Evaluating the Impact of Memory System Performance on Software Prefetching and Locality Optimizations

Abdel-Hameed A. Badawy[†], Aneesh Aggarwal[†], Donald Yeung[†], and Chau-Wen Tseng[‡]
[†]Electrical and Computer Engineering Dept. and [‡]Computer Science Dept.
University of Maryland, College Park
{absalam,aneesh,yeung}@eng.umd.edu, and tseng@cs.umd.edu

ABSTRACT

Software prefetching and locality optimizations are techniques for overcoming the speed gap between processor and memory. In this paper, we evaluate the impact of memory trends on the effectiveness of software prefetching and locality optimizations for three types of applications: regular scientific codes, irregular scientific codes, and pointer-chasing codes. We find 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 2.5 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.

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* [46] 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.

This research was supported in part by NSF Computer Systems Architecture grant CCR-0093110 and NSF CAREER Awards CCR-0000988 and ASC-9625531.

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 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 compare the efficacy of software prefetching and locality optimizations for three types of data-intensive 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.

```
// Affine Array Accesses        // Indexed Array Accesses        // Pointer-Based Structures
// (3D Jacobi Kernel)           // (Molecular Dynamics)          // (Linked List Traversal)
A(N,N,N),B(N,N,N)               X(M),Y(M),E(2,N)                 struct node {val, next} *ptr, *list;
do k=2,N-1                      do t = 1, time                   while (...) {
do j=2,N-1                        if (recalc)                      ptr->next = malloc(node);
do i=2,N-1                          E(...) = ...                   ptr = ptr->next;
  A(i,j,k) = 0.16667 *            do i = 1, N                      ptr->val = ... ;
   (B(i-1,j,k)+B(i+1,j,k)+          d = X(E(1,i))-X(E(2,i))      }
    B(i,j-1,k)+B(i,j+1,k)+          force = d**(-7)-d**(-4)      while (ptr->next) {
    B(i,j,k-1)+B(i,j,k+1))          Y(E(1,i)) += force             ...
                                    Y(E(2,i)) += -force            ptr = ptr->next;
                                                                 }
```

Figure 1: Example affine array, indexed array, and pointer-chasing codes.

In the rest of this paper, we will look at three memory access patterns, examine software prefetching and locality optimizations for each type of application, present experimental evaluations for each application, develop improved algorithms, and finally, discuss related work and conclude.

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.

2.1 Affine Array Accesses

The most basic memory access pattern is affine (linear) accesses to multidimensional arrays. For instance, consider the Jacobi code in Figure 1, typically used in multigrid solvers for partial differential equations (PDEs). The value of a point in A is calculated as the average of values of neighboring points in all three dimensions of B. This *stencil* pattern is repeatedly applied to each point of A, resulting in a smoother solution. All array accesses are affine because array subscripts are combinations of loop index variables with constant coefficients and additive constants. In practice, there are no coefficients and small additive constants are used. These programs are also called *regular* codes because memory access patterns are so regular and well defined.

Affine array accesses are common in dense-matrix linear algebra and finite-difference PDE solvers, as well as database scans and image processing. A major feature of affine array accesses is that they allow memory access patterns to be entirely computed at compile time, assuming array dimension sizes are known. This allows both software prefetching and locality transformations to be calculated 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 1, which calculates forces between pairs of atoms in a molecule. Accesses to the index array E are affine, striding through the array sequentially. However, accesses to arrays X and Y are indexed by the contents of E. Such programs are also called *irregular* because their memory accesses are not fixed. Irregular accesses typically make it difficult to keep data in cache, resulting in many cache misses and low performance.

Indexed array accesses arise in several scientific application domains as 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, for example, data structures are irregular because they model the positions of particles and their interactions. Unfortunately, those irregular computations have poor temporal and spatial locality, and do not utilize processor caches efficiently. Unlike applications with affine accesses, compile-time transformations alone cannot improve locality because the values of the index array are not known at compile time.

2.3 Pointer-Chasing Accesses

Finally, we also consider pointer programs which dynamically allocate memory and use pointer-based data structures such as linked lists, n-ary trees, and other graph structures. For example, consider the list traversal code in Figure 1 which creates a singly-linked list using a data structure *node* containing a pointer to the next element on the list. To traverse this list, the program must determine the pointer value stored in each node.

As with indexed array accesses, programs utilizing pointers have irregular memory access patterns that cannot be determined at compile time. Additionally, the next node cannot be traversed until the pointer stored in the current node is found, sequentializing memory accesses. These programs are thus known as *pointer-chasing* codes. Pointer-chasing codes occur in many application domains, including scientific programs which use advanced data structures.

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. In this section, we briefly describe the software prefetching techniques previously proposed for prefetching different types of memory references.

3.1 Affine Array Prefetching

To perform software prefetching for affine array references commonly found in scientific codes, the well-known compiler algorithm for inserting prefetches proposed by Mowry and Gupta [32] is followed. In this algorithm, locality analysis is used to determine which array references are likely to suffer cache misses. The cache-missing memory references are then isolated by performing loop unrolling and loop peeling transformations. Finally, prefetch instructions are inserted for the isolated cache-missing references. Each inserted prefetch is properly scheduled such that there exists ample time between the initiation of the prefetch and the consumption of the data by the processor (known as the *prefetch distance*) to overlap the latency of the memory access.

3.2 Indexed Array Prefetching

Indexed array accesses, of the form $A(B(i))$, are common in irregular scientific codes. The prefetch algorithm for indexed array accesses, originally proposed in [30], is similar to the algorithm for affine array prefetching. The main difference lies in how prefetch requests are scheduled. In affine array prefetching, each prefetch is scheduled early enough to tolerate the latency of a single cache miss. For indexed array references, the memory indirection between the index array and data array requires more sophisticated prefetch scheduling. If both the index array and the data array references miss in the cache, then the memory latency of two serialized cache misses, rather than just one, must be tolerated. Hence, the prefetch algorithm must schedule the prefetch for the index array access two cache miss times prior to the iteration that consumes the data, and schedule the prefetch for the data array one cache miss time prior to the iteration that consumes the data.

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. In contrast, the memory operations performed for pointer traversal must dereference a series of pointers, a purely sequential operation. The sequentiality of pointer chasing prevents conventional prefetching techniques from overlapping cache misses suffered along a pointer chain, thus limiting their effectiveness.

Jump pointer prefetching [41, 25] 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 away) 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 2 illustrates a while loop that has been instrumented with jump pointer prefetching. Jump pointer prefetching, however, cannot prefetch the first prefetch distance number of link elements 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* [21]. In this technique, an array of prefetch pointers is added to every linked list to point to the first prefetch distance number of link elements. Hence, prefetches can be issued through the memory addresses in the prefetch arrays before traversing each linked list to cover the early nodes, as illustrated in Figure 2.

```
// Prefetching a linked list using
// prefetch arrays and jump pointers
for (i=0; i < PD; i++) {
   prefetch(the_list.prefetch_array[i]);
}
ptr = the_list.head;
while (ptr->next) {
   prefetch(ptr->jump);
   ...
   ptr = ptr->next;
}
```

Figure 2: Prefetch array and jump pointers code.

In addition to inserting the prefetch instructions, both jump pointer prefetching and prefetch arrays require insertion of code to create and maintain the prefetch pointers as the data structure is modified (not shown in Figure 2).

4. LOCALITY OPTIMIZATIONS

Software prefetching tries to hide memory latency while retaining the original program structure. Another alternative is to reduce memory costs by changing the computation order and data layout of the program at compile and run times. These *locality optimizations* try to improve *data locality*, the ability of an application to reuse data in the cache [45]. 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 data-intensive applications which we are investigating.

4.1 Tiling for Affine Accesses

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* (or blocking), which uses loop transformations to form small blocks of loop iterations which are executed together to exploit data locality [43, 45]. Figure 3 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 [38].

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 [23]. This effect is shown in Figure 4. Previous research found *tile size selection* and *array padding* can be applied to avoid conflict misses in tiles. Tile-size-selection algorithms carefully select tile dimensions tailored to individual array dimensions so that no conflicts occur [11]. Array padding expands leading array dimensions, increasing the range of non-conflicting tile shapes [36] and improving the performance of tiled codes over a range of problem sizes [35, 38]. In this paper, we apply a combination of both algorithms to tile both 2D linear algebra and 3D PDE solvers [37, 38].

4.2 Reordering for Indexed Accesses

Index arrays arise in scientific applications such as sparse mesh PDE solvers and molecular dynamics codes, where the access pattern is determined at run time. Unfortunately, index arrays cause data to be accessed in an irregular manner, making spatial locality (reuse of data on a cache line) unlikely when the data is larger than the cache.

Researchers have discovered recently that run-time data and computation transformations can improve the locality of irregular computations [1, 13, 28, 29]. Because computations are typically commutative, 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 approach developed for message-passing machines [12].

In this paper, we apply efficient partitioning algorithms to the input data to bring reuse closer together, then follow up by lexicographically sorting loop iterations based on data access patterns, using algorithms specified elsewhere [17, 18]. Our partitioning algorithm works by viewing data accesses as a graph, then applying a series of graph coarsening passes where connected data is put into the same cluster. The resulting programs achieve much better use of processor caches.

4.3 Memory Allocation For Pointers

Pointer-based programs frequently suffer from poor locality. They are also notoriously difficult to analyze and transform because of their reliance on pointers and heap-allocated recursive data structures. Researchers recently have developed *cache-conscious* heap allocation and transformation techniques to improve locality for pointer-based programs [3, 9]. Algorithms include run-time tree optimization routines which place parent nodes with child nodes for improved locality, and coloring when placing tree nodes to avoid conflict with the root.

```
// Tiled 3D Jacobi
A(N,N,N),B(N,N,N)
do kk=2,N-1,TK     // TI x TJ x TK 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) )
```

Figure 3: Tiled 3D Jacobi example.

Of particular interest is CCMALLOC, a customized memory allocator which allocates memory in a location near to a user-specified address. A heuristic which proved effective reserves space for future allocation requests when allocating new blocks of data [9]. 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. In this paper, we applied this optimization to our pointer-chasing benchmark codes.

5. EXPERIMENTAL EVALUATION

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

5.1 Methodology

Our experimental evaluation employs 7 benchmarks, representing the 3 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. MATMULT multiplies two matrices, REDBLACK performs a 3D red-black successive-over-relaxation, and JACOBI performs a 3D Jacobi relaxation. Both JACOBI and REDBLACK are frequently found in multigrid PDE solvers, such as MGRID from the SPEC/NAS benchmark suite. The next two applications perform indexed array accesses. IRREG is an iterative PDE solver for an irregular mesh, while MOLDYN is abstracted from the non-bonded force calculation in CHARMM, a key molecular dynamics application used at NIH to model macromolecular systems. Finally, the last two applications perform pointer-chasing accesses. HEALTH simulates the Columbian health care system, and MST computes a minimum spanning tree. Both pointer-chasing codes are from the OLDEN benchmark suite [39].

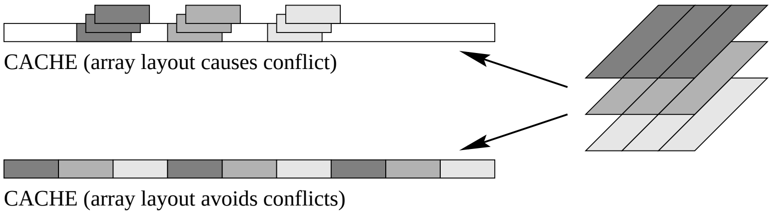


Figure 4: Example of conflict misses under two array layouts.

Application	Problem Size	Access Pattern
MATMULT	200x200 matrices	Affine array
JACOBI	200x200x8 grid	Affine array
REDBLACK	200x200x8 grid	Affine array
IRREG	14K node mesh	Indexed array
MOLDYN	13K molecules	Indexed array
HEALTH	5 levels, 500 iters	Pointer-chasing
MST	1024 nodes	Pointer-chasing

Table 1: Benchmark summary.

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 a detailed architectural simulator.

Our simulator is based on the SimpleScalar tool set [2] and models 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. 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 input data sets required for simulation.

To study the sensitivity of our software prefetching and locality optimization techniques to available memory bandwidth, we modified the SimpleScalar simulator to model bus contention across the L2-memory bus, 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), and varied the L2-memory latency from 80 to 640 cycles. These parameters capture characteristics of future architectures, where processors will be much faster than large DRAM memories. For all experiments, transfers across the L1-L2 bus incur a 7-cycle latency, and experience no contention.

5.2 Varying Memory Bandwidth

We evaluate the performance of software prefetching and locality optimizations under memory bandwidth scaling. In Figure 5, we plot execution time along the y-axis, and vary 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 on average by 46% and 42%, respectively, over unoptimized codes. Comparing the techniques, we see two major differences. First, software prefetching suffers more overhead due to prefetch and related address computation instructions. Tiling incurs some overhead for the extra levels of loops, but this is minimal. We do not include preprocessing overhead for data reordering because it can be amortized over many loop iterations [18].

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 the 5 array-based benchmarks, software prefetching outperforms locality optimizations at high memory bandwidths, while locality optimizations outperform software prefetching at low memory bandwidths.

To quantify this effect, Table 2 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

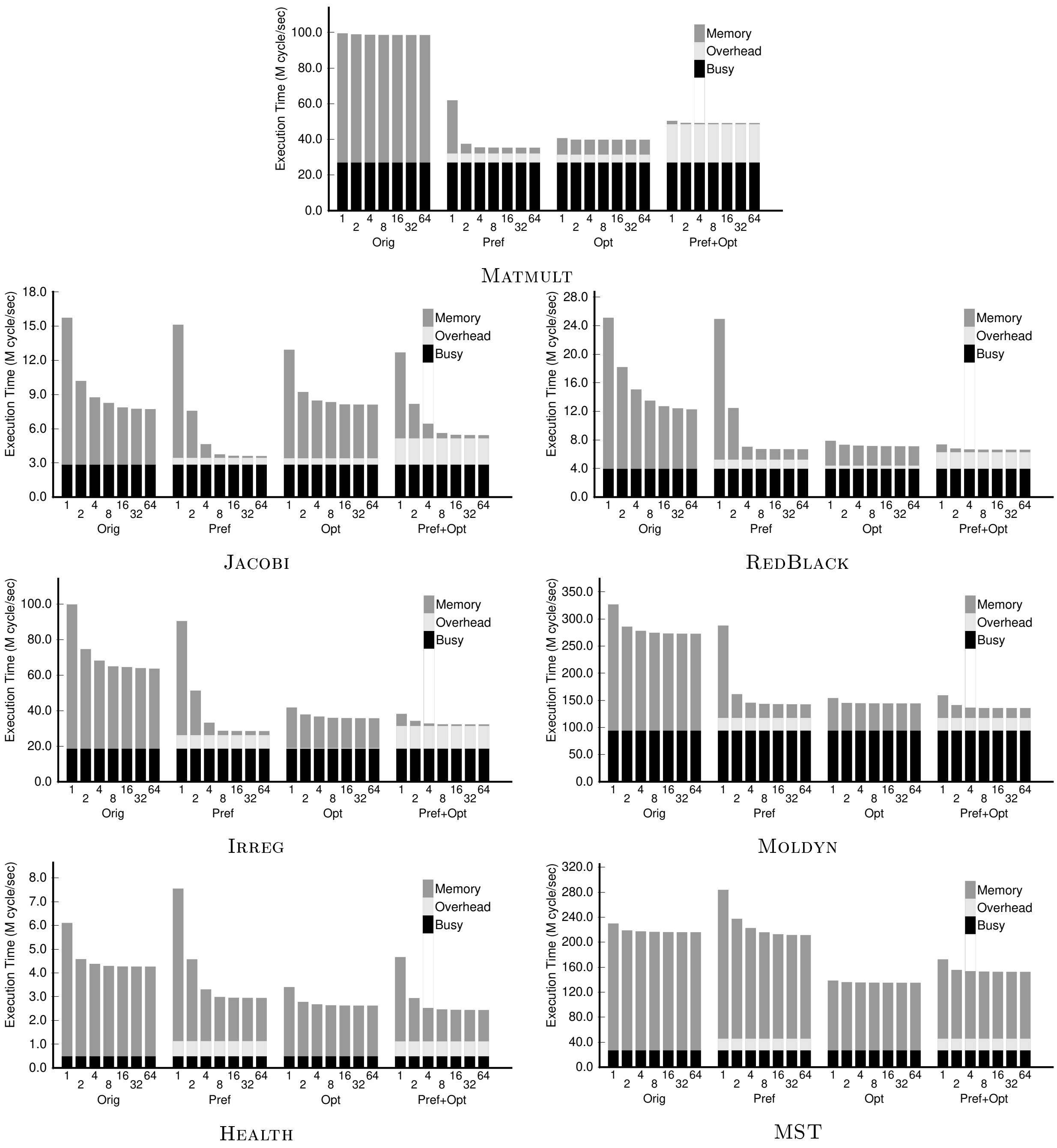


Figure 5: Execution time breakdown under memory bandwidth scaling with no optimizations (Orig), with software prefetching (Pref), with locality optimization (Opt), and with combined optimizations (Opt+Pref). Memory latency is fixed at 80 cycles.

Latency	Matmult	Jacobi	RedBlack	Irreg	Moldyn	Average
80	1.84	1.57	2.97	2.80	3.51	2.54
160	2.84	1.89	3.45	2.93	2.86	2.79
320	3.82	2.05	3.72	3.04	2.99	3.12
640	4.68	2.10	3.90	3.20	3.31	3.44

Table 2: Equi-performance bandwidths for 80, 160, 320, and 640-cycle memory latencies. The last column reports the average over the 5 benchmarks. All memory bandwidths are in Gbytes/sec.

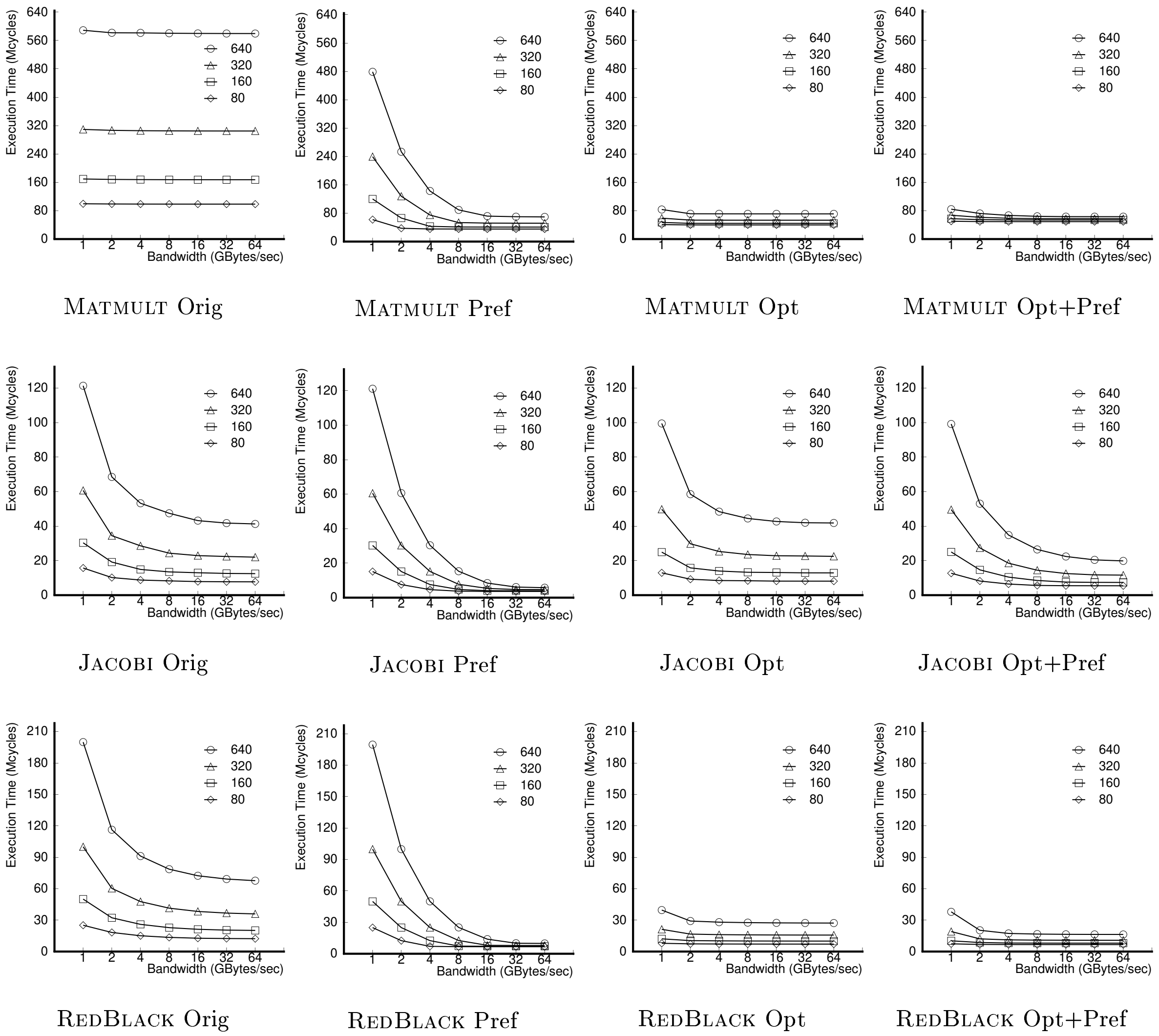


Figure 6: Execution time under both memory bandwidth and latency scaling for affine array benchmarks with no optimizations (Orig), with software prefetching (Pref), with locality optimization (Opt), and with combined optimizations (Opt+Pref).

lower memory bandwidths favor locality optimizations. For an 80-cycle memory latency corresponding to the data in Figure 5, Table 2 shows the average equi-performance bandwidth is 2.54 Gbytes/sec. 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.

For the pointer-chasing benchmarks, locality optimization outperforms software prefetching at all memory bandwidths. 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 5 shows, software prefetching in HEALTH and MST performs worse than the original code at low memory bandwidths.

5.3 Varying Memory Latency

Figures 6 and 7 evaluate software prefetching and locality optimizations under memory latency scaling. Similar to Figure 5, 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 ver-

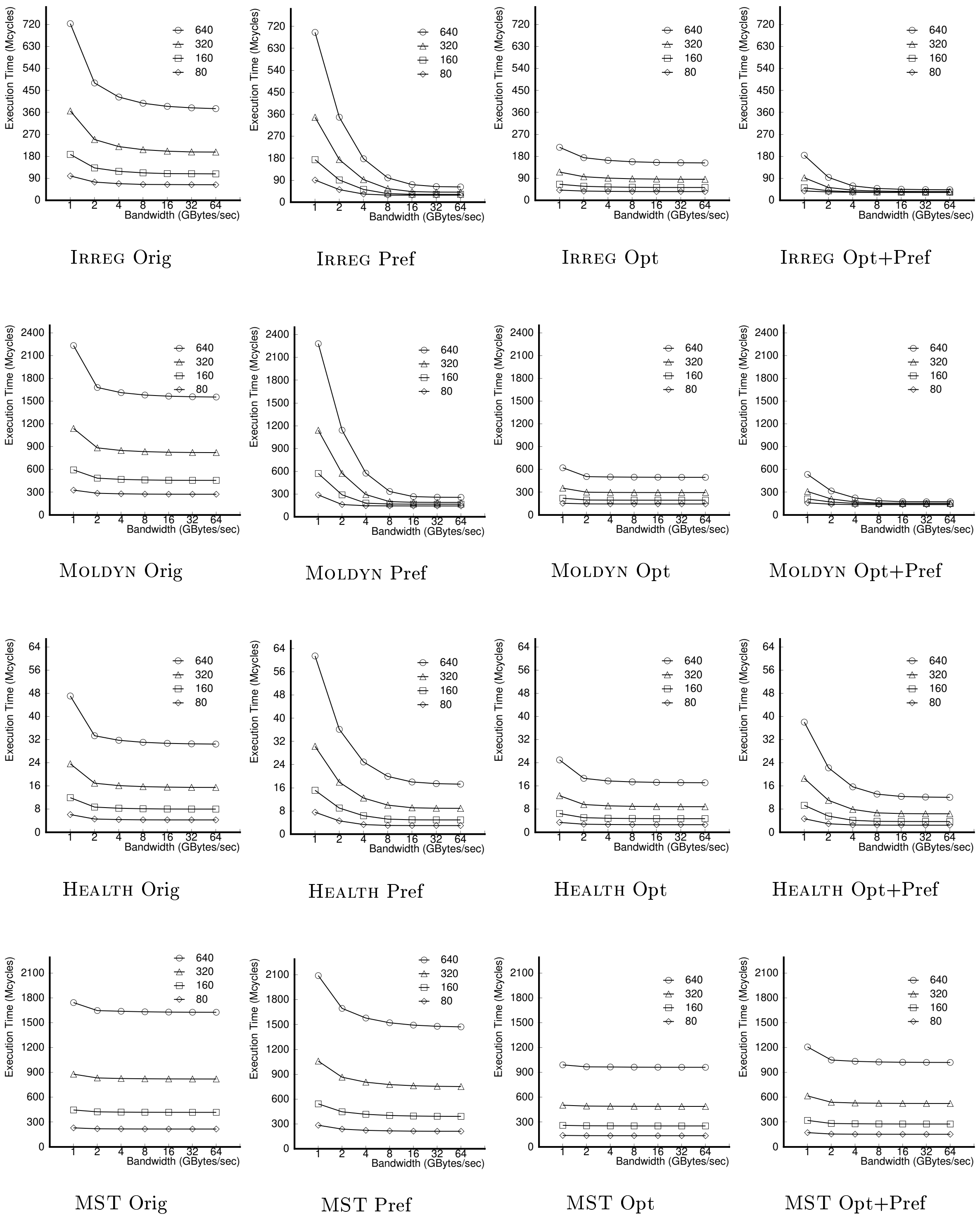


Figure 7: Execution time under both memory bandwidth and latency scaling for indexed array and pointer-chasing benchmarks with no optimizations (Orig), with software prefetching (Pref), with locality optimization (Opt), and with combined optimizations (Opt+Pref).

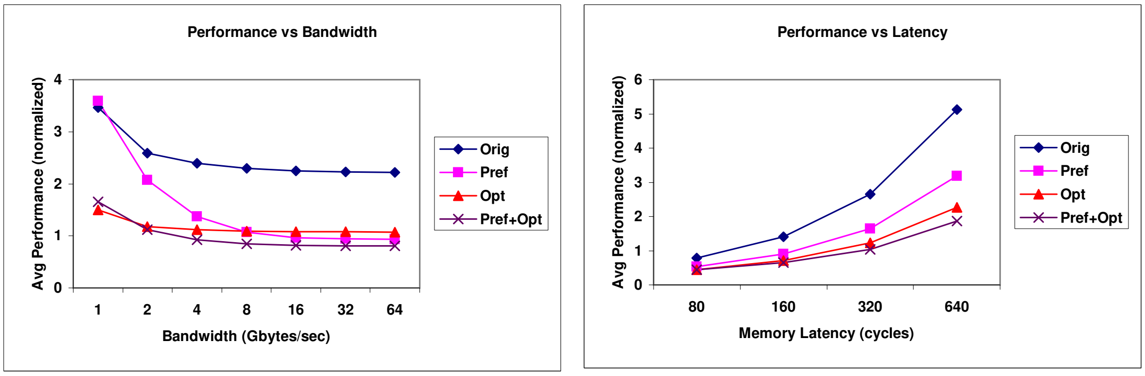


Figure 8: Comparing average performance for different versions of programs relative to memory bandwidth and latency. Performance is normalized relative to the original program with 1 Gbyte/sec bandwidth and 80 cycle latency.

sion 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 increase 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. Table 2 shows equi-performance bandwidths generally increase with memory latency. Consequently, on future systems with high memory latencies, greater memory bandwidth will be required before software prefetching demonstrates a performance advantage over locality optimizations.

For the pointer-chasing benchmarks, locality optimization outperforms software prefetching at all memory latencies and bandwidths. 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.

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, 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. Results are reported in Figures 5, 6, and 7, under "Pref+Opt."

In Figure 8 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 8 shows, combining is much better than prefetching, and only slightly better than locality optimizations.

Under certain conditions, combining techniques encounters high overhead. For the affine array benchmarks, tiling significantly reduces the number of iterations 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 MATMULT and JACOBI in Figure 5. Combining also inherits the overheads from both software prefetching and tiling, further reducing its performance relative to software prefetching alone.

For pointer-chasing benchmarks, combining always underperforms 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.

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. 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.

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.

6.1 Tiling and Prefetching

One problem with combining tiling and software prefetching naively is the high startup overhead from prefetching short tiled loops. We can improve performance by modifying the tiling algorithm to select tiles with more iterations in the innermost loop. Our tiling heuristic uses the Euclidean GCD algorithm [11, 37] to generate a series of non-conflicting tile shapes. 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 tiles* have more iterations in their innermost loop compared to square tiles, thus reducing startup overheads when used in combination with software prefetching. Table 3 reports both square and tall tile sizes for our 3 affine array benchmarks.

Application	Square	Tall
MATMULT	33 × 23	83 × 9
JACOBI	11 × 13	59 × 3
REDBLACK	9 × 10	31 × 3

Table 3: Tile sizes for square and tall-tile versions of the affine array benchmarks.

Figure 9 presents the tall-tile results with and without prefetching for MATMULT, JACOBI, and REDBLACK, and compares them to the corresponding square-tile results from Figure 6. 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 6. These simulation results demonstrate that tall tiles allow us to fully exploit the benefits of software prefetching and tiling simultaneously.

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 the same cache lines. 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 conflict in cache.

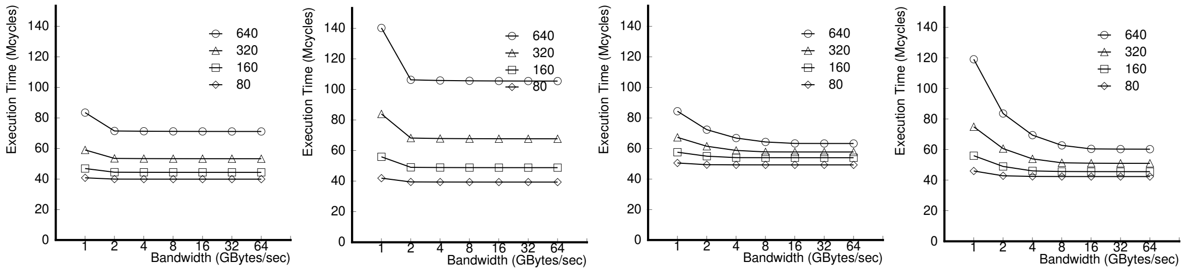
Compilers can avoid this problem and pad arrays to avoid prefetch conflicts [36, 37], even if loops are actually tiled. The approach we employ is to treat the distance between the prefetch data and the actual data as the "height" of a tile, with the variable references determining the tile width. Compiler analysis can then use the Euclidean GCD algorithm to determine whether cache conflicts will occur within this tile, padding leading array dimensions until conflicts are eliminated [37, 38]. This ensures that prefetched data will be able to stay in cache until they are used by the processor.

Figure 10 presents experiments demonstrating the utility of combining array padding with prefetching. Versions of the program JACOBI were created with and without both padding and prefetching. We used a 2-way associative L2 cache in these simulations for the purpose of illustration, since 4-way caches can eliminate conflicts in JACOBI but not more complicated programs. We chose a problem size of 256×256×8. These 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.

Simulation results show JACOBI experiences many cache conflicts which array padding can eliminate, improving performance. Software prefetching alone does not help, since conflicts evict prefetched cache lines before their use. Once padding is applied, prefetching can improve performance beyond that achieved by padding alone.

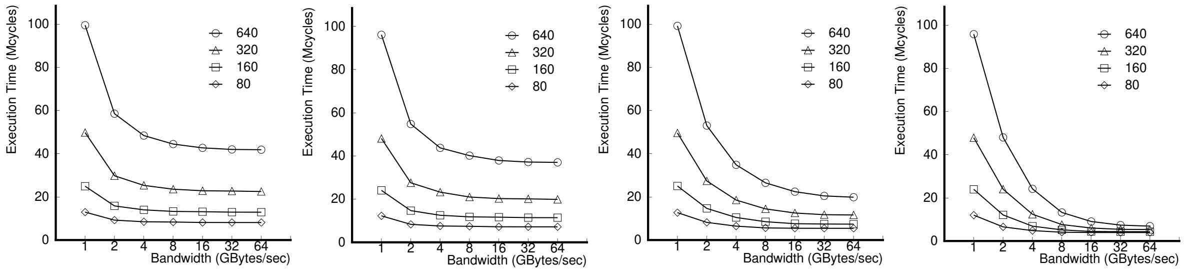
6.3 CCMALLOC and Prefetching

Software prefetching for pointer-chasing codes suffers high overhead to create and manage jump pointers. 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 indexing, just as for affine array accesses. This approach, which we refer to as *index prefetching*, was originally pro-



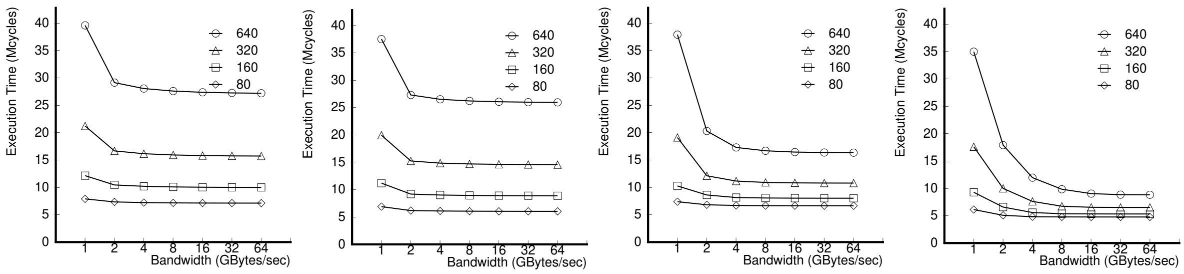
Square Tile Tall Tile Square Tile+Pref Tall Tile+Pref

MATMULT



Square Tile Tall Tile Square Tile+Pref Tall Tile+Pref

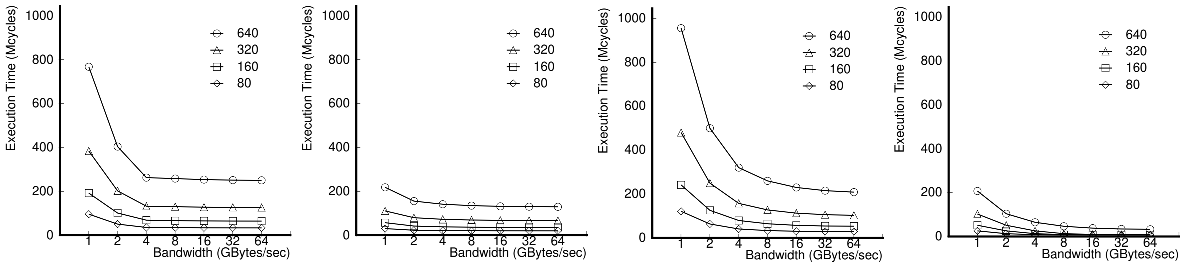
JACOBI



Square Tile Tall Tile Square Tile+Pref Tall Tile+Pref

REDBLACK

Figure 9: Comparing square tiles against tall tiles with and without prefetching.



Orig Orig+Pad Pref Pref+Pad

Figure 10: Padding for prefetching in Jacobi.

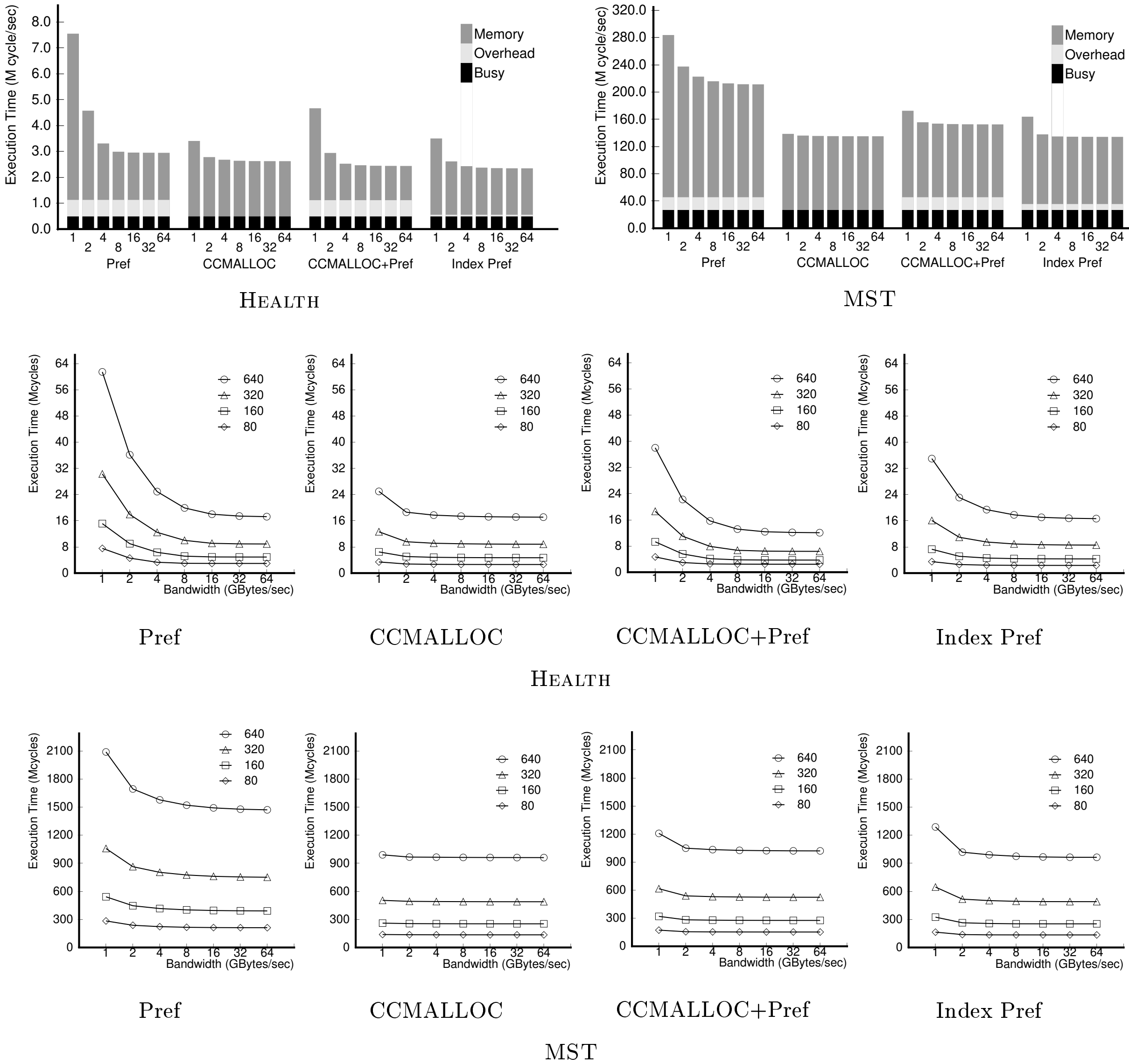


Figure 11: Comparing index prefetching (Index Pref) to prefetch arrays (Pref), CCMALLOC memory allocation (CCMALLOC), and combined optimizations (CCMALLOC+Pref). In the top two graphs, memory latency is fixed at 80 cycles.

posed in [25]. 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. Results are shown in Figure 11.

The upper portion of Figure 11 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 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 though, especially in MST, due to prefetching of link nodes that are conditionally accessed, 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 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-insert operations. As the upper portion of Figure 11 shows for Health, index prefetching hides less memory latency than prefetch arrays due to frequent delete-insert operations. At larger memory latencies, the increased memory stalls outweigh the reduced software overheads, and combining CCMALLOC and prefetch arrays naively outperforms index prefetching.

7. RELATED WORK

Our work is most similar to Saavedra *et al* [42], which evaluated unimodular transformations, tiling, and software prefetching for matrix multiply. Mowry *et al* [33] 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 3 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 [42] which used a cache simulator to evaluate performance, we use detailed execution-driven simulation of a modern processor.

Although relatively little work has compared software prefetching and locality optimizations, a large body of work has studied the techniques in isolation. Software prefetching for affine array accesses has been studied in [31, 22, 4]. Hardware prefetching [7, 34, 15, 14, 20] is similarly limited to affine array accesses, but uses hardware to identify the access pattern automatically. Prefetch engines for affine array accesses [44, 6, 10, 8] 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 four approaches. The first approach inserts additional pointers, called *jump pointers*, into dynamic data structures to connect non-consecutive link elements [21, 41, 25], as described in Section 3.3. The second approach uses only natural pointers for prefetching [40, 27, 25]. These techniques prefetch pointer chains sequentially, but schedule each prefetch as early in the loop iteration as possible to maximize memory latency overlap. The third approach uses a hardware table, called a *Markov predictor* [19], to predict link node addresses for prefetching. Finally, the 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 [25] and is called *data linearization prefetching*. The index prefetching technique evaluated in Section 6.3 is identical to data linearization prefetching.

Data locality has been recognized as a significant performance issue for modern processor architectures. Computation-reordering transformations such as loop permutation and tiling are the primary optimization techniques [45], loop fission (distribution) and loop fusion have also been found to be helpful [26].

Data layout optimizations such as padding and transpose have been shown to be useful in eliminating conflict misses and improving spatial locality [36]. Several cache miss estimation techniques have been proposed to help guide data locality optimizations [16, 45]. Tiling has been proven useful for linear algebra codes [23, 45, 11] and multiple loop nests across time-step loops [43]. In comparison we apply tiling to 3D stencil codes which cannot be tiled with existing methods.

Researchers have examined irregular computations mostly in the context of parallel computing, using run-time [12] and compiler [24] support to support accesses on message-passing multiprocessors. A few have also looked at techniques for improving locality [1, 13].

Few researchers have investigated data layout transformations for pointer-based data structures. Chilimbi *et al.* investigated allocation-time and run-time techniques to improve locality for linked lists and trees [9]. In this paper, we propose further extensions. Calder *et al.* use profiling to guide layout of global and stack variables to avoid conflicts [3]. Carlisle *et al.* investigate parallel performance of pointer-based codes in Olden [5].

8. CONCLUSION

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 2.5 GBytes/sec on today's memory systems, but will increase as memory latencies increase in the future. However, locality optimizations outperform software prefetching for the pointer-chasing benchmarks at all memory bandwidths and latencies due to the reduced effectiveness of prefetching for pointer-based data structures.

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.

Finally, 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.

With current processor speeds, maintaining memory bandwidths of 1-4 Gbytes/sec is probably achievable. The simulation results most relevant are thus those with bandwidth towards the low end. As processors become faster, the memory wall will increase, reducing available memory bandwidth relative to processor speed. Locality optimizations should become thus more important. Similarly, as processor speeds increase, memory latencies are likely to increase past 80 cycles, making the results of our simulation of higher memory latencies more relevant.

Architects might switch to processor-in-memory (PIM) architectures to increase memory bandwidth dramatically. For on-chip data, available memory bandwidth will be more like that towards the high end of our simulations. Our experiments 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.

9. 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.

10. REFERENCES

[1] I. Al-Furaih and S. Ranka. Memory hierarchy management for iterative graph structures. In *Proceedings of the 12th International Parallel Processing Symposium*, Orlando, FL, April 1998.

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

[3] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, CA, October 1998.

[4] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, Santa Clara, CA, April 1991.

[5] M. Carlisle, A. Rogers, J. Reppy, and L. Hendren. Early experiences with Olden. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[6] T.-F. Chen. An Effective Programmable Prefetch Engine for On-Chip Caches. In *Proceedings of the 28th Annual Symposium on Microarchitecture*, pages 237–242. IEEE, 1995.

[7] T.-F. Chen and J.-L. Baer. Effective Hardware-Based Data Prefetching for High-Performance Processors. *Transactions on Computers*, 44(5):609–623, May 1995.

[8] C.-H. Chi. Compiler Optimization Technique for Data Cache Prefetching Using a Small CAM Array. In *Proceedings of the 1994 International Conference on Parallel Processing*, pages I–263–I–266, August 1994.

[9] T. Chilimbi, M. Hill, and J. Larus. Cache-conscious structure layout. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, GA, May 1999.

[10] T. Chiueh. Sunder: A Programmable Hardware Prefetch Architecture for Numerical Loops. In *Proceedings of Supercomputing '94*, pages 488–497. ACM, November 1994.

[11] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.

[12] 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*, 22(3):462–479, September 1994.

[13] C. Ding and K. Kennedy. Improving cache performance of dynamic applications with computation and data layout transformations. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, GA, May 1999.

[14] J. Fu and J. Patel. Data Prefetching in Multiprocessor Vector Cache Memories. In *Proceedings of the 18th Annual Symposium on Computer Architecture*, pages 54–63, Toronto, Canada, May 1991. ACM.

[15] J. Fu, J. Patel, and B. Janssens. Stride Directed Prefetching in Scalar Processors. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 102–110, December 1992.

[16] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: An analytical representation of cache misses. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, Vienna, Austria, July 1997.

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

[18] H. Han and C.-W. Tseng. Improving locality for adaptive irregular codes. In *Proceedings of the Thirteenth Workshop on Languages and Compilers for Parallel Computing*, White Plains, NY, August 2000.

[19] D. Joseph and D. Grunwald. Prefetching using Markov Predictors. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 252–263, Denver, CO, June 1997. ACM.

[20] N. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 364–373, Seattle, WA, May 1990. ACM.

[21] M. Karlsson, F. Dahlgren, and P. Stenstrom. A Prefetching Technique for Irregular Accesses to Linked Data Structures. In *Proceedings of the 6th International Conference on High Performance Computer Architecture*, Toulouse, France, January 2000.

[22] A. Klaiber and H. Levy. An Architecture for Software-Controlled Data Prefetching. In *Proceedings of the 18th International Symposium on Computer Architecture*,

pages 43–53, Toronto, Canada, May 1991. ACM.

[23] M. Lam, E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, Santa Clara, CA, April 1991.

[24] H. Lu, A. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel. Compiler and software distributed shared memory support for irregular applications. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Las Vegas, NV, June 1997.

[25] C.-K. Luk and T. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, Boston, MA, October 1996.

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

[27] S. Mehrotra and L. Harrison. Examination of a Memory Access Classification Scheme for Pointer-Intensive and Numeric Programs. In *Proceedings of the 10th ACM International Conference on Supercomputing*, Philadelphia, PA, May 1996. ACM.

[28] J. Mellor-Crummey, D. Whalley, and K. Kennedy. Improving memory hierarchy performance for irregular applications. In *Proceedings of the 1999 ACM International Conference on Supercomputing*, Rhodes, Greece, June 1999.

[29] N. Mitchell, L. Carter, and J. Ferrante. Localizing non-affine array references. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Newport Beach , LA, October 1999.

[30] T. Mowry. Tolerating Latency Through Software-Controlled Data Prefetching, PhD Thesis. Technical report, Stanford University, March 1994.

[31] T. Mowry. Tolerating latency in multiprocessors through compiler-inserted prefetching. *IEEE Transactions on Computer Systems*, 16(1):55–92, February 1998.

[32] T. Mowry and A. Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, June 1991.

[33] T. Mowry, M. Lam, and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 62–73. ACM, October 1992.

[34] S. Palacharla and R. Kessler. Evaluating Stream Buffers as a Secondary Cache Replacement. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages pp. 24–33, Chicago, IL, May 1994. ACM.

[35] R. Panda, H. Nakamura, N. Dutt, and A. Nicolau. Augmenting loop tiling with data alignment for improved cache performance. *IEEE Transactions on Computers*, 48(2):142–149, February 1999.

[36] G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.

[37] G. Rivera and C.-W. Tseng. A comparison of compiler tiling algorithms. In *Proceedings of the 8th International Conference on Compiler Construction (CC'99)*, Amsterdam, The Netherlands, March 1999.

[38] G. Rivera and C.-W. Tseng. Tiling optimizations for 3d scientific computations. In *Proceedings of SC'00*, Dallas, TX, November 2000.

[39] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren. Supporting dynamic data structures on distributed memory machines. *ACM Transactions on Programming Languages and Systems*, 17(2):233–263, March 1995.

[40] A. Roth, A. Moshovos, and G. Sohi. Dependence Based Prefetching for Linked Data Structures. In *Proceedings of the Eigth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[41] A. Roth and G. Sohi. Effective Jump-Pointer Prefetching for Linked Data Structures. In *Proceedings of the 26th International Symposium on Computer Architecture*, Atlanta, GA, May 1999.

[42] R. Saavedra, W. Mao, D. Park, J. Chame, and S. Moon. The Combined Effectiveness of Unimodular Transformations, Tiling, and Software Prefetching. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 39–45, April 1996.

[43] Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, GA, May 1999.

[44] O. Temam. Streaming Prefetch. In *Proceedings of Europar'96*, Lyon,France, 1996.

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

[46] W. Wulf and S. McKee. Hitting the Memory Wall: Implications of the Obvious. *Computer Architecture News*, 23(1):20–24, March 1995.