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

Aneesh Aggarwal
Abdel-Hameed A. Badawy
Donald Yeung
Electrical and Computer Engineering Dept.
University of Maryland, College Park

Chau-Wen Tseng
Dept. of Computer Science
University of Maryland, College Park

Abstract

Software prefetching and locality optimizations are techniques for overcoming the gap between processor and memory speeds. Using the SimpleScalar simulator, we evaluate the impact of memory bandwidth and latency on the effectiveness of software prefetching and locality optimizations on three types of applications: regular scientific codes, irregular scientific codes, and pointer-based codes. We find software prefetching hides memory costs but increases instruction count and requires greater memory bandwidth. Locality optimizations change the computation order and data layout at compile or run time to eliminate cache misses, reducing memory costs without requiring more memory bandwidth. Combining prefetching and locality optimizations can improve performance, but interactions can also nullify the benefits of prefetching. We propose several algorithms to better integrate software prefetching and locality optimizations.

1 Introduction

Even with large on-chip caches, current microprocessors spend a large percentage of execution time on memory access stalls. Since processor speeds are growing at a greater rate than either memory and network speeds, we expect memory access costs to become even more important. In the not too distant future, it will not be far from the truth to say instructions are free and performance is determined only by memory access costs. Computer architects have been battling this *memory wall* by designing ever larger and sophisticated caches. However, as applications start using pointer-based linked data structures and performing irregular memory accesses, caches no longer perform well without additional help.

Two approaches to improving cache performance are *software prefetching* and *locality optimizations*. Software prefetching executes explicit prefetch instructions to begin loading data from memory to cache. If the prefetch begins early enough, and the data is not evicted prior to use, memory access latency can be completely hidden. Memory bandwidth use is increased, however, since the processor will now consume data at a faster rate. In comparison, locality optimizations use compiler or run-time transformations to the computation order and/or data layout of a program to increase the probability it accesses data already in cache. If successful, average memory latency and bandwidth are both reduced, since there will be fewer memory accesses.

Both approaches for avoiding the memory wall have been studied in isolation. In this paper, we examine how well each approach works for three types of data-intensive applications. We also evaluate both approaches in a unified environment, so we can compare their performance and investigate their interactions when applied in concert. Finally, we also study the impact of memory bandwidth and latency on the performance of each technique. The contributions of this paper are as follows:

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

Technical Report CS-TR-4169 (also UMIACS-TR-2000-57), Dept. of Computer Science, University of Maryland, July 2000

- We evaluate the impact of memory bandwidth and memory latency on application performance with and without prefetching and locality optimizations.

- We propose several enhancements to integrated software prefetching and locality optimizations.

We begin the rest of this paper with a look at three memory access patterns, then examine software prefetching and locality optimizations for each type of application. We present experimental evaluations for each application and develop improved algorithms. Finally, we discuss related work and conclude.

2 Memory Access Patterns

The types of software prefetching and locality optimizations which may be applied are seriously 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, the 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, assume array dimension sizes are known. This allows both software prefetching and compiler 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 ss computational scientists attempt more complex simulations. In computational fluid dynamics (CFD), meshes for modeling large problems are sparse to reduce memory and computations requirements. In n-body solvers such as those arising in astrophysics and molecular dynamics, data structures are by nature irregular because they model the positions of particles and their interactions. Unfortunately, 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* which contains 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 memory access patterns which are irregular and cannot be determined at compile time. Additionally, the next node cannot be traversed until the pointer value stored in the current node is found. These programs are thus known as *pointer-chasing* codes. Pointer-chasing codes occur in codes in many application domains, including scientific programs which use advanced data structures.

```
// Affine Array Accesses
// (3D Jacobi Kernel)

A(N,N,N),B(N,N,N)
do k=2,N-1
do j=2,N-1
do i=2,N-1
  A(i,j,k) = 0.16667 *
    (B(i-1,j,k)+B(i+1,j,k)+
     B(i,j-1,k)+B(i,j+1,k)+
     B(i,j,k-1)+B(i,j,k+1))
```

```
// Indexed Array Accesses
// (Molecular Dynamics)

X(M),Y(M),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
```

```
// Pointer-Based Structures
// (Linked List Traversal)

struct node {val, next};
node *ptr, *list;
while (...) {
  ptr->next = malloc(node);
  ptr = ptr->next;
  ptr->val = ... ;
}
while (ptr->next) {
  ...
  ptr = ptr->next;
}
```

Figure 1 Example Affine Array, Indexed Array, and Pointer-Chasing Codes

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 [5, 22, 32, 31]. Recently, techniques have also been developed to apply prefetching to pointer-based data structures [21, 41, 40, 28, 26]. 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, we follow the well-known compiler algorithm for inserting prefetches proposed by Mowry and Gupta in [32]. 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 [33], 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

Compared to affine array and indexed array prefetching, prefetching for pointer-based data structures is significantly more 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.

Recently, several novel prefetching techniques have been proposed that address the pointer-chasing problem [21, 41, 40, 28, 26]. One promising approach is *jump pointer prefetching* [41, 26]. 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 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. In addition to inserting the prefetch instructions that use the jump pointers, jump pointer prefetching also requires insertion of code to create and maintain the jump pointers as the data structure is modified.

4 Locality Optimizations

Software prefetching attempts to hide memory latency while retaining the original program structure. An alternative approach is to reduce memory costs by changing the computation order and data layout of a program at compile and run time. These *locality optimizations* attempt to improve *data locality*, the ability of an application to reuse data in the cache [44]. 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 access memory in different ways.

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 [44]. 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.

```
// 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

Tiling is very effective with linear algebra codes [12, 23, 24, 35, 37], and has been been extended to handle stencil codes used in iterative PDE solvers as well [42, 38, 44]. 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 [24]. This effect is shown in Figure 2.

Previous research found *tile size selection* and *array padding* can be applied to avoid conflict misses in tiles [12, 35, 37]. 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 nonconflicting tile dimensions through a simple recurrence [12, 37]. An alternative algorithm finds nonconflicting 2D tile using an greedy algorithm which expands tile dimensions while checking no conflicts occur [24]. We can adapt this algorithm for finding nonconflicting 3D tiles by iteratively attempting to increase each tile dimension until none may be increased without introducing conflicts [38].

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 of two. One possible solution is to use padding to enable better tile sizes [36]. 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 [37]. To combine padding with tile

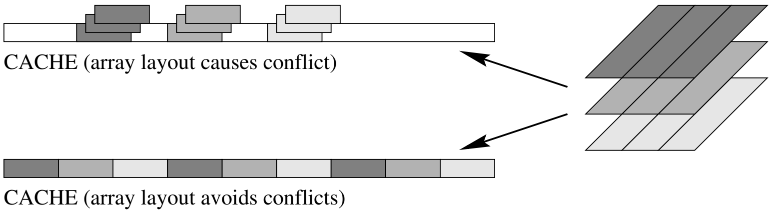


Figure 2 Example of Conflict Misses Under Two Array Layouts

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 predicted miss rate is within a small percentage of predicted optimal [38].

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. Recently, researchers have discovered run-time data and computation transformations can improve the locality of irregular computations [1, 14, 29, 30]. 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 approached developed for message-passing machines [13].

In many irregular scientific applications, each loop iteration tends to compute interactions for a pair of data. Such interactions tend to occur between “nearby” data items. Partitioning data based on either geometric coordinate data or the underlying interaction graph can thus increase the probability accesses will be made to data within the partition, increasing cache hits [1, 29, 18]. In particular, partitions based on geometric coordinates yield good locality if applicable.

To improve data locality of indexed array codes, we thus apply recursive coordinate bisection (RCB), a partitioning technique based on geometric coordinate information. RCB works by recursively splitting the dimension into two partitions, alternating axises. The process is recursively repeated as desired [2]. It is similar to using space-filling curves [29], but has the advantage of being able to handle unevenly distributed data. Once all partitions are selected, data items are stored consecutively within each partition, and partitions within an upper level partition are also arranged consecutively, constructing a hierarchical structure. We also lexicographically sort loop iterations afterwards based on the address of data accessed [14, 18, 30].

4.3 Memory Allocation For Pointers

Pointer-based programs frequently suffer from poor locality. There are many reasons why they underutilize cache. First, like irregular computations, data access patterns are unlikely to be regular sequential accesses. Second, because data is allocated dynamically, logically adjacent data may be in widely scattered memory locations because they were allocated at different times. Finally, the pointer-chasing problem means the pointer data at a memory location must be fetched before processor can determine what data is needed next. On top of all this, pointer-based codes are also notoriously difficult to analyze and transform, because of their reliance on pointers and heap-allocated recursive data structures.

Recently researchers have developed *cache-conscious* heap allocation and transformation techniques to improve locality for pointer-based programs [4, 10]. Algorithms include run-time tree optimization routines which place parent nodes with child nodes for improved locality, and apply 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 the user specified address. A heuristic which proved effective allocates new blocks of data, reserving space for future allocation requests [10]. 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, cache line utilization increases and fragmentation is reduced, decreasing the probability useful cache lines will be evicted from cache. We applied this optimization to our pointer-chasing benchmark code.

5 Experimental Evaluation

While both software prefetching and locality optimizations have been previously studied, we are aware of no studies of the effects of combining both on program performance, especially for a variety of memory bandwidths and latencies. To evaluate the combined performance of software prefetching and locality optimizations, we applied them by hand to six benchmark codes representing three classes of data-intensive applications. We then used simulations to evaluate their performance, both independently and in concert.

5.1 Simulation Environment

To study the effects of limited memory bandwidth on prefetching and locality optimizations, we used a detailed architectural simulator to measure the performance of several memory-bound applications. Our simulator is based on the SimpleScalar tool set [3] and models a state-of-the-art 4-way issue dynamically-scheduled processor that clocks at 1 GHz. 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 to the detailed processor model, our simulator also models the memory system. 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. While these cache sizes are small, they match the small input data sizes for our benchmarks. Memory requests across the L1-L2 bus do not experience contention; however, our simulator does model bus contention across the L2-memory bus.

To study the sensitivity of our prefetching and data locality transformation techniques to available memory bandwidth, we varied the L2-Memory bus bandwidth between 1-64 Gbytes/second. Note that a bandwidth of 1Gb/sec is equivalent to the processor loading one byte per cycle. We also varied memory to L2 latencies from 80 to 640 cycles. These parameters are intended to capture characteristics of future high-performance architectures, when processors are much faster than large DRAM memories.

5.2 Results for Affine Accesses

We begin by evaluating prefetching and tiling for three codes with affine array accesses: MATMULT, multiplying two 200x200 matrices; JACOBI, a 3D Jacobi iteration stencil over 200x200x8 arrays; and REDBLACK, a a 3D red-black successive-over-relaxation (SOR) kernel over 200x200x8 arrays. Both JACOBI and REDBLACK are frequently found in multigrid PDE solvers, such MGRID from the SPEC/NAS benchmarksuite.

Results are shown in Figure 4 for 80 cycle memory latency and different bandwidths. Execution time is plotted along the y-axis, with execution time broken down into memory stall and actual computation. Program version and available memory bandwidth (from 1 byte/cycle to 64 bytes/cycle) is plotted along the x-axis. Groups of stacked bars represent the original version of each program, and versions optimized with either software prefetching, tiling, or both.

Results show performance generally improves as available memory bandwidth increases. When available bandwidth is low, tiling achieves the best performance. When bandwidth is high, prefetching obtains the best performance. Prefetching versions typically have higher BUSY times because of the added overhead for issuing prefetch instructions. Tiling also increases BUSY time since extra overhead is incurred for the extra levels of loops.

For affine codes, combining tiling and prefetching did not help performance. The main problem is that tiled codes had smaller iterations in the innermost loop, increasing the amount of pipelining startup overhead and reducing amount of latency hidden. The CPU overhead is higher than either prefetching or tiling alone. As a result performance is degraded for all but REDBLACK under some conditions.

We also performed simulations for 160, 320, and 640 cycle memory latencies for each benchmark. Results are shown in Figure 5. Execution time is plotted along the y-axis, and memory bandwidth is plotted along the x-axis. Results for each memory latency is displayed as a separate line in each graph. Each version of the program (original, prefetch, optimized, both) is shown in a separate graph. We see that in general execution times are higher with larger memory latencies. However, prefetching is able to take advantage of high memory bandwidths to hide memory latency, while tiling cannot. In fact, the implication for memory system designers seems to be very wide memory bandwidths are useful only for prefetching for high-latency memory systems.

5.3 Results for Indexed Accesses

Next, we evaluate prefetching and RCB applied to two two programs, IRREG and MOLDYN. 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. We ran IRREG with an input data consisting of a 14K node mesh of an airfoil. We ran MOLDYN with an input data set representing interactions between 13K semi-uniformly distributed molecules.

The optimized versions of each program use a run-time *inspector* to apply RCB to rearrange data and computation to improve locality. We measured simulated performance with software prefetching and locality optimizations. Because no preprocessing costs are included, results are representative of long-running applications where preprocessing overhead can be amortized over many loop iterations. Results are shown in Figure 4.

As before, prefetch profits most when bandwidth is high. Instruction overhead is significantly higher with prefetching because the irregular accesses must be prefetched individually rather than as a group (multiple data on a single cache line). Nonetheless, because prefetches do not have to be executed too far in advance with only one level of indirection, software prefetching can hide almost all memory latency for the irregular codes examined. For these irregular computations, combining prefetching and RCB eliminated the most memory cycles, but had high CPU overhead. RCB was able to yield major performance improvements even when little memory bandwidth is available. For these programs with indexed array accesses, program transformations did not affect the efficacy of software prefetching.

We also looked at the effect of varying memory latencies. Results are shown in Figure 6. As before, we see that prefetching is able to hide very high memory latencies with sufficient memory bandwidth, while locality optimizations still suffer somewhat from memory latency. Combining prefetching with locality optimizations yields the best performance.

5.4 Results for Pointer Accesses

Finally, we compare prefetching and cache-conscious memory allocation for HEALTH, a pointer-based program from the OLDEN benchmark suite which simulates the Columbian health care system [39]. The optimized version of the program uses a cache-conscious memory allocator which uses the new-block allocation policy to reserve space for linked list elements, allowing most consecutive elements of a linked list to be placed next to each other.

Simulation results for HEALTH run with five-deep trees are shown in Figure 4. Notice memory overheads are much higher for pointer-based programs, due to the many irregular and indirect accesses. As before, when available bandwidth is low, locality optimizations performs the best. When bandwidth is high, prefetching achieves the best performance. Combining prefetching and smart memory allocation did not improve performance, because the high overhead of jump pointers were not helpful when most list elements are laid out contiguously. Spatial locality from long cache lines were sufficient to reduce cache misses.

We also looked at the effect of varying memory latencies. Results are shown in Figure 6. Unlike previous programs, prefetching was unable to hide high memory latencies. The prefetch distances computed for jump pointers were too high for the short linked lists used in the program. We address this problem later.

5.5 Discussion

Despite comparing software prefetching to three different locality optimizations on three classes of optimizations, our results are surprisingly similar. When memory bandwidth is high, software prefetching by itself outperforms other versions, except for pointer-chasing programs. Once memory bandwidth is reduced, locality optimizations become more effective. Only irregular computations benefited from directly combining locality optimizations and software prefetching, either because the optimizations either interacted poorly, or because either optimization or prefetching alone were able to achieve most of the benefits possible.

With current processor speeds, maintaining memory bandwidths of 1-4 bytes per cycle are 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 towards 1 byte/sec or lower. Since memory bandwidth will probably continue to drop relative to processor speed, locality optimizations should become more important. Similarly as processor speeds increase, memory latencies are likely to increase past 80 cycles, making our high end simulation results more relevant.

One possibility for dramatically increasing memory bandwidth is to switch to processor-in-memory (PIM) architectures. For on-chip data, available memory bandwidth will be more like that towards the high end of the scale, around 16-64 bytes/cycle. Our experiments show such PIM systems should benefit significantly from software prefetching, since only they can support the high level of memory bandwidth required. However, even PIM systems will require locality optimizations to reduce accesses to off-chip data.

6 Algorithm Enhancements

In addition to determining the effects of memory bandwidth and latency on performance, our simulations also pointed out a number of ways to enhance both software prefetching and locality optimizations.

6.1 Tiling and Prefetching

We saw that a problem with combining tiling and software prefetching naively is overhead from small innermost loops. We can improve performance by modifying the tiling algorithm to select tiles loops with more iterations in the innermost loop, even though the cache model may estimate more cache misses. Our tiling heuristic uses either the Euclidean GCD algorithm [12, 37] or a version of the Lam, Rothberg, Wolf algorithm modified for 3D arrays [24, 38] to generate a series of non-conflicting tile shapes. As a result we can simply bias the selection towards tall tiles with larger innermost loops, at the expense of lower cache utilization.

Figure 7 presents our results for our JACOBI and REDBLACK benchmarks. Instead of picking a tile size of 46x39x1, our algorithm can pick a tile of 200x8x1 instead. Software prefetch overhead is reduced four-fold with the taller tile, and there is more time to hide latency within the innermost loop. Simulation results show an improvement, demonstrating taller tiles allow us to combine the benefits of software prefetching and tiling.

6.2 Jump Pointers

One limitation of jump pointer prefetching is that the first D link nodes in a list are not prefetched, where D is the prefetch distance, because there are no jump pointers pointing to these early nodes. As memory latency increases, D must increase to accommodate a larger cache miss stall time. Consequently, jump pointer prefetching becomes less effective at large memory latencies due to unprefetched early nodes.

In our experiments, we found that it is profitable to reduce the prefetch distance in order to minimize the number of unprefetched early nodes. Figure 8 shows the impact of reducing the prefetch distances from the computed values of 31, 62, 124, and 247 at memory latencies of 80, 160, 320, and 640, respectively, to a prefetch distance of 16 across all memory latencies. As a result of this optimization, performance improves for all memory latencies. The performance improvement is most pronounced at large memory latencies where the number of unprefetched nodes is largest.

We note that our prefetch distance reduction optimization for pointer prefetching is similar to prefetch arrays, recently proposed by Karlsson et al in [21]. In prefetch arrays, an array of pointers is created for each linked list. Each element of the pointer array points to one of the D early nodes in the list, hence allowing prefetching of the early nodes. Our optimization achieves a similar benefit since it prefetches the early nodes in a list, but it requires no additional code modifications on top of the basic jump pointer prefetching technique.

6.3 Padding for Software Prefetching

While software prefetching can usually hide memory latency given sufficient memory bandwidth, conflict misses on prefetched data can degrade or even completely eliminate benefits. The problem is that for some applications and problematic data sizes, severe conflict misses may result, with all prefetched data landing in a few cache lines. This problem is especially acute for affine array accesses to arrays which are close to a multiple of the cache size. Compilers can avoid this problem and pad arrays to avoid prefetch conflicts [36, 37], even if tiling is not needed. For instance, in our experiments we found that software prefetching was ineffective for JACOBI and REDBLACK with a problem size of 256x256x8, unless the innermost dimension was padded to 259, as predicted by compiler analysis [36].

6.4 Memory Allocation and Software Prefetching

We can also combine the benefits of memory allocation with software prefetching. Jump pointers are not very useful when intelligent memory allocation can put elements of a linked list contiguously in memory. In these cases software prefetching should directly calculate a distance based on the current address and memory latency, just as for affine array accesses. In our experiments we found we could improve performance for HEALTH by up to 5% by using index prefetching with distance 16 instead of jump pointers when memory latency was 80 cycles. Larger improvements should result for higher memory latencies.

7 Related Work

Conventional software prefetching [31, 22, 5] has investigated prefetching for arrays. Work in hardware prefetching [8, 34, 16, 15, 20] is similarly limited to arrays, but uses hardware to identify what to prefetch automatically. Prefetch engines [43, 7, 11, 9] provide hardware support for prefetching, but rely on the programmer or compiler to identify the access pattern. Like hardware and software prefetching, prefetch engines have focused on arrays as well.

Recently, researchers have begun investigating novel prefetching techniques for pointer-chasing traversal [21, 41, 40, 28, 26]. These new techniques address the pointer-chasing problem using one of three different approaches. The first approach inserts additional pointers into dynamic data structures to connect non-consecutive link elements [21, 41, 26]. These "jump pointers" allow prefetch instructions to name link elements further down the pointer chain without sequentially traversing the intermediate links.

The second approach performs prefetching using only the natural pointers belonging to the dynamic data structure, and thus do not require additional state for prefetching [40, 28, 26]. Existing stateless techniques prefetch pointer chains sequentially, and do not exploit any memory concurrency. Instead, they schedule each prefetch as early in the loop iteration as possible to maximize the amount of work available for memory latency overlap. Finally, a third approach uses a hardware table known as a "Markov predictor" to store temporally related cache-missing memory addresses observed at run time [19]. The table is used to direct prefetching by predicting the addresses of future cache misses.

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 [44], though loop fission (distribution) and loop fusion have also been found to be helpful [27].

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 [17, 44]. Tiling has been proven useful for linear algebra codes [24, 44, 12] and multiple loop nests across time-step loops [42]. 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 [13] and compiler [25] support to support accesses on message-passing multiprocessors. A few have also looked at techniques for improving locality [1, 14].

Few researchers have investigated data layout transformations for pointer-based data structures. Chilimbi, Hill, and Larus investigate allocation-time and run-time techniques to improve locality for linked lists and trees [10]. We propose extension to their work. Calder *et al.* use profiling to guide layout of global and stack variables to avoid conflicts [4]. Carlisle *et al.* investigate parallel performance for pointer-based codes in Olden [6].

8 Conclusions and Future Work

We believe software and architecture support is need 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 preliminary results have been encouraging, much work remains to be done.

Our results have mostly been achieved for individual kernels. These kernels are important because they take up the vast majority of processing time in larger programs, but we must still evaluate the effectiveness of our techniques for larger, more realistic programs. Currently our locality optimizations are 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. We need to expand the range of applications, so that in addition to improving current codes, we can also gain insights for providing better solutions for future applications.

Our goal of reducing the memory bottleneck is essential if we wish to continue increasing the computation power available to scientists and engineers. Because of trends in computer architectures, our insights are likely to prove very useful for improving the memory performance of commercial applications such as image processing and high-performance databases.

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] M. Berger and S. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Transactions on Computers*, 37(12):570–580, 1987.

[3] Doug Burger and Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. CS TR 1342, University of Wisconsin-Madison, June 1997.

[4] 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.

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

[6] 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.

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

[8] Tien-Fu Chen and Jean-Loup Baer. Effective Hardware-Based Data Prefetching for High-Performance Processors. *Transactions on Computers*, 44(5):609–623, May 1995.

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

[10] 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.

[11] Tzi cker Chiueh. Sunder: A Programmable Hardware Prefetch Architecture for Numerical Loops. In *In Proceedings of Supercomputing '94*, pages 488–497. ACM, November 1994.

[12] 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.

[13] 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.

[14] 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.

[15] John W. C. Fu and Janak H. 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.

[16] John W. C. Fu, Janak H. Patel, and Bob L. Janssens. Stride Directed Prefetching in Scalar Processors. In *In Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 102–110, December 1992.

[17] 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.

[18] 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.

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

[20] Norman P. 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] Magnus Karlsson, Fredrik Dahlgren, and Per 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] Alexander C. Klaiber and Henry M. Levy. An Architecture for Software-Controlled Data Prefetching. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 43–53, Toronto, Canada, May 1991. ACM.

[23] I. Kodukula, K. Pingali, R. Cox, and D. Maydan. An experimental evaluation of tiling and shacking for memory hierarchy management. In *Proceedings of the 1999 ACM International Conference on Supercomputing*, Rhodes, Greece, June 1999.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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

[29] 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.

[30] 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.

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

[34] Subbarao Palacharla and R. E. 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.

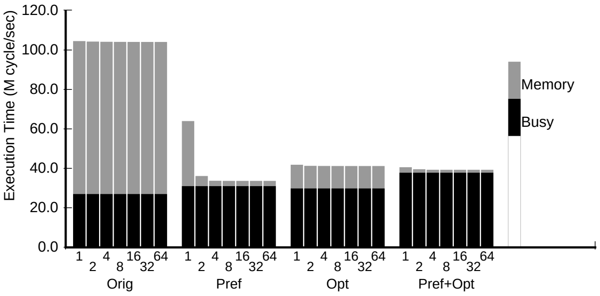
[40] Amir Roth, Andreas Moshovos, and Gurindar S. Sohi. Dependence Based Prefetching for Linked Data Structures. In *In Proceedings of the Eigth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

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

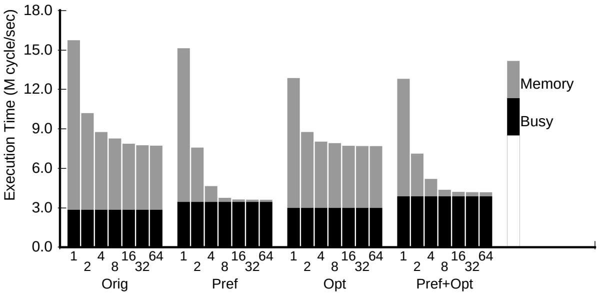
[42] Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, GA, May 1999.

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

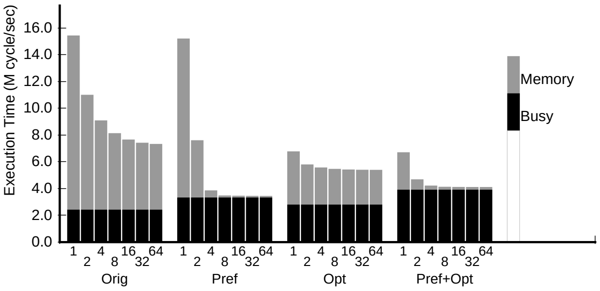
[44] M. E. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.



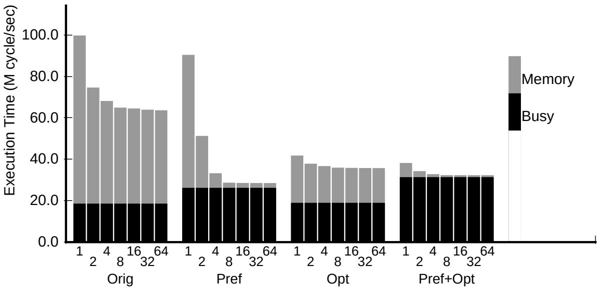
MATMULT



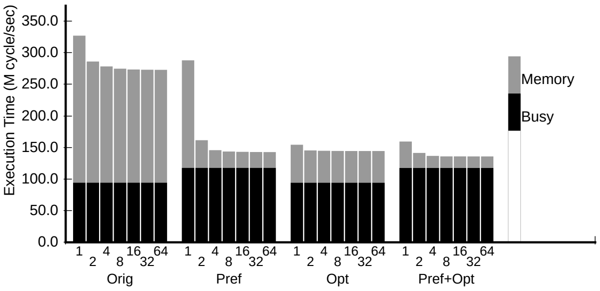
JACOBI



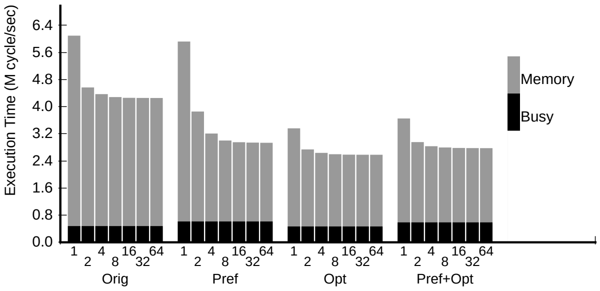
REDBLACK



IRREG

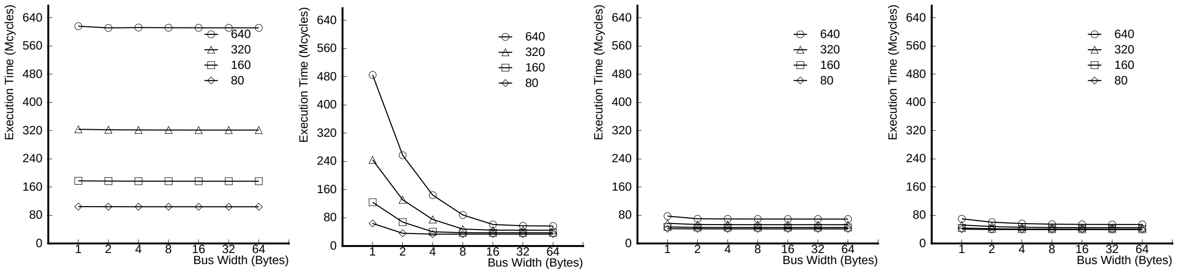


MOLDYN



HEALTH

Figure 4 Execution Times vs Memory Bandwidth (Original, Optimized, Prefetch, Opt+Pref)

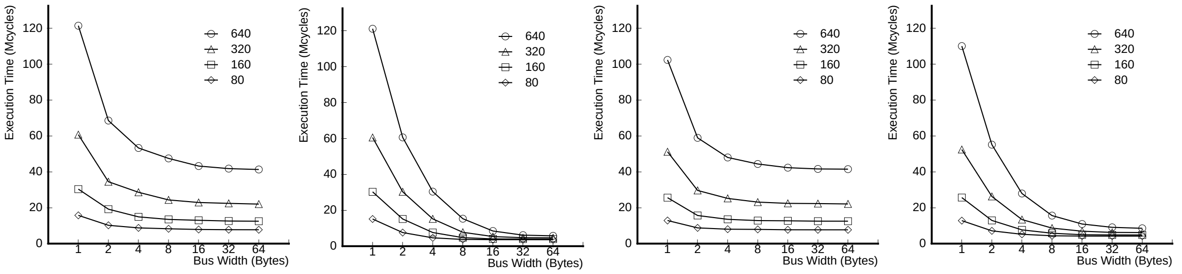


MATMULT Orig

MATMULT Pref

MATMULT Opt

MATMULT Opt+Pref

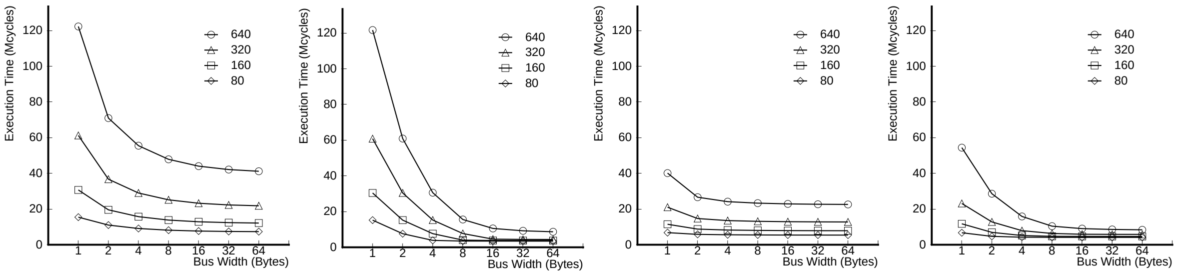


JACOBI Orig

JACOBI Pref

JACOBI Opt

JACOBI Opt+Pref



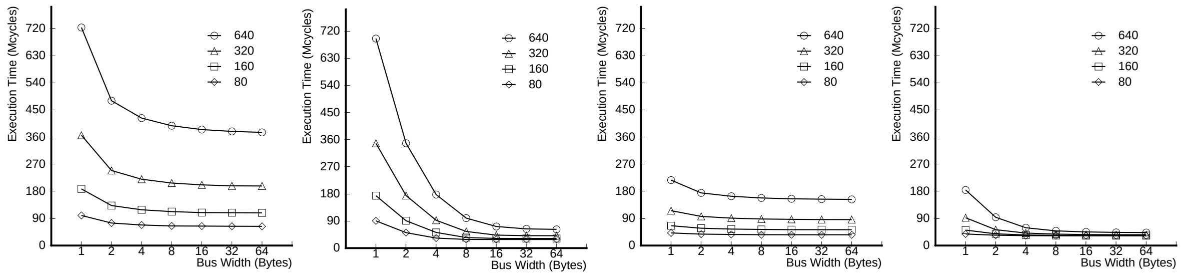
REDBLACK Orig

REDBLACK Pref

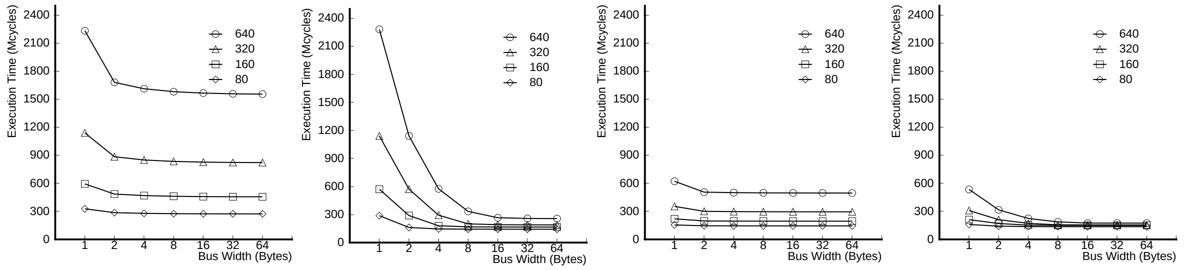
REDBLACK Opt

REDBLACK Opt+Pref

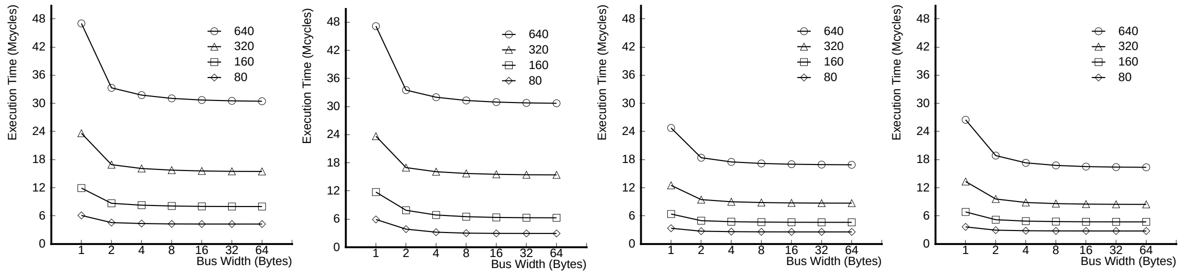
Figure 5 Execution Times vs Memory Latency (Affine Array Accesses)



IRREG Orig   IRREG Pref   IRREG Opt   IRREG Opt+Pref

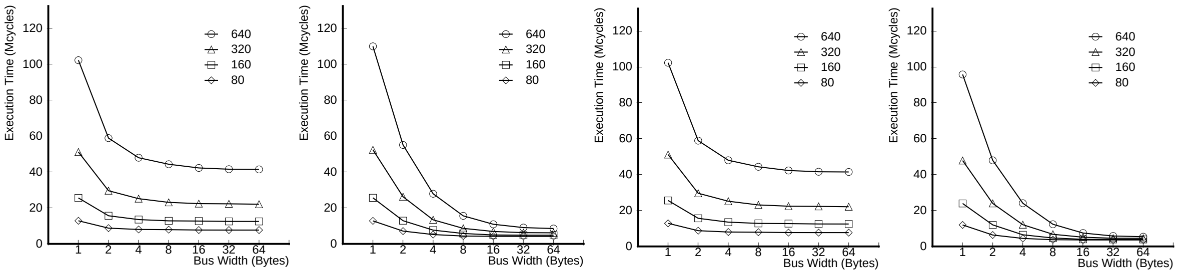


MOLDYN Orig   MOLDYN Pref   MOLDYN Opt   MOLDYN Opt+Pref



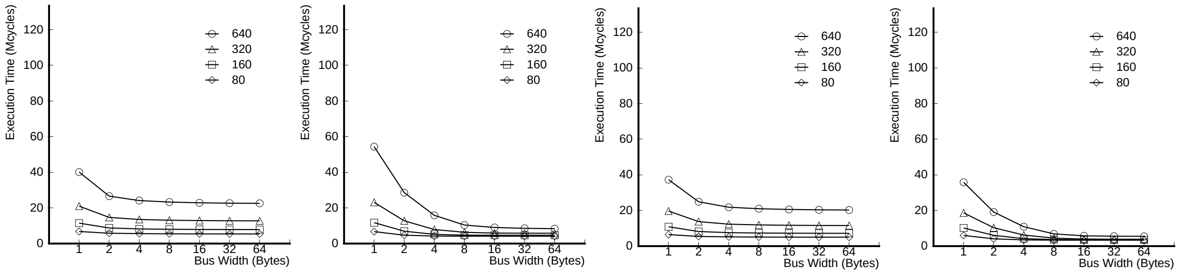
HEALTH Orig   HEALTH Pref   HEALTH Opt   HEALTH Opt+Pref

Figure 6 Execution Times vs Memory Latency (Indexed Array and Pointer Accesses)



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

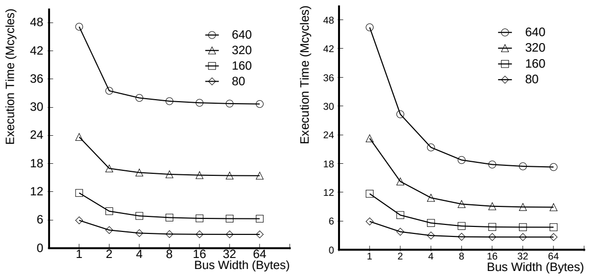
JACOBI



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

REDBLACK

Figure 7 Prefetching with Square and Tall Tiles



Pref (jump=calc) Pref (jump=16)

Figure 8 HEALTH with Computed and Fixed Jump Pointers