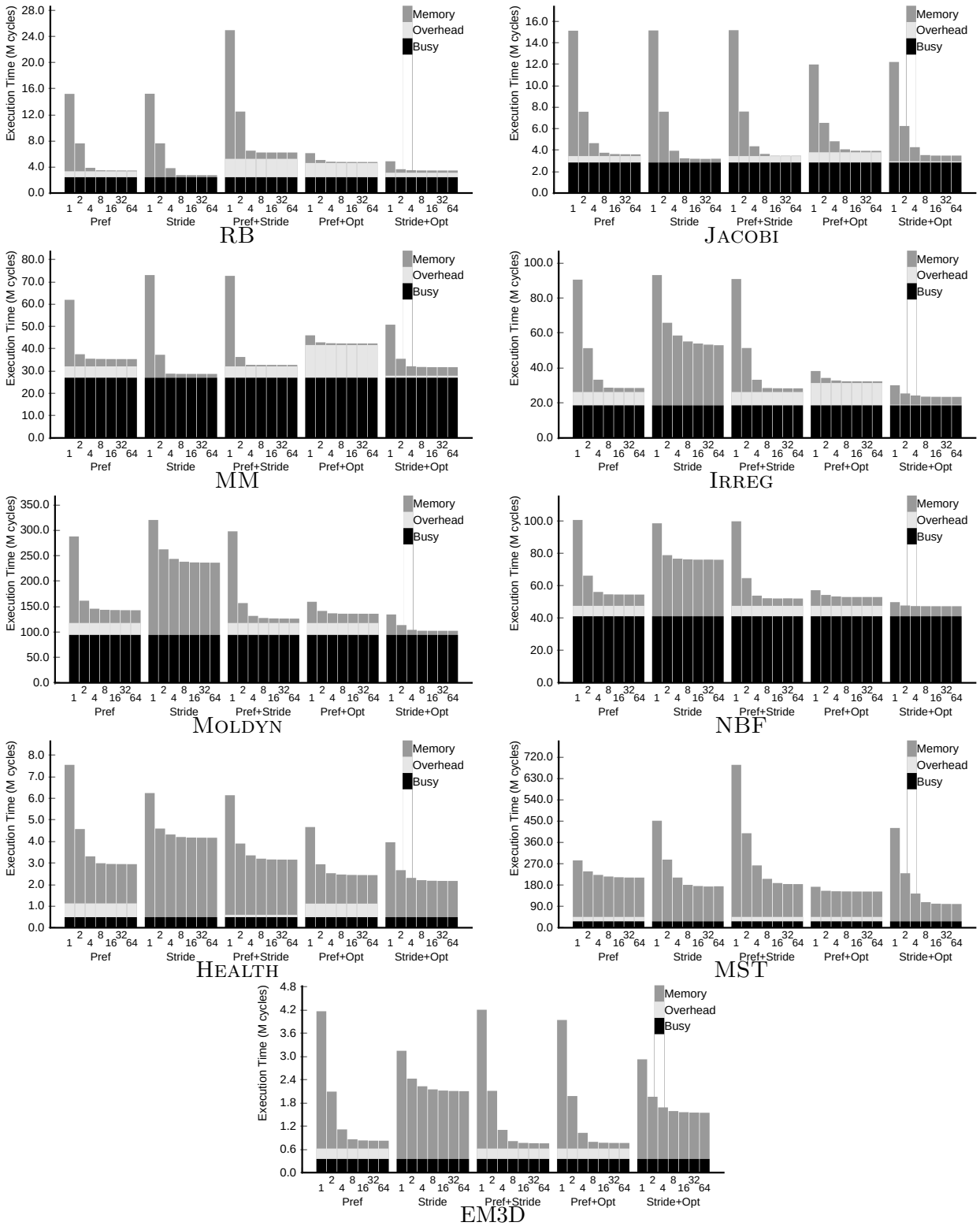

Figure 19: Execution time breakdown under memory bandwidth scaling with software prefetching (Pref), hardware prefetching (Stride), combined software prefetching and hardware prefetching (Pref+Stride), combined software prefetching and locality optimizations (Pref+Opt), and combined hardware prefetching and locality optimizations (Stride+Opt), all at 80 cycles of memory latency.

Comparing the "Pref+Stride" and "Pref" bars, we see the combination of hardware and software prefetching achieves similar performance to software prefetching alone. One exception is MST where hardware prefetching frequently fetches useless data, resulting in reduced performance at low bandwidths. However, aside from this case, the two techniques combine quite well. This result suggests compilers and programmers can apply software prefetching on systems with hardware prefetchers, and achieve good performance without worrying about their interaction. (Note, higher performance could probably be achieved if software prefetching is selectively disabled for striding memory references to reduce runtime overhead since these references can be prefetched by hardware prefetching. But evaluation of such an approach is beyond the scope of this study.)

Finally, comparing the "Pref+Opt" and "Stride+Opt" bars, we see the combination of hardware prefetching and locality optimization outperforms the combination of software prefetching and locality optimization in almost all cases. What's surprising is "Stride+Opt" is superior even for most of the indexed array and pointer-chasing benchmarks. As discussed above, these benchmarks exhibit irregular access patterns that are non-striding. However, locality optimizations (*e.g.* runtime access reordering for indexed arrays and CCMALLOC for pointer chasing) not only increase temporal reuse, but they increase spatial reuse as well. As a result, many of the irregular memory references exhibit enough striding after locality optimization that hardware prefetching becomes effective. Also, as discussed above, hardware prefetching does not consume processor resources to issue prefetches, resulting in a performance advantage over software prefetching due to the lower runtime overhead.

There are a few exceptional cases. For EM3D at high bandwidths, software prefetching still holds a significant performance advantage over hardware prefetching when combined with locality optimization. CCMALLOC is unable to improve the locality for one critical memory reference in EM3D, so stride prefetching fails to cover its cache misses while software prefetching covers them. And for MM and MST, the performance degradation in hardware prefetching due to fetching useless data exceeds the benefit of reduced prefetch overhead. So, "Pref+Opt" still holds a performance advantage over "Stride+Opt" at low bandwidths for these benchmarks. However, our primary conclusion remains that "Stride+Opt" outperforms "Pref+Opt" overall. This result indicates locality optimization, when applied in concert with prefetching, not only reduces memory traffic, but also enables stride-based hardware prefetching for benchmarks that do not normally exhibit striding.

8. Related Work

Our work is most similar to Saavedra *et al* [43], which evaluated unimodular transformations, tiling, and software prefetching for matrix multiply. Mowry *et al* [44] also evaluated software prefetching and tiling for two scientific applications. In comparison, this paper focuses on memory trends and quantifies their impact on software prefetching and locality optimizations. Prior work has considered a single technology point only. Furthermore, we examine three classes of applications requiring different types of optimizations to study the memory trend effects in a broader context. We also propose enhancements to address problems that arise when combining techniques. Finally, compared to [43] which used a cache simulator to evaluate performance, we use detailed execution-driven simulation of a modern processor.

Although relatively little work has compared prefetching and locality optimizations, a large body of work has studied memory latency tolerance and data locality techniques in isolation. Software prefetching for affine array accesses has been studied in [4, 5, 6]. Hardware prefetching techniques

studied in [40, 45, 41, 46, 47] are similarly limited to affine array accesses, but use hardware to identify the access pattern automatically. Prefetch engines for affine array accesses [48, 49, 50, 51] provide hardware support for prefetching, but rely on the programmer or compiler to identify the access pattern.

Prefetching for pointer-chasing traversals uses one of several possible approaches. The first approach inserts additional pointers, called *jump pointers*, into dynamic data structures to connect non-consecutive link elements [52, 7, 8, 11], as described in Section 3.3. Another approach uses only natural pointers for prefetching [53, 8, 9, 10]. These techniques prefetch pointer chains sequentially, but schedule each prefetch as early as possible to maximize memory latency overlap. A third approach uses a correlation-based predictor in hardware, also known as a *Markov predictor*, to predict link node addresses for prefetching [54, 55, 42]. A fourth approach uses a special allocation technique to allocate nodes contiguously in memory which enables indexed access to the link nodes. This approach was first proposed in [8] and is called *data linearization prefetching*. The index prefetching technique evaluated in Section 6.3 is identical to data linearization prefetching.

In addition to these pointer prefetching techniques, there have been other techniques more recently that target pointer-chasing and other irregular applications. One such technique is *pre-execution* [56, 57, 58, 59, 60, 61]. Pre-execution uses spare hardware contexts in a multithreaded processor to run one or more *helper threads* in front of the main computation. The helper threads trigger cache misses on behalf of the main thread, prefetching them into cache. Another recent technique is Content-Directed Prefetching (CDP) [62]. CDP observes data as it is moved from memory into the L2 cache, and issues prefetches for any values that resemble memory addresses in anticipation of their access by the processor. A less hardware-centric approach compared to CDP is Guided Region Prefetching (GRP) [63]. GRP relies on the compiler to convey hints to the memory system that identify pointer accesses, indexed array accesses, and accesses that exhibit spatial reuse. Special prefetch hardware triggers prefetches on L2 misses based on the compiler hints. Lastly, another recent technique is to migrate a prefetching mechanism into the memory system itself [64, 65]. Such *memory-side prefetchers* reduce the round-trip latency to main memory, increasing the throughput of serialized pointer chasing references [65], and can implement large correlation tables for pointer prefetching since they can place such tables in DRAM [64].

Aside from prefetching, another related memory latency tolerance technique is multi-threading. Multi-threaded architectures tolerate latency by means of context switching between different processes or threads of control. Context switches can be triggered by a particular event like a cache miss [66] (the MIT Alewife [67] is a machine that makes use of such an idea), or they can be made on every cycle [68]. More recently, researchers have investigated Simultaneous Multi-Threading [69] in which instructions from multiple threads can be chosen for execution within a single cycle on a multi-issue processor. One drawback of multi-threading techniques, however, is they cannot improve single-thread performance; they only increase overall processor throughput.

Burger *et al* [70] forecasted the increasing importance of memory latency tolerance techniques, such as prefetching and multi-threading, given the widening gap between processor and memory performance. In [70], they study the impact of increased latency tolerance via out-of-order execution on processor pin bandwidth, and concluded that pin bandwidth limitations will become as significant a bottleneck to processor performance in the future as memory latency. Compared to our work, Burger's study evaluated out-of-order execution only, and did not evaluate other memory latency tolerance techniques.

In addition to memory latency tolerance techniques, researchers have also attacked the memory bottleneck problem by improving data locality. Computation-reordering transformations such as

loop permutation and tiling are the primary optimization techniques [13]; loop fission (distribution) and loop fusion have also been found to be helpful [71].

Data layout optimizations such as padding and transpose have been shown to be useful in eliminating conflict misses and improving spatial locality [21]. Several cache miss estimation techniques have been proposed to help guide data locality optimizations [72, 13]. Tiling has been proven useful for linear algebra codes [14, 16, 13] and multiple loop nests across time-step loops [20]. In comparison we apply tiling to 3D stencil codes which cannot be tiled with existing methods. Rivera in [18, 19] studied existing tiling techniques [73] and devised techniques to tile 3D scientific computations.

Researchers have examined irregular computations mostly in the context of parallel computing, using run-time [26, 27, 28] and compiler [74] support to support accesses on message-passing multiprocessors. A few have also looked at techniques for improving locality [22, 23, 24, 25]. Metrics such as reference affinity have been used to guide algorithms for splitting data structures and regrouping arrays to improve spatial locality [75]. Strout *et al.* have examined models for determining which combination of run-time data and iteration reordering heuristics will result in the best locality for a given dataset [76, 77].

Few researchers have investigated data layout transformations for pointer-based data structures [78, 79]. Chilimbi *et al.* investigated allocation-time and run-time techniques to improve locality for linked lists and trees [30]. In this paper, we propose further extensions. Calder *et al.* use profiling to guide layout of global and stack variables to avoid conflicts [29]. Carlisle *et al.* investigate parallel performance of pointer-based codes in Olden [80].

9. Conclusion

Software prefetching and locality optimizations are two promising techniques for addressing the processor-memory performance gap. Our work evaluates the effectiveness of these software techniques for three classes of applications, and studies their interactions when applied in concert. We also study the impact of stride-based hardware prefetching on our software techniques.

Several conclusions can be drawn from our work. First, the relative effectiveness of software prefetching and locality optimizations depends on available memory bandwidth. For our array-based benchmarks, software prefetching outperforms locality optimizations at high memory bandwidths, while locality optimizations outperform software prefetching at low memory bandwidths. The equi-performance bandwidth is 2.41 GBytes/sec on today's memory systems, but will increase as memory latencies increase in the future. However, locality optimizations outperform software prefetching at all memory bandwidths and latencies for 2 out of 3 pointer-chasing benchmarks due to the reduced effectiveness of prefetching for pointer-based data structures. The one exception is EM3D for which software prefetching achieves high performance; hence, EM3D behaves similarly to our array-based benchmarks.

Second, combining software prefetching and locality optimizations inherits the merits of both techniques. Combining yields better performance than either software prefetching or locality optimizations alone when memory latency is very high since it exploits both. Combining is also more robust to changes in memory system parameters than either latency tolerance or latency reduction techniques in isolation. However, naively combining techniques does not outperform the best choice amongst software prefetching and locality optimizations alone at all bandwidths and latencies.

Third, the combined effectiveness of software prefetching and locality optimizations can be enhanced through new algorithms. For affine array benchmarks, tall-tile selection reduces prefetch startup overheads, allowing combining to outperform software prefetching and locality optimizations alone for practically all memory bandwidths and latencies. Also, padding can remove conflicts between prefetched data for affine array benchmarks, and is crucial when prefetching for problem sizes that suffer from cache conflicts. For pointer-chasing benchmarks, combining index prefetching and ccmalloc memory allocation can reduce prefetch overheads, but this is not effective when a large number of link nodes cannot be contiguously allocated, as in Health, or when ccmalloc allocation already gets most of the gain, as in MST.

Finally, stride-based hardware prefetching can outperform software prefetching for striding applications due to lower runtime overhead, but software prefetching significantly outperforms hardware prefetching for benchmarks exhibiting irregular access patterns. When combining hardware and software prefetching, performance similar to software prefetching alone is achieved, suggesting that software prefetching can be successfully applied on CPUs with stride prefetchers. Also, locality optimizations enable stride prefetching for benchmarks that do not normally exhibit striding, allowing combined hardware prefetching and locality optimization to outperform combined software prefetching and locality optimization, again due to lower runtime overhead.

On current memory systems, maintaining memory bandwidths of 1-4 Gbytes/sec is achievable. Thus, the simulation results most relevant to today's systems are those with bandwidth towards the low end in our bandwidth variation experiments. As processors become faster, the memory wall will increase, reducing available memory bandwidth relative to processor speed. Thus, locality optimizations should become more important in the future. At the same time, as processor speeds increase, memory latencies will increase into the 100s of cycles. Thus, the simulation results most relevant for future systems are those with latencies towards the high end in our latency variation experiments.

One possibility for dramatically increasing memory bandwidth is to switch to processor-in-memory (PIM) architectures [81]. For on-chip data, available memory bandwidth will be more like that towards the high end of our bandwidth variation experiments. Our results show such PIM systems should benefit significantly from software prefetching. However, even PIM systems will require locality optimizations to reduce accesses to off-chip data.

10. Future Work

We believe software and architecture support is needed to reduce the memory bottleneck for advanced microprocessors. In this paper, we demonstrated how prefetching and locality optimizations can be used to improve locality and performance for several types of applications. While our results are encouraging, much work remains to be done.

First, our results have mostly been achieved for individual kernels. These kernels are important because they are taken from larger programs, and make up the vast majority of processing time in these other workloads. However, we must still actually evaluate the effectiveness of our techniques for larger, more realistic programs.

Second, our locality optimizations are currently applied semi-automatically, only partially implemented in the compiler and run-time system. For these optimizations to be widely used, we must automate them as much as possible. Also, we hand-instrumented prefetching for all applications, a task that the compiler needs to automate. Hence, an important direction for future work is to implement our algorithms in a compiler and to re-perform our study.

Third, a somewhat surprising result from Section 7 is locality optimizations enable stride-based hardware prefetching for benchmarks that do not normally exhibit striding. Further investigation into combining stride prefetching and locality optimization using more benchmarks is an interesting direction for future work, particularly given the increasing number of commercial CPUs that employ stride prefetchers.

Finally, while our current study focuses on performance, another important consideration is power. We believe an extremely important and interesting future direction is to investigate the impact of the optimizations studied in this work on system power and energy dissipation. Given the importance of this direction, we have already begun work in this area [82]; however, much more work is required.

11. Acknowledgments

The authors would like to thank Gabriel Rivera for providing insightful discussions about the tiling and padding techniques, and for providing the affine array codes used in this paper. Also, we would like to thank Hwansoo Han for providing the indexed array codes used in this paper. Finally, we would like to thank Mrs. Inukai for reviewing the paper for us.

References

[1] A.-H. Badawy, A. Aggarwal, D. Yeung, and C.-W. Tseng, "Evaluating the Impact of memory system performance on software prefetching and locality optimizations," in *Proc. of the 15th Int'l Conference on Supercomputing*, (Sorrento, Italy), ACM, June 2001.

[2] W. A. Wulf and S. A. McKee, "Hitting the Memory Wall: Implications of the Obvious," *Computer Architecture News*, vol. 23, pp. 20–24, March 1995.

[3] T. Mowry and A. Gupta, "Tolerating latency through software-controlled prefetching in shared-memory multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 12, pp. 87–106, June 1991.

[4] D. Callahan, K. Kennedy, and A. Porterfield, "Software prefetching," in *Proc. of the 4th Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, (Santa Clara, CA), Apr. 1991.

[5] A. C. Klaiber and H. M. Levy, "An Architecture for Software-Controlled Data Prefetching," in *Proc. of the 18th Int'l Symp. on Computer Architecture*, (Toronto, Canada), pp. 43–53, ACM, May 1991.

[6] T. Mowry, "Tolerating latency in multiprocessors through compiler-inserted prefetching," *ACM Trans. Computer Systems*, vol. 16, pp. 55–92, Feb. 1998.

[7] M. Karlsson, F. Dahlgren, and P. Stenstrom, "A Prefetching Technique for Irregular Accesses to Linked Data Structures," in *Proc. of the 6th Int'l Conference on High Performance Computer Architecture*, (Toulouse, France), January 2000.

[8] C.-K. Luk and T. Mowry, "Compiler-based prefetching for recursive data structures," in *Proc. of the 7th Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, (Boston, MA), Oct. 1996.

[9] S. Mehrotra and L. Harrison, "Examination of a Memory Access Classification Scheme for Pointer-Intensive and Numeric Programs," in *Proc. of the 10th Int'l Conference on Supercomputing*, (Philadelphia, PA), ACM, May 1996.

[10] A. Roth, A. Moshovos, and G. S. Sohi, "Dependence Based Prefetching for Linked Data Structures," in *Proc. of the 8th Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[11] A. Roth and G. S. Sohi, "Effective Jump-Pointer Prefetching for Linked Data Structures," in *Proc. of the 26th Int'l Symp. on Computer Architecture*, (Atlanta, GA), May 1999.

[12] T. C. Mowry, "Tolerating Latency Through Software-Controlled Data Prefetching, PhD Thesis," tech. rep., Stanford University, March 1994.

[13] M. Wolf and M. Lam, "A data locality optimizing algorithm," in *Proc. of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, (Toronto, Canada), June 1991.

[14] S. Coleman and K. S. McKinley, "Tile size selection using cache organization and data layout," in *Proc. of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, (La Jolla, CA), June 1995.

[15] I. Kodukula, K. Pingali, R. Cox, and D. Maydan, "An experimental evaluation of tiling and shacking for memory hierarchy management," in *Proc. of the Int'l Conference on Supercomputing*, (Rhodes, Greece), June 1999.

[16] M. Lam, E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," in *Proc. of the 4th Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, (Santa Clara, CA), Apr. 1991.

[17] R. Panda, H. Nakamura, N. Dutt, and A. Nicolau, "Augmenting loop tiling with data alignment for improved cache performance," *IEEE Trans. on Computers*, vol. 48, pp. 142–149, Feb. 1999.

[18] G. Rivera and C.-W. Tseng, "A comparison of compiler tiling algorithms," in *Proc. of the 8th Int'l Conference on Compiler Construction*, (Amsterdam, The Netherlands), Mar. 1999.

[19] G. Rivera and C.-W. Tseng, "Tiling optimizations for 3D scientific computations," in *Proc. of SC'00*, (Dallas, TX), Nov. 2000.

[20] Y. Song and Z. Li, "New tiling techniques to improve cache temporal locality," in *Proc. of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, (Atlanta, GA), May 1999.

[21] G. Rivera and C.-W. Tseng, "Data transformations for eliminating conflict misses," in *Proc. of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, (Montreal, Canada), June 1998.

[22] I. Al-Furaih and S. Ranka, "Memory hierarchy management for iterative graph structures," in *Proc. of the 12th Int'l Parallel Processing Symp.*, (Orlando, FL), Apr. 1998.

[23] C. Ding and K. Kennedy, "Improving cache performance of dynamic applications with computation and data layout transformations," in *Proc. of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, (Atlanta, GA), May 1999.

[24] J. Mellor-Crummey, D. Whalley, and K. Kennedy, "Improving memory hierarchy performance for irregular applications," in *Proc. of the Int'l Conference on Supercomputing*, (Rhodes, Greece), June 1999.

[25] N. Mitchell, L. Carter, and J. Ferrante, "Localizing non-affine array references," in *Proc. of the Int'l Conference on Parallel Architectures and Compilation Techniques*, (Newport Beach , LA), Oct. 1999.

[26] R. Das, M. Uysal, J. Saltz, and Y.-S. Hwang, "Communication optimizations for irregular scientific computations on distributed memory architectures," *Journal of Parallel and Distributed Computing*, vol. 22, pp. 462–479, Sept. 1994.

[27] H. Han and C.-W. Tseng, "A comparison of locality transformations for irregular codes," in *Proc. of the 5th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, (Rochester, NY), May 2000.

[28] H. Han and C.-W. Tseng, "Improving locality for adaptive irregular codes," in *Proc. of the 13th Workshop on Languages and Compilers for Parallel Computing*, (White Plains, NY), Aug. 2000.

[29] B. Calder, C. Krintz, S. John, and T. Austin, "Cache-conscious data placement," in *Proc. of the 8th Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, (San Jose, CA), Oct. 1998.

[30] T. Chilimbi, M. Hill, and J. Larus, "Cache-conscious structure layout," in *Proc. of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, (Atlanta, GA), May 1999.

[31] E. M. Rajiv Gupta and Y. Zhang, *Profile Guided Code Optimizations, The Compiler Design Handbook: Optimizations and Machine Code Generation, Chapter 4.* CRC Press, 2002.

[32] D. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0," CS TR 1342, University of Wisconsin-Madison, June 1997.

[33] W. F. van Gunsteren and H. J. C. Berendsen, "GROMOS: GROningen MOlecular Simulation software," tech. rep., Laboratory of Physical Chemistry, University of Groningen, The Netherlands, 1988.

[34] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren, "Supporting dynamic data structures on distributed memory machines," *ACM Trans. on Programming Languages and Systems*, vol. 17, pp. 233–263, Mar. 1995.

[35] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, "LogP: Towards a Realistic Model of Parallel Computation," in *4th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, (San Diego, CA), pp. 1–12, IEEE, May 1993.

[36] S. Chandra, J. R. Larus, and A. Rogers, "Where is Time Spent in Message-Passing and Shared-Memory Programs," in *Proc. of the 6th Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, (San Jose, California), pp. 61–73, 1994.

[37] A.-H. Badawy, "Evaluating the Impact of Memory System Performance on Software Prefetching and Locality Optimizations," CS-TR 4392, University of Maryland, May 2002.

[38] http://www.intel.com/design/Pentium4/index.htm, "Intel Pentium 4 Processor." 2002.

[39] R. Kalla, "Power5: IBM's Next Generation POWER Microprocessor," in *Proc. of the 15th Symp. on High-Performance Chips (HOTCHIPS-15)*, (Stanford University, Palo Alto, CA), August 2003.

[40] T.-F. Chen and J.-L. Baer, "Effective Hardware-Based Data Prefetching for High-Performance Processors," *Trans. on Computers*, vol. 44, pp. 609–623, May 1995.

[41] J. W. C. Fu, J. H. Patel, and B. L. Janssens, "Stride Directed Prefetching in Scalar Processors," in *Proc. of the 25th Int'l Symp. on Microarchitecture*, pp. 102–110, December 1992.

[42] T. Sherwood, S. Sair, and B. Calder, "Predictor-Directed Stream Buffers," in *Proc. of the 33rd Int'l Symp. on Microarchitecture*, December 2000.

[43] R. 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 Int'l Parallel Processing Symp.*, pp. 39–45, April 1996.

[44] T. Mowry, M. Lam, and A. Gupta, "Design and Evaluation of a Compiler Algorithm for Prefetching," in *Proc. of the Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pp. 62–73, ACM, October 1992.

[45] J. W. C. Fu and J. H. Patel, "Data Prefetching in Multiprocessor Vector Cache Memories," in *Proc. of the 18th Int'l Symp. on Computer Architecture*, (Toronto, Canada), pp. 54–63, ACM, May 1991.

[46] 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 Int'l Symp. on Computer Architecture*, (Seattle, WA), pp. 364–373, ACM, May 1990.

[47] S. Palacharla and R. E. Kessler, "Evaluating Stream Buffers as a Secondary Cache Replacement," in *Proc. of the 21st Int'l Symp. on Computer Architecture*, (Chicago, IL), pp. 24–33, ACM, May 1994.

[48] T.-F. Chen, "An Effective Programmable Prefetch Engine for On-Chip Caches," in *Proc. of the 28th Symp. on Microarchitecture*, pp. 237–242, IEEE, 1995.

[49] C.-H. Chi, "Compiler Optimization Technique for Data Cache Prefetching Using a Small CAM Array," in *Proc. of the 1994 Int'l Conference on Parallel Processing*, pp. I–263–I–266, August 1994.

[50] T.-C. Chiueh, "Sunder: A Programmable Hardware Prefetch Architecture for Numerical Loops," in *Proc. of Supercomputing '94*, pp. 488–497, ACM, November 1994.

[51] O. Temam, "Streaming Prefetch," in *Proc. of Europar'96*, (Lyon,France), 1996.

[52] B. Cahoon and K. McKinley, "Data flow analysis for software prefetching linked data structures in java," in *Proc. of the 10th Int'l Conference on Parallel Architectures and Compilation Techniques*, (Barcelona, Spain), IEEE, September 2001.

[53] N. Kohout, S. Choi, D. Kim, and D. Yeung, "Multi-Chain Prefetching: Effective Exploitation of Inter-Chain Memory Parallelism for Pointer-Chasing Codes," in *Proc. of the 10th Int'l Conference on Parallel Architectures and Compilation Techniques*, (Barcelona, Spain), pp. 268–279, IEEE, September 2001.

[54] M. J. Charney and A. P. Reeves, "Generalized Correlation Based Hardware Prefetching," EE-CEG 95-1, Cornell University, February 1995.

[55] D. Joseph and D. Grunwald, "Prefetching using Markov Predictors," in *Proc. of the 24th Int'l Symp. on Computer Architecture*, (Denver, CO), pp. 252–263, ACM, June 1997.

[56] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen, "Speculative Precomputation: Long-range Prefetching of Delinquent Loads," in *Proc. of the 28th Int'l Symp. on Computer Architecture*, (Goteborg, Sweden), June 2001.

[57] D. Kim and D. Yeung, "Design and Evaluation of Compiler Algorithms for Pre-Execution," in *Proc. of the 10th Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, (San Jose, CA), pp. 159–170, ACM, October 2002.

[58] S. S. W. Liao, P. H. Wang, H. Wang, G. Hoflehner, D. Lavery, and J. P. Shen, "Post-Pass Binary Adaptation for Software-Based Speculative Precomputation," in *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, (Berlin, Germany), June 2002.

[59] C.-K. Luk, "Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors," in *Proc. of the 28th Int'l Symp. on Computer Architecture*, (Goteborg, Sweden), ACM, June 2001.

[60] A. Roth and G. S. Sohi, "Speculative Data-Driven Multithreading," in *Proc. of the 7th Int'l Conference on High Performance Computer Architecture*, pp. 191–202, January 2001.

[61] C. Zilles and G. Sohi, "Execution-Based Prediction Using Speculative Slices," in *Proc. of the 28th Int'l Symp. on Computer Architecture*, (Goteborg, Sweden), June 2001.

[62] R. Cooksey, S. Jourdan, and D. Grunwald, "A Stateless, Content-Directed Data Prefetching Mechanism," in *Proc. of the 10th Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, (San Jose, CA), pp. 1–12, October 2002.

[63] Z. Wang, D. Burger, K. S. McKinley, S. K. Reinhardt, and C. C. Weem, "Guided Region Prefetching: A Cooperative Hardware/Software Approach," in *Proc. of the 30th Int'l Symp. on Computer Architecture*, June 2003.

[64] Y. Solihin, J. Lee, and J. Torrellas, "Using a User-Level Memory Thread for Correlation Prefetching," in *Proc. of the 29th Int'l Symp. on Computer Architecture*, May 2002.

[65] C.-L. Yang and A. R. Lebeck, "Push vs. Pull: Data Movement for Linked Data Structures," in *Proc. of the Int'l Conference on Supercomputing*, (Santa Fe, NM), ACM, May 2000.

[66] K. Kurihara, D. Chaiken, and A. Agarwal, "Latency Tolerance through Multithreading in Large-Scale Multiprocessors," in *Proc. Int'l Symp. on Shared Memory Multiprocessing*, (Japan), IPS Press, April 1991.

[67] A. Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung, "The MIT alewife machine: Architecture and performance," in *Proc. of the 22nd Int'l Symp. on Computer Architecture (ISCA'95)*, pp. 2–13, 1995.

[68] B. J. Smith, "Architecture and Applications of the HEP multiprocessor Computer System," in *Real-*

Time Signal Processing IV, pp. 241–248, 1981.

[69] D. Tullsen, S. Eggers, and H. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," in *Proc. of the 22nd Int'l Symp. on Computer Architecture*, (Santa Margherita Ligure, Italy), pp. 392–403, ACM, June 1995.

[70] D. Burger, J. R. Goodman, and A. Kagi, "Memory Bandwidth Limitations of Future Microprocessors," in *Proc. of the 23rd Int'l Symp. on Computer Architecture*, (Philadelphia, PA), pp. 78–89, ACM, May 1996.

[71] K. S. McKinley, S. Carr, and C.-W. Tseng, "Improving data locality with loop transformations," *ACM Trans. on Programming Languages and Systems*, vol. 18, pp. 424–453, July 1996.

[72] S. Ghosh, M. Martonosi, and S. Malik, "Cache miss equations: An analytical representation of cache misses," in *Proc. of the Int'l Conference on Supercomputing*, (Vienna, Austria), July 1997.

[73] C. Douglas, J. Hu, M. Kowarschik, U. Rüde, and C. Weiss, "Cache optimization for structured and unstructured grid multigrid," *Electronic Trans. on Numerical Analysis*, vol. 10, pp. 21–40, 2000.

[74] H. Lu, A. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel, "Compiler and software distributed shared memory support for irregular applications," in *Proc. of the 6th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, (Las Vegas, NV), June 1997.

[75] Y. Zhong, M. Orlovich, X. Shen, and C. Ding, "Array regrouping and structure splitting using whole-program reference affinity," in *Proc. of the 2004 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'04)*, (Washington, DC), June 2004.

[76] M. M. Strout, L. Carter, and J. Ferrante, "Compile-time composition of run-time data and iteration reorderings," in *Proc. of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2003.

[77] M. Strout and P. Hovland, "Metrics and models for reordering transformations," in *Proc. of the Second ACM SIGPLAN Workshop on Memory System Performance (MSP 2004) held in conjunction with (PLDI'04)*, (Washington, DC), June 2004.

[78] Y. Zhang and R. Gupta, "Data compression transformations for dynamically allocated data structures," in *Proc. of the Int'l Conference on Compiler Construction*, pp. 14–28, 2002.

[79] D. N. Truong, F. Bodin, and A. Seznec, "Improving cache behavior of dynamically allocated data structures," in *Proc. of the 8th IEEE International Conference on Parallel Architecture and Compilation Techniques (PACT'98)*, (Paris, France), pp. 322+, October 1998.

[80] M. Carlisle, A. Rogers, J. Reppy, and L. Hendren, "Early experiences with Olden," in *Proc. of the 6th Workshop on Languages and Compilers for Parallel Computing*, (Portland, OR), Aug. 1993.

[81] G. Bell, R. Sites, W. Dally, D. Ditzel, and Y. Patt, "Architects Look to Processors of Future," *MICROPROCESSOR REPORT, MICRODESIGN RESOURCES*, vol. 10, Aug. 1996.

[82] A.-H. Badawy, D. Yeung, and C.-W. Tseng, "The Effects of Software Prefetching and Locality Optimizations on System Energy and Power," *Poster in the Research Review Day at UMD*, March 03.