ABSTRACT

Title of Thesis: EVALUATING THE IMPACT OF MEMORY SYSTEM PERFORMANCE ON SOFTWARE PREFETCHING AND LOCALITY OPTIMIZATIONS

Degree candidate: Abdel-Hameed A. Badawy

Degree and year: Master of Science, 2002

Thesis directed by: Professor Donald Yeung
Department of Electrical and Computer Engineering

Software prefetching and locality optimizations are two techniques for overcoming the speed gap between processor and memory known as the memory wall as suggested by Wulf and Mckee [57]. This thesis evaluates 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. For many applications, software prefetching outperforms locality optimizations when there is sufficient bandwidth in the underlying memory system, but locality optimizations outperform software prefetching when the underlying memory system doesn't provide sufficient bandwidth. The break-even point, or equivalently the crossover bandwidth point, occurs at roughly 2.4 GBytes/sec , for 1 GHz processors on today's memory systems, and will increase on future memory systems. This thesis also studies the interactions between software prefetching and locality optimizations when applied in concert. Naively combining the two techniques provides a more robust application performance in the face of variations in memory bandwidth and/or latency, but does not yield additional performance gains. In other words, the performance won't be better than the best performance of the two techniques alone. Also, several algorithms are proposed and evaluated to better combine software prefetching and locality optimizations, including an enhanced tiling algorithm, padding for software prefetching, and index prefetching.

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

by

Abdel-Hameed Abdel-Salam Badawy

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Master of Science
2002

Advisory Committee:

Professor Donald Yeung
Professor Manoj Franklin
Professor Chau-Wen Tseng

ACKNOWLEDGMENTS

This is going to be a long list of people to thank since prophet Muhammad said "Whoever is not thankful to people is not thankful to ALLAH(GOD)" so please bear with me.

I would like to thank Aneesh Agrawal for his efforts which helped me to finish this work leading to this thesis. Aneesh implemented the memory system model on top of SimpleScalar which was called sim-bw. He gave me a jump-start on the benchmarks and the simulator while he was an intern at DEC in summer of 2000. Also, I would like to thank all my lab-mates including [Choi, Dongkeun, Deepak, Gautham, Aamer, and Zahran] for their support during the deadlines of HPCA'2001 and ICS'2001. They are just a wonderful group of people to work nearby them.

Also, I am grateful to Gabe Rivera (professor Chau-Wen's student) who helped me understand his work on tile selection techniques and for his willingness to discuss his work with me. Also, I would like to thank Hansoo Han for giving us his locality optimized code for the irregular array benchmarks.

I would like to thank professor Chau-Wen for his suggestions, guidance, and support. He really has a great inertia to put work together. I learnt from him never to give up submitting papers to conferences no matter what and never give up. I learnt from him to always think positively and work as fast as possible.

At last but not least,I would also like to thank my advisor, professor Donald Yeung. I appreciate very much his valuable contributions to my understanding of computer architecture at large and to the topics covered in this thesis, and specifically my experience as a graduate student under his supervision. He deserves much more than what I can say here.

I would also like to thank my family for their love support, encouragement and supplications. I would love thank my wife for her continual support in every way possible. Also, I would like to thank her family for their supplications and love.

Finally, I thank GOD, or equivalently ALLAH in Arabic, the all mighty who have given human beings countless blessings. To name one, our ability to do research and to ask and answer questions that lead humanity to prosper and flourish. All the praise and thank are due to the LORD of the whole universe. Allah the self-sufficient master, whom all creatures need and needs none of them. The one who begets not, nor was he begotten and there is none co-equal or comparable unto him.

Contents

List of Figures

List of Tables

ABBREVIATIONS

ISA	Instruction Set Architecture
L1	Level 1
L2	Level 2
KB	Kilobyte (1024 Bytes)
MB	Megabyte (1024 Kilobytes)
PD	Prefetch Distance
JPP	Jump Pointer Prefetching
PDE	Partial Differential Equations
AVG	Average
MM	Matrix Multiply
Jac	Jacobi
RB	RedBlack
Irreg	Irregular Mesh Solver
Mol	Moldyn
Health	Columbian Health Simulation
MST	Minimum Spanning Tree
gpart	Graph Partitioning
rcb	Recursive Coordinate Bisection
Orig	Original Code
Pref	Code instrumented with Prefetching
Opt	Code instrumented with the appropriate Locality Optimization Technique
Pref+Opt	Code instrumented with both Prefetching and Locality Optimization

Chapter 1

Introduction

1.1 The Memory Wall

The performance of microprocessors continues to improve at an impressive pace. In fact, microprocessor performance increases by 58% per year. However, memory system performance improves by only 7% in the same amount of time [23], leading to an exponential increase in the *processor-memory performance gap* [57]. In the early 1980's, memory systems were fast enough to keep up with processors. Unfortunately, since processor and memory performance increase at different exponential rates, their difference in performance increases exponentially. Figure 1.1 illustrates how extreme this problem is, plotting both memory and processor performance over time. The time axis is on a linear scale; however, the performance axis is on a logarithmic scale. Figure 1.1 shows that processor performance is a factor of 100 higher than memory performance in 2000, and it will reach a factor of 5000 in 2010.

The most effective known solution to alleviate this problem is to use caches. Caches are small, fast memories that store recently accessed data. The

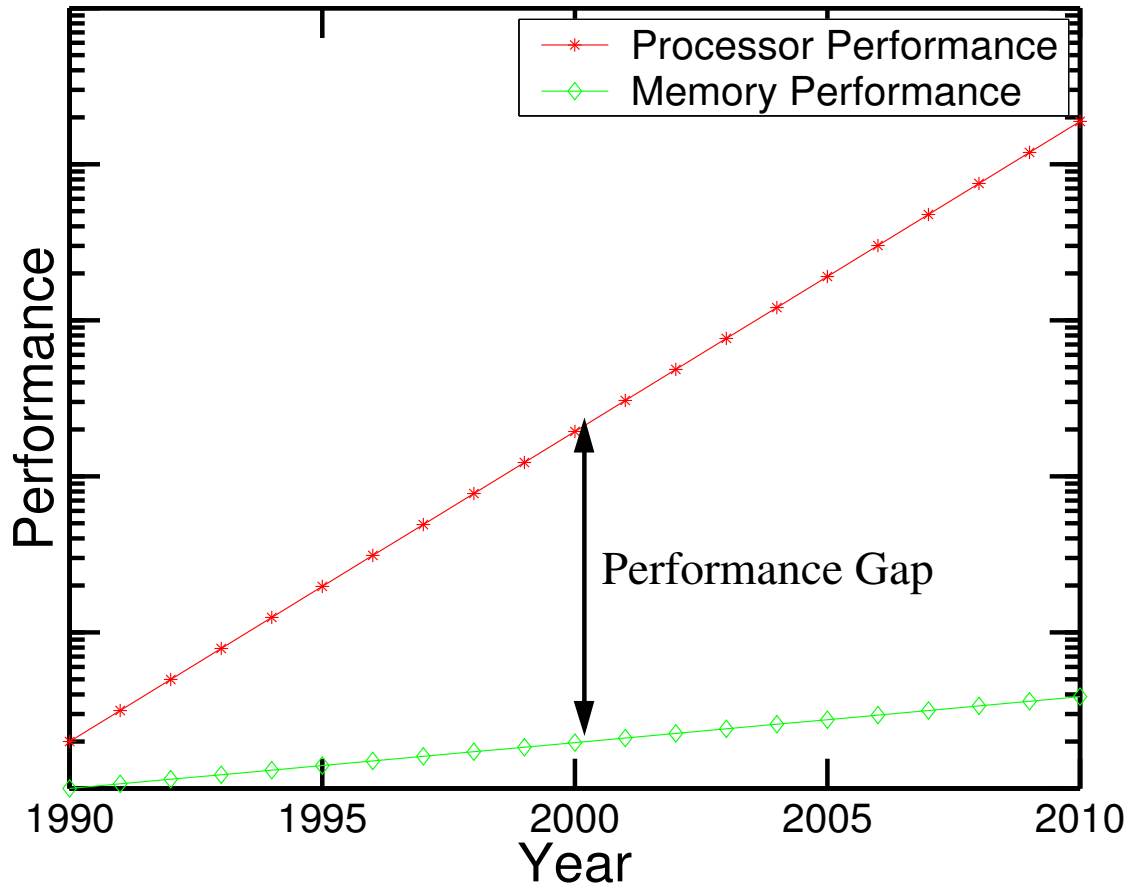


Figure 1.1: Comparison of processor performance to memory performance over time.

principle of temporal locality states that once data is accessed, it is likely to be accessed again in the near future. Therefore, data will likely be accessed multiple times and only need to be retrieved from main memory the first time it is accessed. This allows successive accesses to be satisfied from the faster cache, effectively reducing the average latency required to access data. However, since caches are limited in size, they do not in all cases have enough capacity to fit the application's working set especially for large applications. Also, some applications do not have sufficient temporal locality to allow caches to reduce the average latency of a data access. Consequently, while caches are typically effective, they do not completely address the memory gap problem.

When a processor accesses data from main memory, it must wait for the memory system to retrieve the data. This is called a memory stall. As the processor-memory performance gap continues to widen, memory stalls increase and application performance becomes increasingly limited by the memory system performance. Other techniques are required to fully address the memory wall problem. The rest of this Chapter introduces two existing techniques, software prefetching and data locality transformations.

1.2 Introducing Software Prefetching

Two promising approaches for improving memory performance are *software prefetching* and *locality optimizations*. This section briefly introduces software prefetching. Software prefetching executes explicit prefetch instructions to initiate loading data from memory to cache early. Prefetching works by

pre-loading data from memory before the processor requests it so that it is ready when the processor performs the access, thus hiding the latency of the memory access from the processor. Prefetching can be controlled in either hardware or software. In this thesis, software prefetching is considered. In software prefetching, the compiler identifies loops that are likely to cause frequent cache misses, and inserts prefetch instructions into the application loop code to prefetch the data in advance of its use. The compiler does this by pairing each LOAD instruction with a PREFETCH instruction that prefetches data to the cache. Scheduling is done for the PREFETCH instruction so that the data is available to the processor when it is requested.

When data is prefetched, it is loaded from memory and put into cache [7, 34] or a special buffer called a *prefetch buffer* [28, 47]. When data is prefetched into a prefetch buffer, it is moved into the L1 cache when it is referenced by the processor. This prevents inaccurately prefetched data from polluting the cache and thus evicting useful cache blocks. In this thesis, prefetching into the L1 cache is considered only. Figure 1.2a illustrates software prefetching using a simple code example. This code sequentially loads data from an array, shown on the right side of Figure 1.2a, and performs some computation on each array element. Figure 1.2b shows the execution time-line for the array traversal in Figure 1.2a. It shows that most of the execution time is spent in memory stall. With Prefetching shown in Figure 1.2c, each element of the array can be preloaded to avoid memory stalls and thus the latency is hidden underneath the execution of the loop computations. On each loop iteration, the

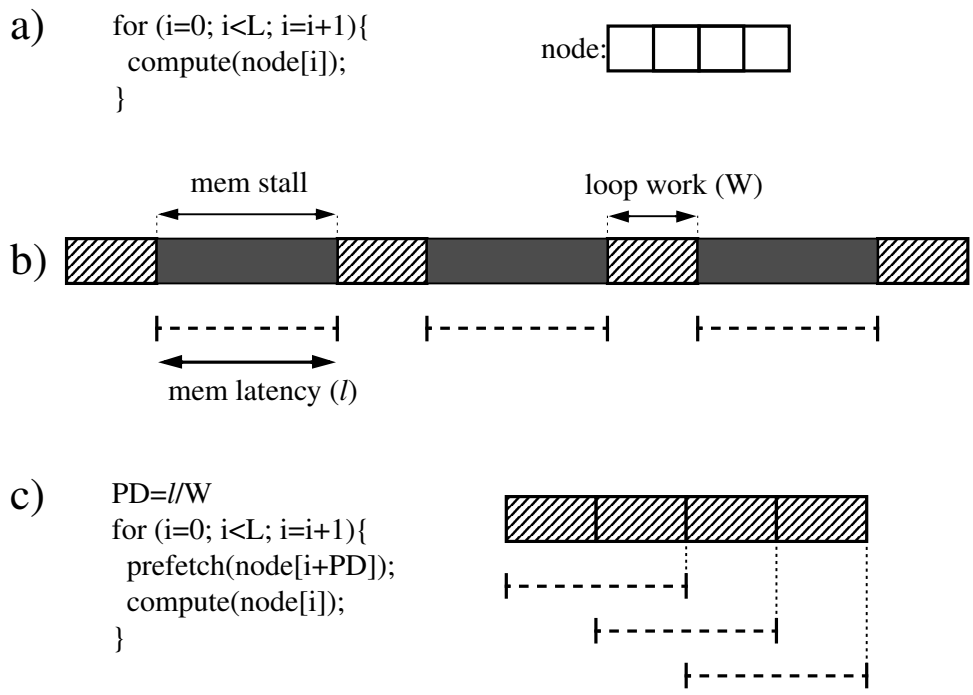


Figure 1.2: Array traversal. a) Code with data structure. b) Time-line without prefetching. c) Code and time-line with prefetching.

array element accessed PD loop iterations ahead is prefetched. The value PD is referred to as the *prefetch distance*, and determines how far ahead (in terms of loop iterations) to prefetch data from memory. In other words, PD determines the number of loop iterations necessary to hide the latency of one array element access. Figure 1.2c also shows the execution time-line for the array example with a prefetch distance of two.

The code shown in Figure 1.2c does not prefetch the first PD elements of the array because the first element prefetched is node[0+PD]. To address this problem, a prologue loop is added to the code just before the traversal code loop (not shown in Figure 1.2). The prologue loop prefetches the first PD elements of the array without performing any computation on the data itself. The time-line in Figure 1.2c is shown for "steady state" iterations only, so it doesn't show the

prologue loop prefetches. The value of the prefetch distance, PD, is computed according to Equation 1.1

$$PD = \lceil l/W \rceil \tag{1.1}$$

where l is the memory latency and W is the amount of work per loop iteration. Unfortunately, the values of l and W are not exactly known at compile time. The value of l depends on whether data is found in the L1 cache, L2 cache, or main memory. The value of W varies if there are conditional statements in the loop body. Since both l and W are not constant, typically worst case values are chosen. Also, note that PD must be rounded up to the nearest whole number, thus we use the ceiling in the equation for the prefetch distance. Both conservative estimates of the prefetch distance and roundoff error can result in PD being too high and data being prefetched too early. The problem with early prefetches is that the data being pre-loaded to the cache may be evicted before it is consumed by the processor.

Prefetching is effective only if sufficient memory bandwidth exists to transfer all prefetched data in time. If the memory system cannot transfer the data fast enough, memory stalls will remain unresolved and degrade overall performance. As processor speeds increase, memory bandwidth requirements increase too since the processor will consume data at a faster pace, requiring the memory system to supply data more rapidly in order to avoid stalling the processor.

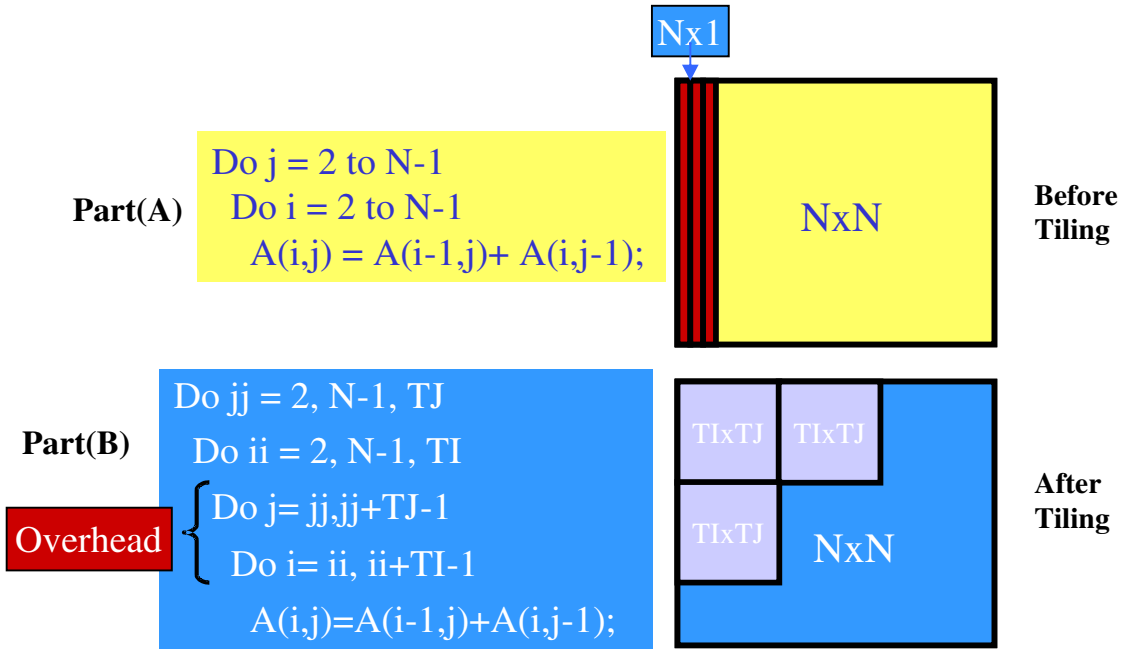


Figure 1.3: Example Locality Optimization (Tiling). Part(A) Original code without any optimizations. Part(B) Tiled code.

1.3 Introducing Locality Optimization

In comparison to prefetching outlined in the previous section, locality optimizations use compiler or run-time transformations to change the computation order and/or data layout of a program to increase the locality of the processor's memory access patterns, improving the probability that the processor accesses data that is already in the cache. Thus locality optimizations are not latency tolerance techniques like prefetching. Instead, they are latency reduction techniques since they use compiler or run-time transformations in order to make better use of the data that is already present in the cache.

Figure 1.3 illustrates a particular form of locality optimization called tiling [14, 29, 30, 42, 44]. This technique is particularly useful for statically allocated regular arrays. Figure 1.3 part(A) illustrates a two dimensional version

of the Jacobi code. The original computation proceeds down the columns of the array. Since the computation of each array element uses neighboring values along rows, there is reuse across outer loop iterations. Unfortunately, this reuse cannot be exploited unless multiple columns fit in the cache simultaneously, which does not occur for large arrays or small caches. The locality optimization algorithm is applied to the loop either by hand or the compiler.

Figure 1.3 part(B) illustrates JACOBI code with tiling. Two more loops are inserted to force the computation to go tile by tile, exploiting reuse along rows and columns more effectively. This causes the number of cache misses and the amount of traffic moved from memory to cache to go down, thus improving the overall system performance. In effect, locality optimization techniques increase the reuse of the data that is already in the cache so that when it gets evicted it will not be used again later since the application has exhausted this piece of data for all the computations that needs this piece of data. In general, when locality optimization techniques are applied to a specific application and it turns out to be successful, both average memory latency and bandwidth usage are reduced. On the negative side, the additional loops inserted by tiling introduce overhead, similar to the overhead that prefetch instructions introduce in software prefetching.

1.4 Thesis Organization

Both software prefetching and locality optimizations have been studied in isolation. This thesis conducts an in-depth evaluation that compares the two

techniques under different memory system design points. The evaluation uses benchmarks from three broad classes of data-intensive applications. In addition, the evaluation uses a single unified simulation environment based on the Simple-Scalar tool set [5] with a detailed memory system to enable a meaningful comparison. The primary focus of the work is to compare the importance of *latency tolerance* provided by software prefetching and *latency reduction* provided by locality optimizations on future high-performance memory systems. The work also investigates the interactions of software prefetching and locality optimizations when applied in concert both naively and then with some enhancements to increase the effectiveness of their combination.

The contributions of this thesis are as follows:

- Compare the efficacy of software prefetching and locality optimizations for three types of data-intensive applications in terms of performance.
- Quantify the impact of memory system parameters, such as bandwidth and latency, in future memory systems on the relative effectiveness of software prefetching and locality optimizations.
- Examine the performance of integrated software prefetching and locality optimizations, then propose and evaluate several enhancements to increase their combined performance.

The rest of this thesis will be organized as follows. First, related work is discussed in Chapter 2. Second, the three memory access patterns are explained

in Chapter 3. Then, different optimizations for each access pattern is discussed. Chapter 4 discusses software prefetching optimizations. Chapter 5 discusses locality optimizations. The experimental results are presented in Chapter 6. Improved algorithms are discussed in Chapter 7. Chapter 8 presents the conclusions. Finally, Chapter 9 discusses the future extensions to this thesis.

Chapter 2

Related Work

This thesis is similar to Saavedra *et al* [49], in which they evaluated unimodular transformations, tiling, and software prefetching for matrix multiply only using a cache simulator as their measurement environment. Mowry *et al* [39] evaluated software prefetching and tiling for two scientific applications. This thesis is focused on memory system parameters scaling and quantification of their impact on software prefetching and locality optimizations. The previous works have considered only a fixed technology point. Furthermore, 3 classes of benchmarks are studied in this thesis requiring different types of optimizations. New enhancements are proposed to better combine software prefetching and locality optimizations, as well as an enhancement to software prefetching when applied to array-based benchmarks. The experimental evaluation methodology used is a detailed execution-driven simulator for a modern processor and memory system.

Most of the work done before has been devoted to the study of the two techniques in isolation, and little work has been done to study the combined techniques. To our knowledge, the only one to do this is Saavedra *et al* [49]. The conclusions in that paper were negative. They concluded that the combination

suffers degradation in performance due to destructive interference but didn't suggest anything to address the problem.

Software prefetching for affine array accesses has been studied in [37, 28, 7] as will be described in detail in Section 4.1. Hardware prefetching [10, 41, 19, 18, 26] uses hardware to identify the access pattern automatically. Prefetch engines for affine array accesses [53, 9, 13, 11] provide hardware support for prefetching, but rely on the programmer or compiler to identify the access pattern, *i.e.* the hardware itself doesn't detect the access pattern automatically like hardware prefetching. Intel and AMD are both having some hardware prefetching techniques in their latest processors (Pentium 4 and Athlon 4 chips) [33, 24].

Prefetching for pointer-chasing traversals uses one of four approaches. The first approach inserts additional pointers, called *jump pointers*, into the data structure of the application to connect non-consecutive list elements [27, 48, 32], as will be described in detail in Section 4.3. The second approach uses natural pointers for prefetching [47, 34, 32]. This technique prefetches pointer chains sequentially, but schedules 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* [25], 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 list nodes. This approach was first proposed in [32] and is called *data linearization prefetching*. This is what is called in this thesis "index prefetching" technique and is

evaluated in Section 7.3.

Data locality has been studied extensively in the literature. Computation-reordering transformations such as loop permutation and tiling are the primary optimization techniques [56]. Chapter 5 will discuss these techniques in greater detail. Data layout optimizations such as padding and transpose have been shown to be useful in eliminating conflict misses and improving spatial locality [43]. Padding is studied as an enhancement to software prefetching in Section 7.2. Several cache miss estimation techniques have been proposed to help guide data locality optimizations [20, 56]. Tiling has been proven useful for linear algebra codes [30, 56, 14] and multiple loop nests across time-step loops [51]. In comparison, tiling for 3D stencil codes is applied in these benchmarks which cannot be tiled with existing methods. Tiling for 3D and 2D arrays is discussed in detail in Section 5.1.

Researchers have examined irregular computations mostly in the context of parallel computing, using the run-time [16] or compiler [31] to support accesses on message-passing multiprocessors. A few have also looked at techniques for improving locality [1, 17]. The techniques for irregular computations are discussed in detail in Section 5.2.

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 [12]. Further extensions are introduced in this work to use this technique in conjunction with software prefetching. Also, index prefetching utilized this allocation methodology

as discussed later in Section 7.3. Calder*et al.* use profiling to guide layout of global and stack variables to avoid conflicts [6]. Carlisle *et al.* investigate parallel performance of pointer-based codes in Olden [8].

Chapter 3

Memory Access Patterns

The type of software prefetching and locality optimizations to use depend on the memory access pattern of the application code. This Chapter discusses three common memory access patterns that occur in the benchmarks used by this thesis. In the following sections, affine array accesses, indexed array accesses and pointer-chasing accesses are discussed.

3.1 Affine Array Accesses

Affine array access is the most basic access pattern. This pattern arises when traversing arrays, as shown in Figure 3.1 Part(A). This figure shows the 2D Jacobi code. All the array elements accessed are statically known at compile time since the indices of the arrays are linear functions of the loop induction variable. The access pattern used in the 2D Jacobi code fragment is usually called a stencil. In a stencil, three columns are needed to perform the computation. It is necessary to have all three columns in the cache for high performance. Figure 3.2 shows the computation elements and how the computation progresses in the array. $3D$ solvers suffer from very bad cache

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

Part(A)

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

Part(B)

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

Part(C)

Figure 3.1: Example of 3 different access patterns. Part(A) Affine Array Access example code. Part(B) Indexed Array Access example code. Part(C) Pointer-Chasing Access example code.

performance [4, 52, 55]. A version of JACOBI extended to 3 dimensions is shown in Figure 3.3. This figure shows a pictorial image of the computation order. The dark squares are the neighboring elements that are averaged. The computation averages 6 elements: 4 in the main plane and one element for each plane above and below the main plane as shown in the figure. For high performance in this case, the cache needs to hold three entire $N \times N$ planes. Assuming double precision data and a write-through cache so that array A doesn't interfere with B, a 16K Byte L1 cache can hold a 3D array of size $26 \times 26 \times M$, where M is the third dimension of the array which can be any number. As the problem size increases the problem gets worse. In comparison, the same sized cache can hold data up to $682 \times M$ array in case of a 2D version of JACOBI.

Affine array accesses are common in dense-matrix codes such as linear algebra and PDE solvers. Such access patterns can also be found in image

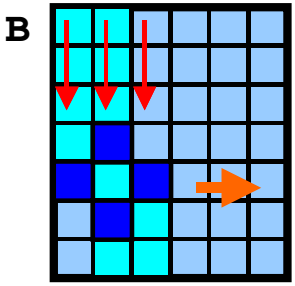


Figure 3.2: The computation order and direction of progress in 2D JACOBI.

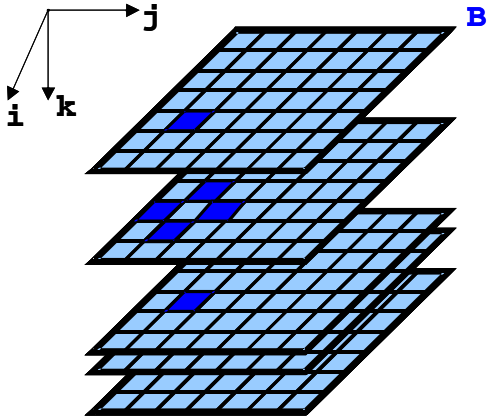


Figure 3.3: The computation order and direction of progress in 3D JACOBI.

processing and signal processing codes; they are particularly common in DSP applications. An important feature of affine array accesses is that they allow

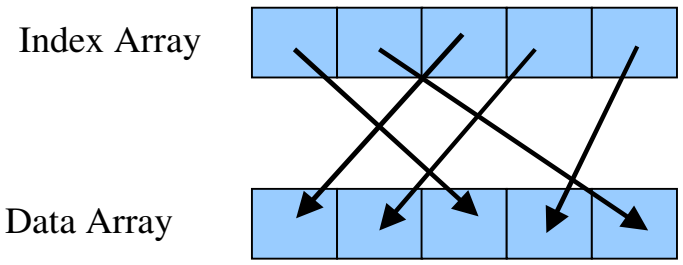


Figure 3.4: Showing the relationship between the index and indexed arrays before inspector-executor runs.

known. The data array, which is the main computation array, is indexed by another array which is called the *index array*. This pattern results in an irregular access pattern since the data array elements accessed depend on the contents of the index array.

Figure 3.1 Part(B) is a simple piece of code extracted from one of the applications that uses indexed arrays. The array named *index* is accessed in an affine manner similar to the arrays in the previous section. The two arrays $X1$ and $X2$ are indexed by the contents of the *index* array. The accesses are irregular due to the randomness of the data stored in the array *index*. The cache performance of applications using indexed arrays can be poor since both spatial and temporal locality in such applications is typically low due to the irregularity of the access pattern.

Figure 3.4 shows the relationship between the index array and the indexed "data" array. Unlike Affine array accesses, which can be improved using compile-time transformations, indexed array accesses cannot be improved at compile-time since the *index* array values are known only at run-time. To improve the performance of such applications, run-time transformations are

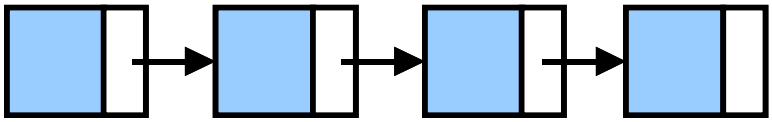


Figure 3.5: Chasing pointers to access list nodes.

required to change the access patterns dynamically.

3.3 Pointer-Chasing Accesses

The third and final access pattern is pointer-chasing accesses. As the name suggests, this access pattern is characterized by pointer dereference operations. This results in an access pattern that is as random in nature as indexed arrays discussed above. Applications using linked lists, trees, and graph data structures are examples of codes that produce such access patterns.

Figure 3.1 Part(C) shows an example of creating and traversing a singly-linked list. The list nodes are dynamically allocated at run-time. The length of the list is usually a parameter that is known only at run-time. Figure 3.5 can be used to help describe the list access pattern. As shown in this figure, the list nodes are linked through pointers. To traverse the list, the pointers are dereferenced one after another in a serial manner. This access pattern cannot be analyzed by the compiler to improve its locality since the pointer locations are not known statically. Memory locations are known only at run-time which makes the locality optimizations of these types of codes possible at run-time only. Whereas, prefetching is not hindered by the fact that the memory location are not known at compile-time but is hindered by the fact that

the accesses are inherently serial. In pointer-chasing codes, it is common for spatial and temporal locality to be low. Thus, the cache behavior of these applications can be poor. The pointer-chasing accesses force the accesses to be sequentialized since the next node cannot be accessed until the pointer pointing to that node is found in the current node. Pointer-chasing codes are quite common in applications such as databases, and advanced pointer-based data structures found in many applications.

Chapter 4

Software Prefetching

Software prefetching is a well-known prefetching technique for pre-loading data into the cache from memory. Software prefetching relies on the programmer or the compiler to insert explicit prefetch "Pre-load" instructions and schedule them far enough in advance to hide or "tolerate" the latency of the memory accesses. The job of the compiler or the programmer is to identify memory accesses that are likely to miss in the cache, and to issue a prefetch for that piece of data to avoid stalls due to cache misses.

There exist several techniques to implement software prefetching for affine array codes [7, 28, 38, 37]. These techniques can be easily extended for indexed array codes [40]. Researchers have also proposed software prefetching techniques for pointer-chasing applications [27, 48, 47, 34, 32].

The advantage of software prefetching is that it is controlled by software and hence does not need much special hardware support. The only support needed is lockup-free caches which allow multiple outstanding misses, and ISA support in the form of a prefetch instruction. Consequently, software prefetching is relatively cheap compared to hardware prefetching.

In this thesis, three software prefetching algorithms proposed previously in the literature for the three different memory access patterns described in Chapter 3 are studied both in isolation and in combination with the appropriate locality optimization technique. This chapter describes the three software prefetching algorithms.

4.1 Affine Array Prefetching

Affine array prefetching inserts prefetches for loops traversing affine arrays. The algorithm used is Mowry's algorithm [38]. In this thesis, Mowry's algorithm is applied by hand, even though it has been automated by a compiler. This algorithm prefetches only the missing data items. The algorithm is comprised of three steps. The first is to identify the instances of the data items that are going to miss in the cache. The second is to perform loop unrolling and loop splitting to isolate the memory references that will miss. The degree of loop unrolling is determined by the size of the data element and the size of the cache block. The third and final step is to schedule the prefetches such that they arrive in cache just prior to being accessed by the computation. The distance (in loop iterations) a prefetch instruction is scheduled before its consumption is called the *prefetch distance*. The 2D Jacobi kernel from Figure 3.1 Part(A) has been instrumented with Mowry's algorithm. The instrumented code appears in Figure 4.1. Figure 3.2 can be used to visualize the computation order and progress for 2D JACOBI.

The first step is to identify the references that will miss in the cache. The

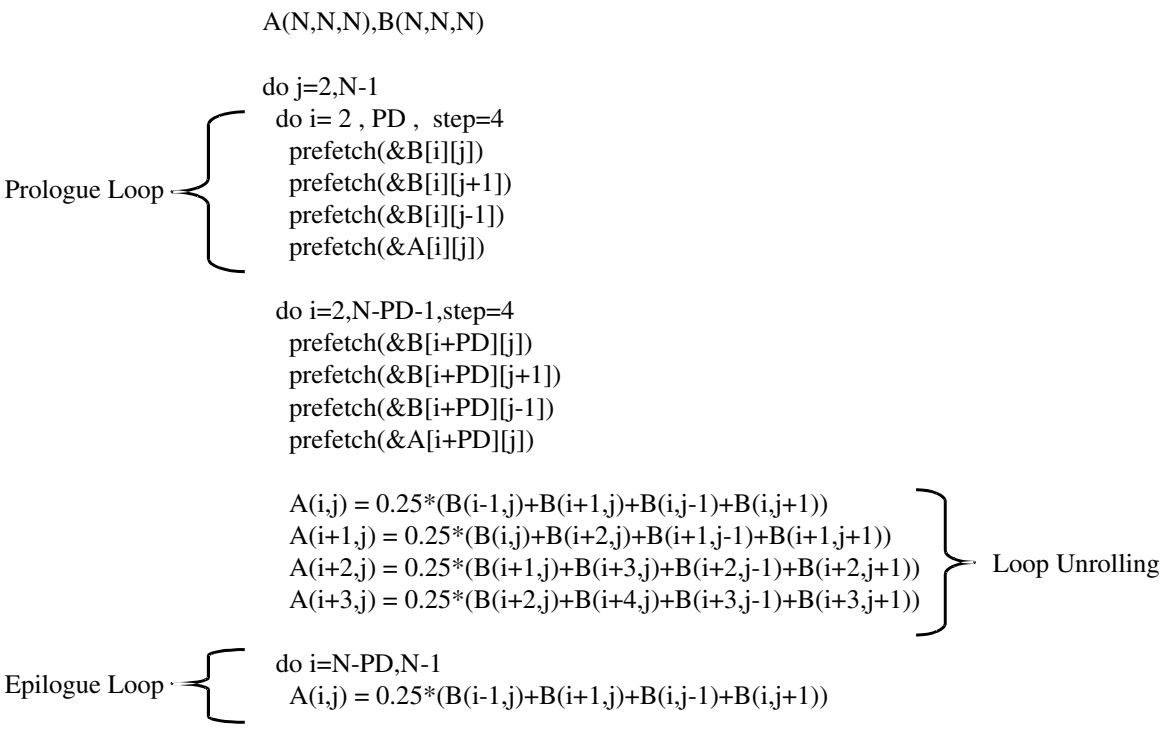


Figure 4.1: Example affine array prefetching for the 2D Jacobi kernel using Mowry's algorithm [38]. The prefetch algorithm involves three steps: loop unrolling, prefetch scheduling, and loop peeling.

missing references are those in the innermost loops. Using compiler analysis to determine the locality of the data access pattern, the missing elements are identified. In the unrolling step, the loop is unrolled to expose the leading memory reference from each cache block that will cause a cache miss. Only the missing reference from a cache block is prefetched. When the prefetch comes back, the whole cache block will be brought to the cache and thus no further prefetch instructions for any of the elements belonging to that cache block are necessary. Thus, the loop unrolling step minimizes prefetch overhead since only one prefetch is issued per cache block.

The 2D JACOBI code in Figure 4.1 shows the prefetch instructions, loop unrolling, and the scheduling of the prefetch instructions. Each statement in the code reference four B array elements and one A array element. All elements referenced in the same statement lie in different cache blocks except for the two elements $B(i-1,j)$ and $B(i+1,j)$ which lie on the same cache block, thus an unnecessary prefetch is saved. The compiler or the programmer can figure out that only four prefetches are needed for each statement, assuming a data element size of 8 bytes and a 32-byte cache block. Hence the loop is unrolled four times.

Scheduling the prefetches is necessary since there isn't enough work in a single unrolled loop iteration under which to hide the memory latency. Thus, the prefetch instructions need to be issued some number of loop iterations in advance to give them enough slack to hide the cache miss. The distance in loop iterations necessary to hide a cache miss latency is called the *prefetch distance*. Computing the prefetch distance is done using the formula $\lceil \frac{l}{w} \rceil$; l is the memory latency and

w is the work in one loop iteration, as mentioned in Chapter 1. Since every prefetch instruction needs to be scheduled exactly PD iterations ahead, a "prologue loop" should be inserted before the main computation loop to prefetch the first PD elements. Similarly, the last PD iterations will not need any prefetching. Thus, an "epilogue loop" needs to be inserted after the main loop computation to perform the last PD iterations without prefetching. The transformation to handle the first and last PD loop iterations in the prologue and epilogue loops is called "loop peeling".

Imagine a pipeline that executes the loop iterations. In such a pipeline, the prologue loop is filling up the pipeline with the first few data elements needed, and the *epilogue loop* executes the last PD iterations without prefetching, essentially draining the pipeline. Figure 4.1 illustrates the prologue and epilogue loops created by the loop peeling transformations.

4.2 Indexed Array Prefetching

Indexed array accesses, of the form $X1(index(i))$, are very similar to affine array accesses except they have a single level of indirection since each reference is actually two references performed back to back. The algorithm used to instrument prefetching for indexed array accesses is also the Mowry algorithm [40]. Figure 4.2 shows a simplified molecular dynamics kernel (very similar to the MOLDYN, IRREG, and NBF benchmarks used in the performance evaluation) after all the necessary transformations have been applied.

The strategy is to prefetch two back-to-back (serialized) prefetches, one

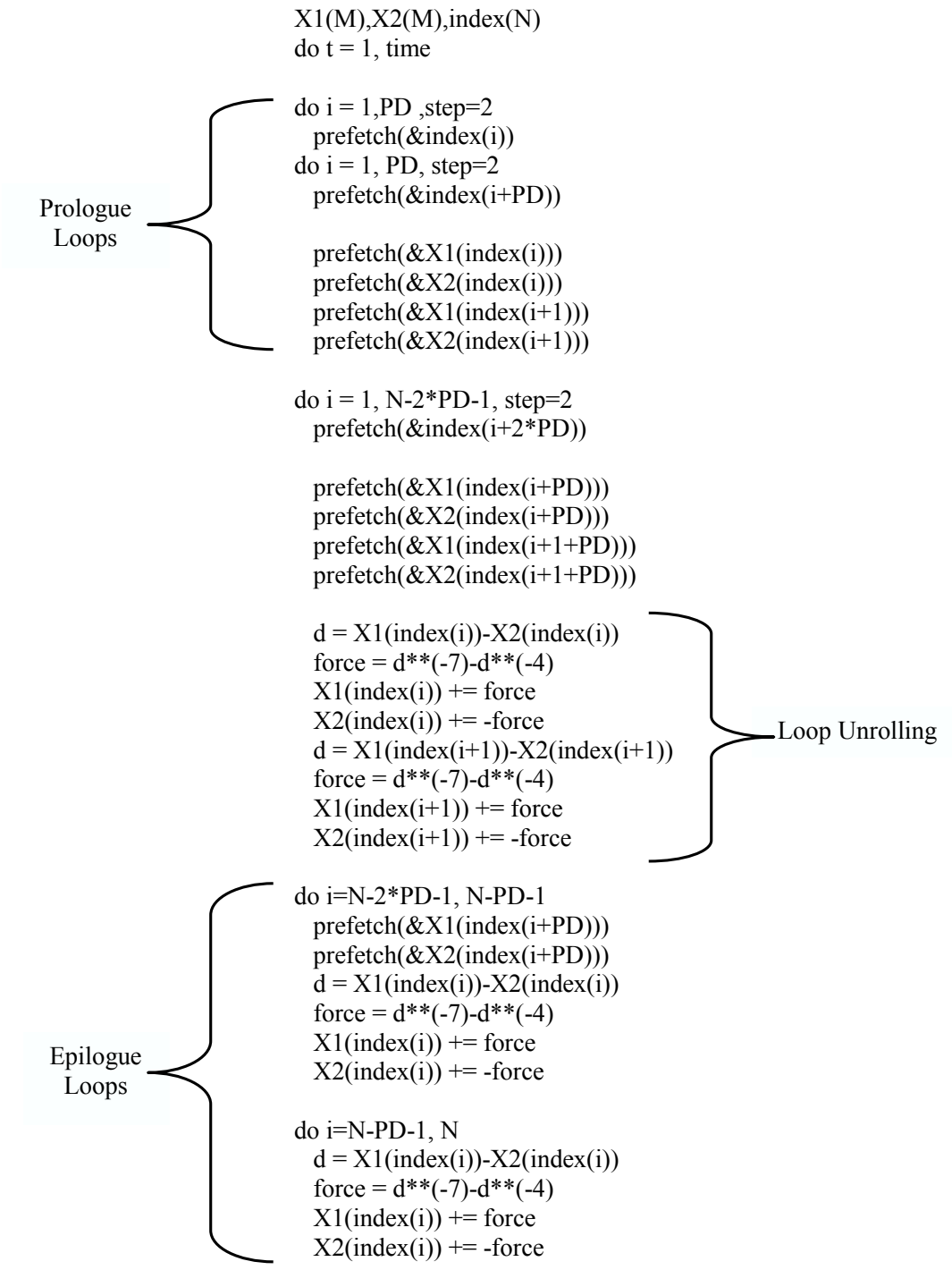


Figure 4.2: Example indexed array prefetching for a molecular dynamics kernel using the algorithm in [38]. The prefetch algorithm is similar to the algorithm for affine arrays, with several extensions to handle indexed arrays.

for the index array value, and another for the data array value. The index array can be prefetched like the affine array case, as described in Section 4.1. Then, when the prefetch for that index array element completes, a prefetch for the data array can be issued using the index array data that came back. Thus, the index array prefetches should start PD iterations before the data array. In addition, two prologue and epilogue loops are required, as shown in Figure 4.2: one loop to start the index array prefetches before the data array, and one loop to start the data array prefetches. Then, the computations start.

Looking back at the pipeline example mentioned above in explaining how prefetching works, there are two prefetch pipelines with two different prefetch distances for indexed arrays. Thus, at the end of the computations, two epilogue loops are needed: one to prefetch the last PD entries of the data array while performing the computations for $2\times PD$ iterations before the end of the computations, and another final loop to perform only the remaining PD computations at the end of the array.

Loop unrolling is used to reduce the number of prefetches in the index array. Loop unrolling is not effective at reducing the prefetch overhead for the data array since the compiler cannot figure out which elements of the data array belong to the same cache block. Hence, the compiler must conservatively schedule a prefetch for every reference of the data array increasing the prefetch overhead for this array. The loop unrolling degree is 2 in Figure 4.2 only to limit the size of the example code. Thus, loop unrolling will help the index array only since it is treated as an affine array. The *prefetch distance* is computed in the

same way as affine array codes.

4.3 Pointer-Chasing Prefetching

Prefetching pointer-chasing accesses is the most challenging of the three prefetching techniques. The problem with pointer-chasing accesses is that the memory references performed along a pointer chain are inherently serial. As shown in Figure 3.5, each node of the list can be accessed only after all previous list node pointers have been dereferenced sequentially. The serial nature of pointer references is known as the *pointer-chasing problem.*

One promising technique for addressing the pointer-chasing problem is *jump pointer prefetching* [48, 32]. In this technique, the list data structure is modified to permit prefetching of list nodes further down the pointer chain without traversing the intermediate list nodes. Jump pointer prefetching instruments the list nodes with extra pointers, called *jump pointers.* As this name suggests, jump pointers point to some number of nodes down the list to permit access to later nodes. Similar to previous prefetching techniques, the distance between where a jump pointer originates and where it points to is the prefetch distance. Jump pointers break the sequentiality of the list node accesses. Computing the prefetch distance in jump pointer prefetching is the same as in other prefetching techniques.

Figure 4.3-Part A shows a "while" loop that has been instrumented with jump pointer prefetching. Figure 4.4 shows how prefetch pointers are inserted into a list, and how these prefetch pointers point to list elements further down

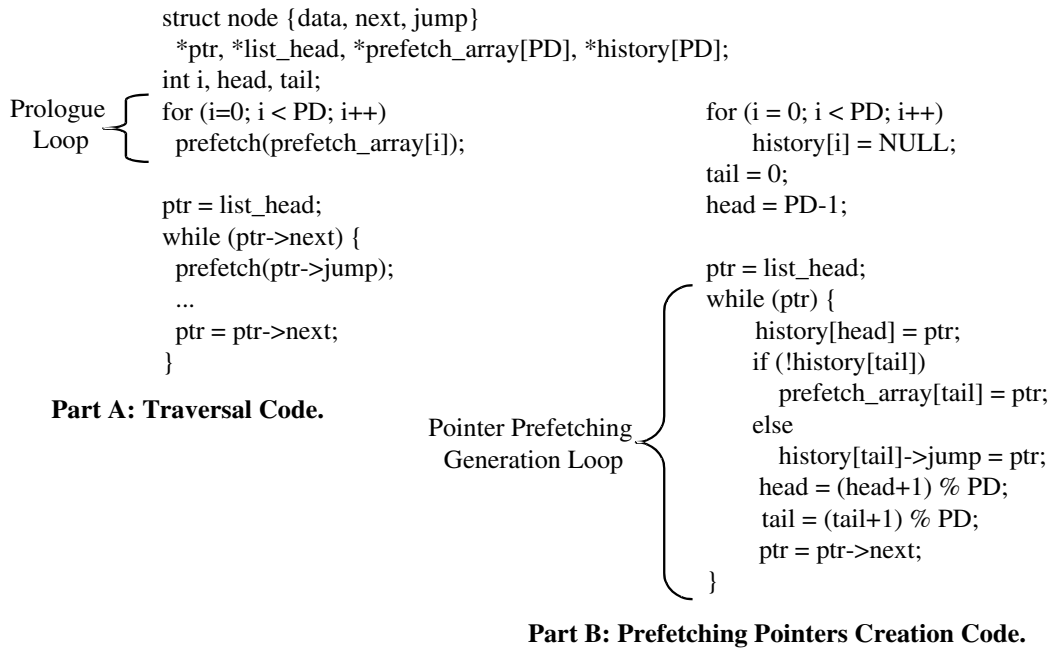


Figure 4.3: Example pointer prefetching for a linked list traversal using jump pointers and prefetch arrays [27]. Part(A) shows the traversal code instrumented with prefetching through jump pointers and prefetch arrays. Part(B) shows the prefetch pointer initialization code.

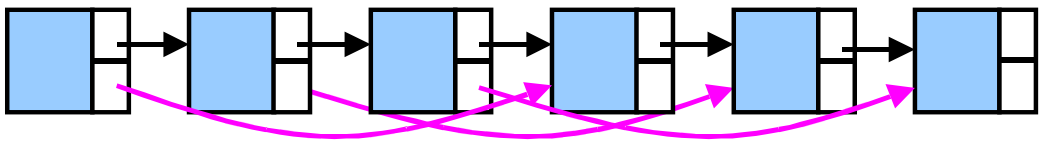


Figure 4.4: Jump Pointers inserted into the list nodes.

the list, this breaking the sequentiality of pointer accesses.

One problem with jump pointers is that there are no jump pointers pointing to the first PD nodes of the list. Thus, a technique is needed to help generate something similar to the prologue loop in affine and indexed arrays. This technique is called *prefetch arrays* [27]. In this enhancement to jump pointer prefetching, an array of pointers is constructed to point to the first PD elements of the list, and a "prologue loop" is added to prefetch the first PD elements

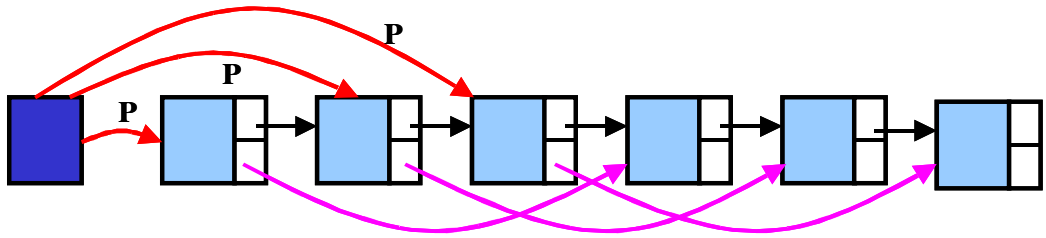


Figure 4.5: Prefetch Array pointers labelled "P" added to the list nodes already having jump pointers.

using the array of pointers. Figure 4.5 shows prefetch array pointers for the linked list example, and labels them with the letter "P". Figure 4.3-Part A shows the additional prologue loop code needed to issue prefetches through the prefetch arrays.

Before prefetching can start, the prefetch pointers must be set. Figure 4.3-Part B shows an example of prefetch pointer creation code which uses a *history pointer array* [32] to set the prefetch pointers. The history pointer array, called "history" in Figure 4.3-part B, is a circular queue that records the last PD link nodes traversed by the creation code. Whenever a new link node is traversed, it is added to the head of the circular queue and the head is incremented. At the same time, the tail of the circular queue is tested. If the tail is NULL, then the current node is one of the first PD link nodes in the list since PD link nodes must be encountered before the circular queue fills. In this case, we set one of the "prefetch array" pointers to point to the node. Otherwise, the tail's jump pointer is set to point to the current link node. Since the circular queue has depth PD, all jump pointers are initialized to point PD link nodes *ahead*, thus providing the proper prefetch distance.

Normally, the compiler or programmer ensures the prefetch pointer initialization code gets executed prior to prefetching, for example on the first traversal of a linked list data structure. Furthermore, if the application modifies the linked data structure after the prefetch pointers have been initialized, it may be necessary to update the prefetch pointers either by re-executing the initialization code or by using other fix up codes, adding extra overhead to the pointer prefetching technique.

Chapter 5

Locality Optimizations

Locality optimizations are the second technique to be evaluated in this thesis. This techniques are orthogonal to software prefetching. Software prefetching tries to hide the latency of references by issuing loads early for those references that are expected to miss in the cache. Locality optimizations try to change data layout and/or computation order of the programs so that the application's *data locality* is increased [56]. Data locality optimizations improve the application's data reuse which already exists but is not exposed. Changing the computation order and data layout of the program at compile and/or run-times exposes this data reuse.

Reuse comes in two forms. One is temporal reuse where data accesses to the same location of memory are repeated in time. Temporal reuse can be exploited by reordering the computations to finish all computations on a particular element before moving on to the next element. The second is spatial reuse where data accesses to nearby locations are performed together. Spatial reuse can be exploited by reordering the computations to perform computations on elements that are close in space before moving on to other computations.

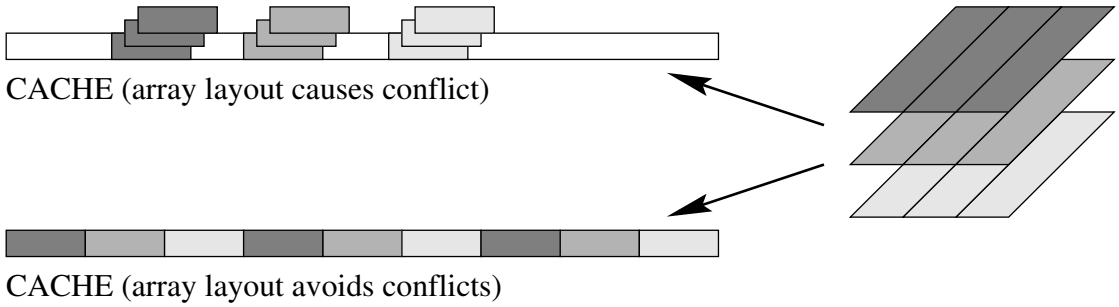


Figure 5.1: Example of conflict misses under two array layouts.

```
// Tiled 3D Jacobi                    //Inspector-Executor for          // Pointer-Based Structures
A(N,N,N),B(N,N,N)                     //Molecular Dynamics              // (Linked List Traversal)
do kk=2,N-1,TK   } TIxTJxTK           inspect_reorder(&E(2,N))        struct node{val, next} *ptr, *list;
do jj=2,N-1,TJ   }   Tile             do t = 1, time                  while (...) {
do ii=2,N-1,TI   }                      if (recalc)                     ptr->next = ccmalloc(node);
Tiled  do k=kk,kk+TK-1                    E(...) = ...                  ptr = ptr->next;
Loops  do j=jj,jj+TJ-1                do i = 1, N                     ptr->val = ... ;
       do i=ii,ii+TI-1                  d = X(E(1,i))-X(E(2,i))       }
         A(i,j,k) = 0.16667 *           force = d**(-7)-d**(-4)       while (ptr->next) {
          (B(i-1, j, k) + B(i, j-1, k)+ X1(E(1,i)) += force              ptr = ptr->next;…;
           B(i+1, j, k) + B(i, j+1, k)+ X2(E(2,i)) += -force           }
           B(i, j, k-1) + B(i, j, k+1))
```

Part(A) **Part(B)** **Part(C)**

Figure 5.2: Example Locality optimized codes for affine array, indexed array and pointer-chasing codes.

In data locality optimizations, the compiler or run-time code examines the application to see what type of data locality can be exploited to improve the application's cache performance. Several data locality optimizations have been proposed in the literature. These optimizations target different access patterns. In the next three sections, locality optimizations for the three access patterns discussed earlier in Chapter 3 will be explained.

5.1 Tiling for Affine Accesses

Locality optimizations for affine array accesses are straightforward to apply because the access patterns can be analyzed exactly at compile time. The

transformation for such access patterns is called *tiling*. In tiling, loop permutation and strip-minning are applied to arrange the access pattern into small tiles that can fit completely in the cache [56]. The main idea is to instrument the innermost loop to make the accessed data fit in the cache and thus achieve the requirement of better reuse. A significant problem with tiling techniques is that conflict misses can occur. Such misses will cause tile data to be evicted from cache before they are fully reused [30]. This effect is shown in Figure 5.1.

Figure 5.2 part(A) shows tiling applied to the 2D Jacobi code introduced in Chapter 3. The goal in tiling for Jacobi is to keep the three columns needed for each computation in the cache. To address the cache conflicts problem, *tile size selection* and *array padding* can be applied to avoid conflict misses in tiles [14, 42, 44].

Avoiding conflicts in tiling problems, especially for 3D problems is discussed in Rivera *et al.* [45]. They consider 4 possible solutions to reduce conflict misses when applying tiling. They dismiss 3 of these solutions and are left with one good solution. Tile size selection avoids conflicts by carefully selecting tile dimensions tailored to the particular array dimensions so that no conflicts occur. An algorithm called the Euclidean remainder algorithm is used to compute sequences of nonconflicting tile dimensions [14, 44] for 2D arrays. Also, an extension of this technique is discussed in [45] to apply this algorithm to 3D arrays. A cost function is required in these algorithms to pick one tile size out of all the generated tile sizes. A greedy algorithm is used to search for improved tile size candidates by increasing the tile dimensions and testing for conflicts. The

tile size is continually extended until no more extensions are possible without introducing conflicts.

Even when choosing non-conflicting tiles, performance of tiling may suffer for certain array dimensions. For instance, given a $341 \times 341 \times M$ array, the best tile size available is (110,4). The problem with this tile size is that the second tile dimension is small. When one or more tile dimensions are small, performance suffers. Note that the original problem was doing computations assuming a tile size of (341,1), the new tile size won't be effective enough in improving the locality of the application. The solution is to use padding to enable better tile sizes [43]. Padding algorithms for picking the optimal tile size are NP-complete. However, good tile sizes can be obtained using heuristics that have reasonable complexity.

5.2 Reordering for Indexed Accesses

As was discussed previously in Chapters 3 and 4, indexed array accesses are difficult to optimize since compile-time analysis cannot determine the access patterns. Instead, indexed array accesses must be optimized at runtime [1, 17, 35, 36]. Saltz *et al.* designed a compiler which generates calls to an inspector to process memory access patterns at run-time. The same approach can be used to improve the locality of indexed array access patterns.

Figure 3.4 shows the relationship between the index array and the data (indexed) array. Indexed array accesses can be optimized using an inspector-executor approach [16] as shown in Figure 5.2 part(B). The outcome of

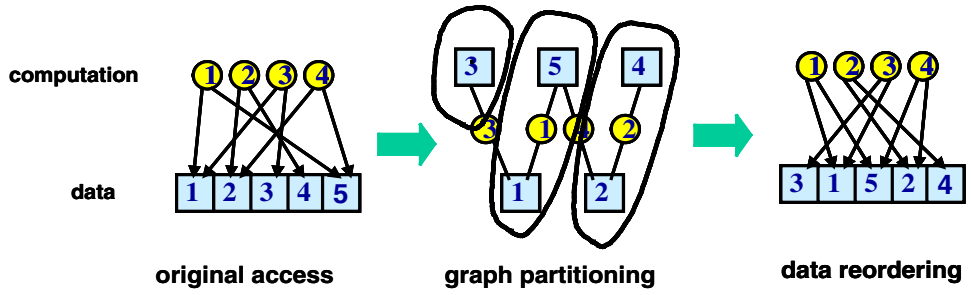


Figure 5.3: Indexed Arrays Reordering technique.

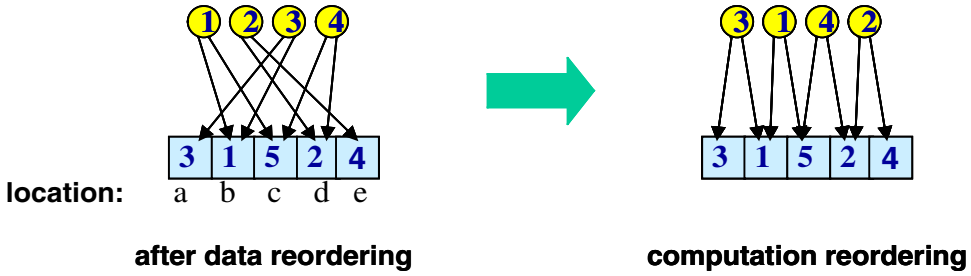


Figure 5.4: Indexed Array computation reordered after the reordering operations.

such a transformation is a change of the layout of the data that forces the access to the data array to be more regular, thus resulting in better cache performance.

The reordering process is shown in Figure 5.3. This figure shows the partitioning and the reordering of the data and index arrays taking place. Since almost all molecular dynamics and electromagnetic codes interact pairs of data according to their geometric coordinate data. The problem lends itself to a directed graph. Several data and computation locality transformations exist to solve the problem of improving the locality of such graphs. One of these techniques is called Graph Partitioning techniques (GPART). This technique is based on hierarchical clustering. It generates quality partitions like the ones in

Figure 5.3 quickly. The main advantage of this technique is that it has low overhead since it only considers edges between partitions. GPART closely matches the performance of more sophisticated partitioning algorithms, with one third of the overhead [1, 35, 21].

An example is shown in Figure 5.4. Circles represent computations (loop iterations), squares represent data (array elements), and arrows represent data accesses. Initially, memory accesses are irregular, but either computation or data may be reordered to improve temporal and spatial locality. Note that each iteration accesses two array elements. Computations can be viewed as edges connecting data nodes, resulting in a graph. Locality optimizations can then be mapped to a graph partitioning problem. Partitioning the graph and putting nodes in a partition close in memory can then improve spatial and temporal locality. Applying lexicographic sorting after partitioning captures even more locality.

The hierarchical structure in GPART is similar to that of recursive coordinate bisection (RCB), which is a data reordering algorithm used when data is unevenly distributed. RCB is based on geometric coordinate information. RCB recursively splits each dimension into two by finding the median of the data coordinates in that dimension. After partitioning, data items are stored consecutively within each partition. Loop iterations are lexicographically sorted based on the data accessed [17, 21, 36]. RCB has higher overheads than other techniques but is most likely to work well with unevenly distributed data. The right half of Figure 5.4 shows the computations after the partitioning technique

has been applied and shows how the computations will access the array in a different order compared to the original access order. These algorithms are defined in more detail in [21, 22].

5.3 Memory Allocation For Pointers

Pointer-based applications typically suffer from poor cache performance just like indexed array codes since the allocation of these data structures is dynamic and typically exhibits low spatial locality. Pointer-based applications are harder to optimize than indexed array codes because of their dependence upon pointers and dynamic allocation of new data items. Due to the pointer chasing problem, link nodes must be traversed sequentially, as explained in Sections 3.3 and 4.3.

Cache-conscious allocation has been introduced to improve the locality of such accesses [6, 12]. This technique packs nodes of data that are logically contiguous onto the same cache line so that they are physically contiguous in memory, thus increasing spatial locality. The node packing is achieved using a custom memory allocator called "CCMALLOC".

Figure 5.5 shows how nodes from a linked list are allocated in a cache-conscious fashion using CCMALLOC. Figure 5.2 part(C) shows a simple list allocation code that uses CCMALLOC to allocate list nodes on nearby cache blocks dynamically. The modification is quite simple: replacing MALLOC with CCMALLOC. This optimization is applied to all pointer-chasing benchmarks used.

CCMALLOC works in the following way: it takes a pointer as an

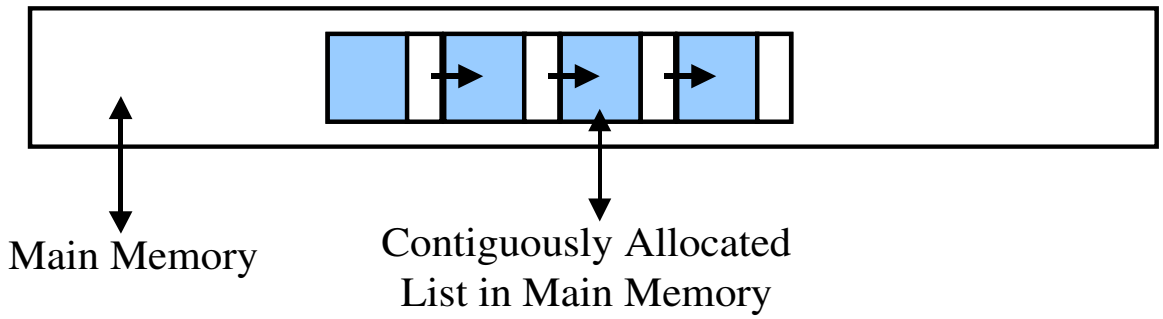


Figure 5.5: Linked list contiguously allocated in Main Memory.

argument and allocates current and future nodes close to it. It reserves space for future data blocks when allocating the first node [12].

Note one problem with CCMALLOC is that dynamic data structures which change after allocation may not benefit from this optimization. Frequent insert and delete operations after allocation will make logically contiguous nodes physically non-contiguous.

Chapter 6

Experimental Evaluation

In this Chapter, the performance of software prefetching as well as that of locality optimizations is evaluated independently and when combined together naively. The experimental methodology in doing the evaluation is explained in detail. Then, the results for software prefetching and locality optimizations under different memory bandwidths and latencies are shown for all 9 benchmarks studied. Finally, the naive combination is discussed. Later in Chapter 7, enhancements for more effectively combining software prefetching and locality optimizations are presented and evaluated.

6.1 Methodology

In this section, the methodology used to evaluate the different techniques is presented. This thesis evaluates 9 benchmarks. The evaluation considers different versions of each benchmark: original, prefetching, locality optimized, and the combined version which includes both prefetching and locality optimizations. All experiments are performed on the same cycle accurate simulator, and all benchmarks were run to completion.

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
NBF	144K mols	Indexed array
HEALTH	5 levels, 500 iters	Pointer-chasing
MST	1024 nodes	Pointer-chasing
EM3D	10K nodes	Pointer-chasing

Table 6.1: Benchmark summary.

As discussed in Chapter 3, three different access patterns are evaluated. For each access pattern, three benchmarks are instrumented and evaluated. Table 6.1 lists all the benchmarks used along with their problem sizes and memory access patterns.

The Affine array benchmarks are MATMULT, which multiplies two matrices, REDBLACK, which performs a 3D red-black successive-over-relaxation, and JACOBI, which performs a 3D JACOBI relaxation. Both JACOBI and REDBLACK are frequently found in PDE solvers, such as MGRID from the SPEC/NAS benchmark suite.

The indexed array benchmarks are IRREG, which is an iterative PDE solver for an irregular mesh, MOLDYN, which is abstracted from the non-bonded force calculation in CHARMM [16], a key molecular dynamics application used at NIH to model macromolecular systems, and NBF (Non Bonded Force kernel), which performs a molecular dynamics simulation. NBF is taken from the GROMOS benchmark suite [54].

Finally, the pointer-chasing benchmarks are HEALTH, which simulates the

Columbian health care system, MST, which computes a minimum spanning tree, and EM3D, which simulates electromagnetic wave propagation through 3D objects. HEALTH, MST, and EM3D are from the OLDEN benchmark suite [46].

For reasons that will appear later on in the results provided for EM3D, a more detailed explanation of what the application is actually doing and how it is doing it is introduced here. The major data structure of EM3D is an array that contains the set of magnetic and electric nodes [15] since Electromagnetic waves travel through space as alternating magnetic and electric fields. EM3D simulates these fields with two sets of nodes: enodes and hnodes, where each set of nodes corresponds to one of the fields. The area or surface on which the electromagnetic field is to be simulated is modeled by a mesh of the e and hnodes. Each node is given an initial value, and a list of it's neighboring nodes. (enodes have hnodes for neighbors and hnodes have enodes for neighbors.) Each node is connected to each of its neighbors via an edge that has an associated coupling coefficient. The list of nodes, edges, and the edge coefficients form a bipartite graph [50].

The original code is instrumented by hand to generate the prefetching, locality optimized and the combined versions of each application. These codes are compiled for our target architecture which is based on the SimpleScalar tool set [5] and models a 1GHz 4-way issue dynamically-scheduled processor. The simulator simulates all aspects of the processor, including the functional units, the reorder buffer, the branch predictors, register renaming, the instruction fetch unit, the load-store unit, the caches (data and instructions), and of course, the register file. The simulator is cycle accurate. The original memory system model

from the SimpleScalar tool set was modified to account for the contention on the L2-memory system bus. It was assumed that the L1-L2 bus link has infinite bandwidth. No MSHRs are modeled. This approach maximizes the concurrency in the memory system to expose memory bandwidth limitations. Also, a prefetch instruction was added to the ISA of the processor.

The experiments were done using the following cache organization: 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. The latency of the L1 cache is one cycle, while the L1-L2 bus latency is 7 cycles and has infinite bandwidth. Although the cache sizes are small, they are matched to the small problem sizes used for the benchmarks in order to limit the simulation time.

Using this simulator model, each benchmark version is evaluated with different L2-memory latencies and different memory bandwidths to study the effects of memory system parameters on software prefetching and locality optimization performance. The L2-memory latency is varied from 80 to 640 cycles in powers of two, and memory bandwidth is varied from 1 Gbytes/sec to 64 Gbytes/sec also, in powers of two. The lower end of both the latency and bandwidth ranges simulated captures the trends of existing memory systems which have latencies of around 100 cycles and bandwidths of around 2-3 GBytes/sec. The mid and high end of the latency and bandwidth ranges simulated capture the characteristics of future architectures. All these number are capturing the trends in single processor system and doesn't extend to multi-processor systems.

Latency in Cycles	RedBlack	Jacobi	Matmult
80	12	8 , 36	24
160	24	16, 68	44
320	48	28, 136	88
640	96	56, 268	176

Table 6.2: Prefetch distances for RedBlack, Jacobi and Matmult for the different latencies.

Latency in Cycles	Irreg	Moldyn	NBF
80	8 , 20 , 20 , 40	1, 1, 2	2
160	12, 40 , 40 , 80	2, 2, 3	4
320	24, 80 , 80 , 160	4, 4, 5	8
640	44, 160, 160, 319	7, 7, 9	16

Table 6.3: Prefetch distances for Irreg, Moldyn and NBF for the different latencies.

Latency in Cycles	Health	MST	EM3D
80	31	3	2
160	62	3	3
320	124	3	6
640	247	3	11

Table 6.4: Prefetch distances for Health, MST and EM3D for the different latencies.

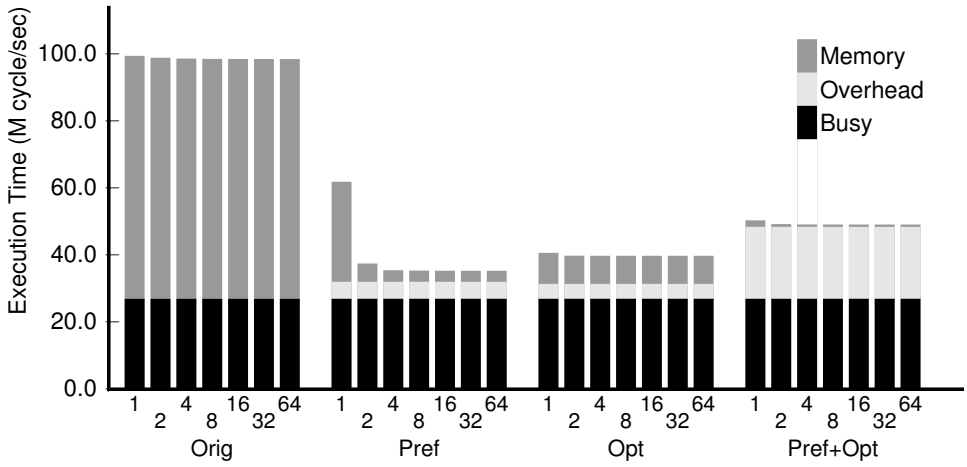
Table 6.2 shows the prefetch distances of the three affine array benchmarks for the different latencies. These prefetch distances are computed for the original codes when instrumented with prefetching. Note that in JACOBI, there are two loops instrumented with prefetching; hence, there are two prefetch distances, one for each loop. For every benchmark, the prefetch distances are reported with respect to the four memory system latencies simulated. Table 6.3 shows the prefetch distances for all the indexed array benchmarks used in the thesis. IRREG has four loops that have been prefetched. MOLDYN has only three loops, while NBF has only one loop.

Table 6.4 shows the prefetch distances for the pointer-chasing benchmarks used in this thesis. Note that the prefetch distances for HEALTH are extremely large. The prefetch distance for a latency of 640 cycles is 247, which is large compared to the length of the list nodes of HEALTH (around 120 nodes). Prefetching has no impact on HEALTH at this high latency since no list nodes are prefetched when the prefetch distance exceeds the size of the linked lists.

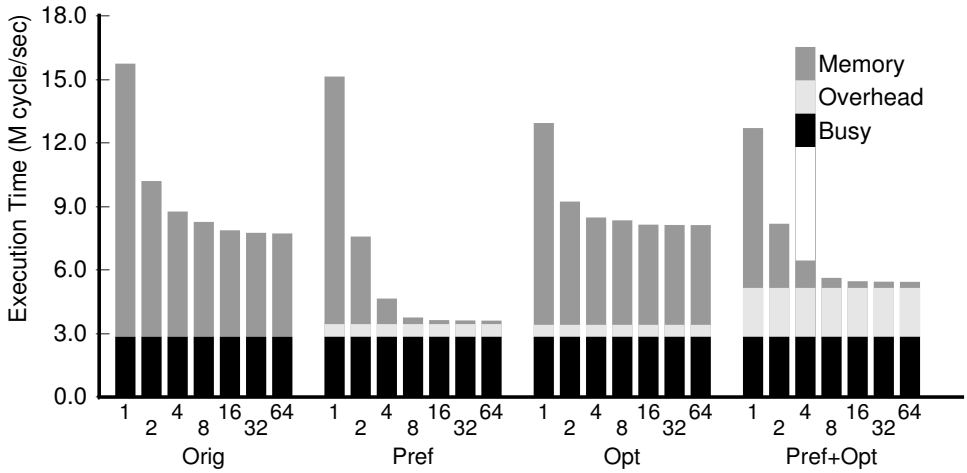
6.2 Varying Memory Bandwidth

In this section, the results for a fixed memory latency of 80 cycles are discussed while varying the bandwidth from 1 Gbytes/sec-64 Gbytes/sec. Results are shown for the four versions of each benchmark (original, prefetching, locality optimization, and combined), but the combined results will be discussed in a later section.

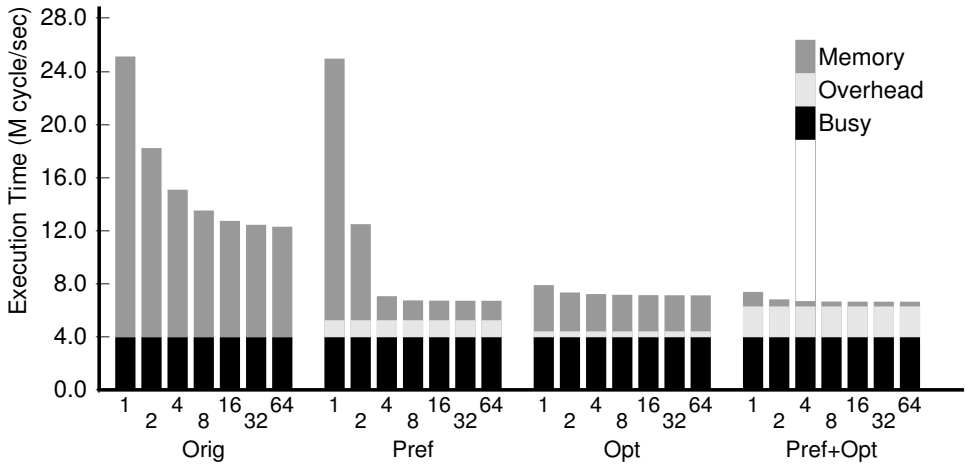
Figures 6.1, 6.2, and 6.3 show the results for the 9 benchmarks with fixed



MATMULT

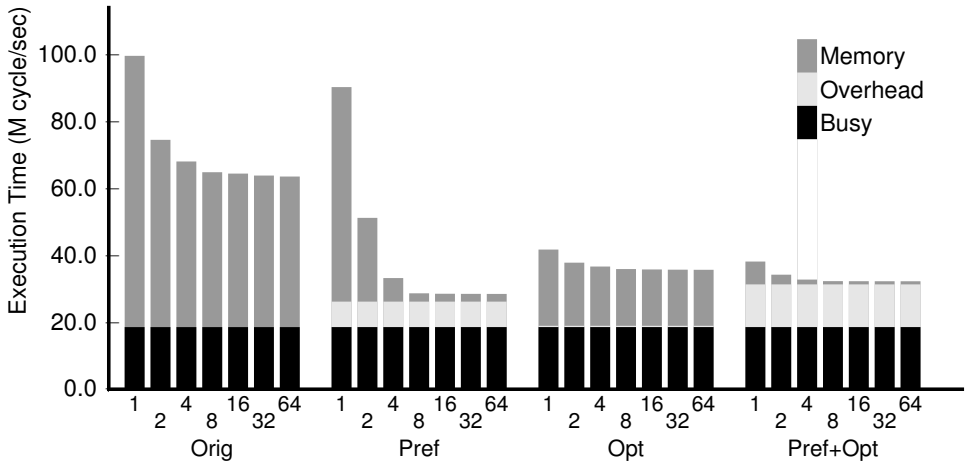


JACOBI

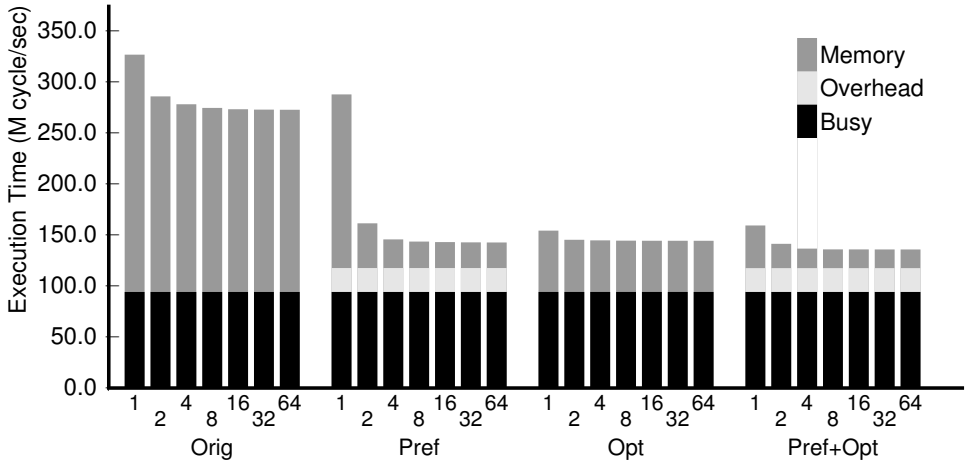


REDBLACK

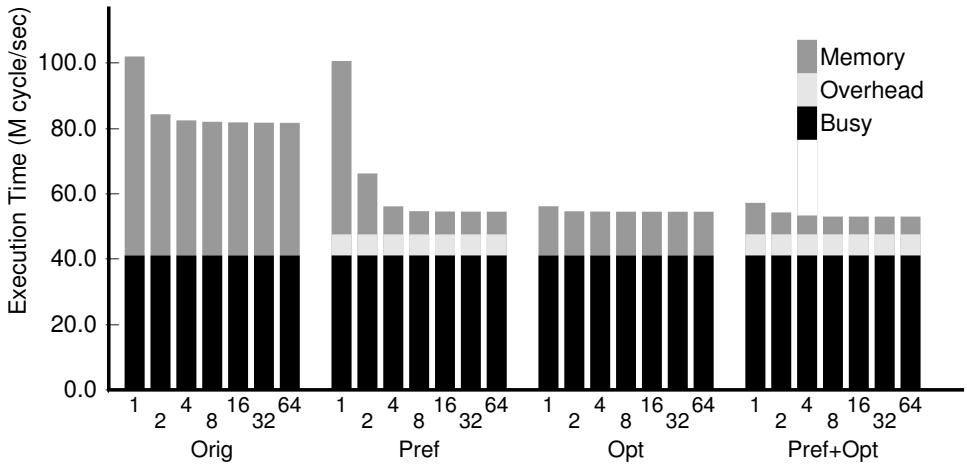
Figure 6.1: Affine Array applications execution time breakdown under memory bandwidth scaling with no optimizations (Orig), with software prefetching (Pref), with locality optimization (Opt), and with combined optimizations(Opt+Pref). Memory latency is fixed at 80 cycles.



IRREG



MOLDYN



NBF

Figure 6.2: Indexed Array 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.

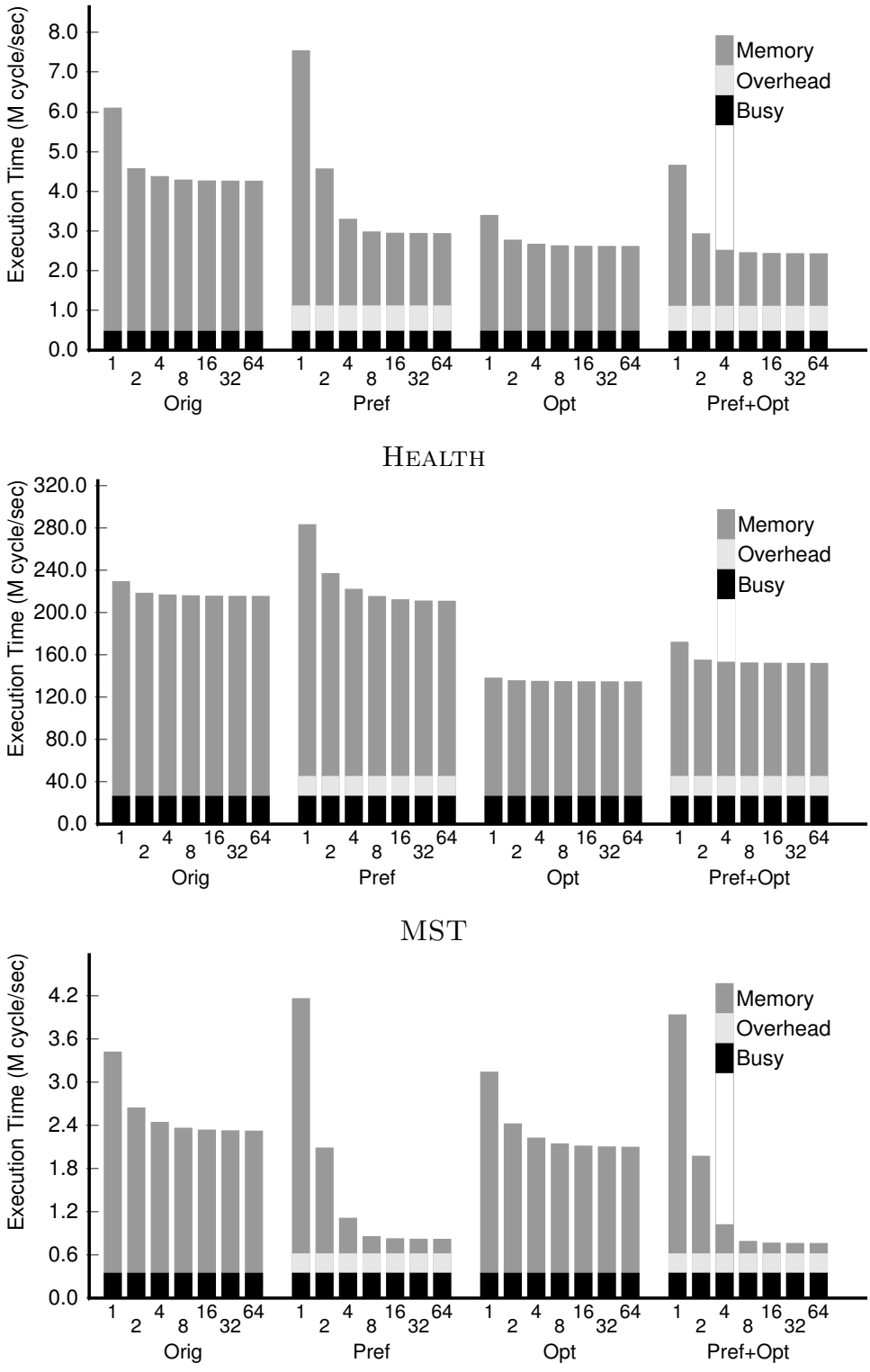


Figure 6.3: Pointer-chasing applications execution time breakdown under memory bandwidth scaling with no optimizations (Orig), with software prefetching (Pref), with locality optimization (Opt), and with combined optimizations(Opt+Pref). Memory latency is fixed at 80 cycles.

latency and varied bandwidth. In all these figures, execution time is plotted along the y-axis against the memory bandwidth which is varied from 1-64 Gbytes/sec along the x-axis, in powers of 2, keeping memory latency fixed at 80 cycles. Each execution time bar is broken down into memory stall, which is the amount of execution time spent waiting for some memory operation to complete, software overhead, which is the code inserted into the benchmarks to instrument them with the different optimization techniques, and Busy, which is the amount of time the code needs to execute. The Busy component is the amount of time to execute the code on a perfect memory system where every memory operation would take only one cycle. The overhead component is measured as the incremental difference between the original code and the optimized codes when executed on a perfect memory system. The stall component is the rest of the execution time after subtracting the busy and overhead components when the application is executed on a real memory system. Bars are grouped into 4 groups, including the original version which is labeled "Orig", the software prefetching version which is labeled "Pref", the locality optimized version which is labeled "Opt", and the combined version which is labeled "Pref+Opt".

First, let us discuss the results for affine arrays in Figure 6.1 and indexed arrays in Figure 6.2. Performance increases with increasing memory bandwidth for all versions of each benchmark. Both software prefetching and locality optimizations outperform the un-optimized original codes by 45% on average. The prefetching overhead is larger than that of locality optimizations. For affine arrays, tiling has overhead due the additional loop nests. For indexed arrays, RCB

Latency	MM	JAC	RB	Irreg	MOL	NBF	EM3D	Average
80	1.84	1.57	2.97	2.80	3.51	N/A	1.75	2.41
160	2.84	1.89	3.45	2.93	2.86	3.37	1.83	2.74
320	3.82	2.05	3.72	3.04	2.99	3.62	1.87	3.02
640	4.68	2.10	3.90	3.20	3.31	3.81	1.89	3.27

Table 6.5: Equi-performance bandwidths for 80, 160, 320, and 640-cycle memory latencies. The last column reports the average over the 9 benchmarks. All memory bandwidths are in Gbytes/sec.

has no measurable overhead since the runtime inspector is amortized over lots of computations. This is discussed in greater detail in the work done by Hwansoo Han and Chau-Wen Tseng in [22]. Prefetching overheads are mainly due to the prefetch instructions and address computations. For indexed arrays, unrolling doesn't reduce prefetch overhead as with the case of affine arrays as discussed earlier in Section 4.2.

The effectiveness of each of the two techniques is dependent on the "technology point" or the memory system parameters at which the comparison is performed. Prefetching overlaps latency with useful work. At high bandwidths, prefetching can hide practically all the memory latency, and outperforms locality optimizations. Locality optimizations reduce latency by more effectively using the cache; however, it does not get rid of all the latency. But at low latency, it outperforms prefetching since it reduces memory traffic. Consequently, the conclusion drawn from the data is that for all the array-based benchmarks (affine and indexed), software prefetching outperforms locality optimizations at high memory bandwidths, while locality optimizations outperform software prefetching at low memory bandwidths.

This thesis aims to show where each of the two techniques is outperforming the other. Only then, a conclusion about which technique is better for which applications on what memory system can be formalized. Table 6.5 reports the memory bandwidths at which software prefetching and locality optimizations achieve equal performance. For bandwidths higher than this equi-performance bandwidth, software prefetching outperforms locality optimizations, while for bandwidths lower than this equi-performance bandwidth, locality optimizations outperform software prefetching.

On a memory system with memory system latency of 80 cycles, the average *equi-performance bandwidth* for all array-based applications is 2.12 GBytes/sec. Note that NBF doesn't have an equi-performance bandwidth at 80 cycles since locality optimizations outperform software prefetching at all bandwidths for that particular latency. The implication of these results is that while software prefetching has superior maximum performance. "Latency hiding techniques" such as software prefetching cannot gain the optimal performance gains without using latency reduction techniques on current memory systems, which has bandwidths in the range of 1-3 GBytes/sec.

Second, let us look at Figure 6.3. This figure shows the results for the pointer-chasing benchmarks. In HEALTH, the prefetching overheads are much higher than cache-conscious allocation due to the jump pointer management and creation code. Prefetching doesn't get rid of all the memory latency because the loops are fairly short, around 120 elements as explained earlier. Prefetching tries to hide the memory latency underneath the useful work that is done in the

computation loops. Short lists do not provide prefetching enough work to hide the latency underneath. At low bandwidth, performance is worse than the original code because of the extra data that is being fetched since the data structures have more data due to the jump pointers. CCMALLOC or "cache-conscious allocation" has much smaller overheads. The overhead is hardly measurable. Overall, this allocation technique is much better than software prefetching at all bandwidths.

For MST, software prefetching isn't as effective in removing memory stalls even at high bandwidth since the lists are extremely short, around 4 nodes. This is even worse than HEALTH which has longer lists than MST. Software prefetching needs enough slack "computations" to hide the memory latency underneath which is not the case for MST. No crossover is detected for both HEALTH and MST; thus, locality optimizations outperform software prefetching at all bandwidths.

EM3D has different behavior compared to HEALTH and MST, and resembles the behavior of the array benchmarks. In EM3D, software prefetching achieves better performance than CCMALLOC at high bandwidths, and worse performance than CCMALLOC at low bandwidths; the equi-performance bandwidth is 1.7 Gbytes/sec at 80 cycles of latency. EM3D has an "array of lists" data structure which permits very effective prefetching of the lists. The data structure of EM3D was explained in more detailed in Section 6.1.

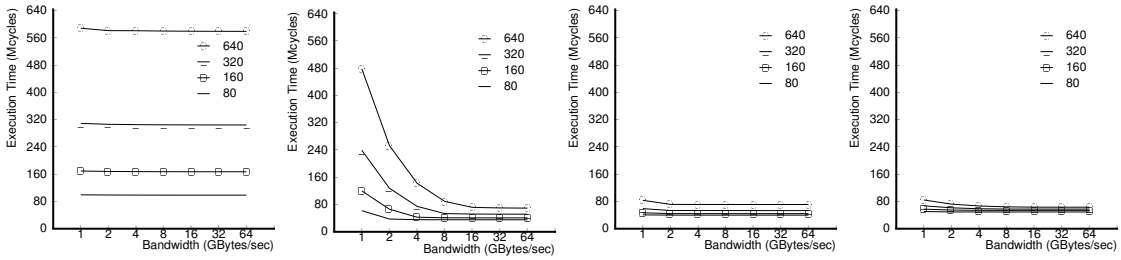
6.3 Varying Memory Latency

In the previous section, the results shown were for a fixed latency of 80 cycles with bandwidth scaling. In this section, memory latency is varied between 80 and 640 cycles along with bandwidth scaling in powers of 2.

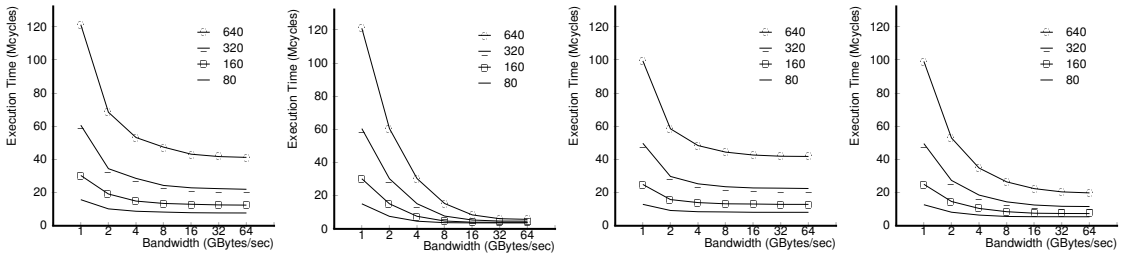
Figures 6.4, 6.5, and 6.6 show the results for the 9 benchmarks. The axes of the figures are exactly the same as in the fixed latency figures, but performance is not broken down into the three execution components. Instead, each figure plots four lines, representing bandwidth scaling results at 4 different memory latencies (80, 160, 320, and 640 cycles). In contrast to the graphs in Figures 6.1, 6.2, and 6.3 where each graph reports results for all versions of the benchmark, each version of each benchmark is shown in a separate graph.

As in Section 6.2, the focus of this section will be on comparing the techniques separately, leaving a discussion of the combined techniques for Section 6.4. Clearly, it can be concluded from all the figures of varying latency that execution times increase with increasing latency. Given the larger memory stall components at higher latencies, the performance differential between the techniques becomes magnified.

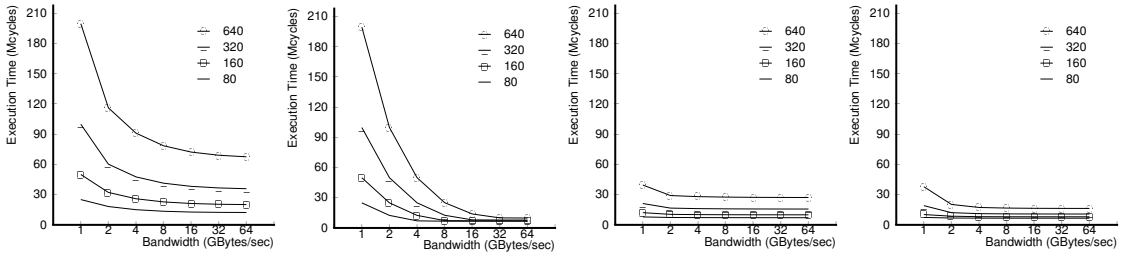
For both the affine array and indexed array benchmarks, software prefetching effectively hides the increasing memory latencies given sufficient memory bandwidth. This is clearly seen at the high bandwidth end of the graphs for those benchmarks where all the different latency lines come close together. Locality optimizations suffer performance degradation as memory latencies grow;



MM Orig MM Pref MM Opt MM Opt+Pref

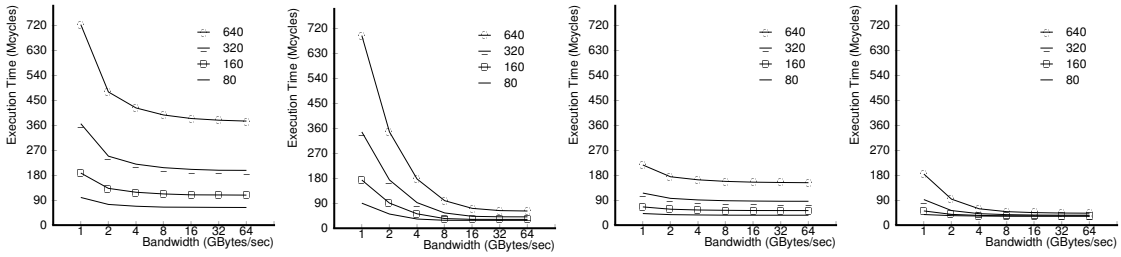


JACOBI Orig JACOBI Pref JACOBI Opt JACOBI Opt+Pref

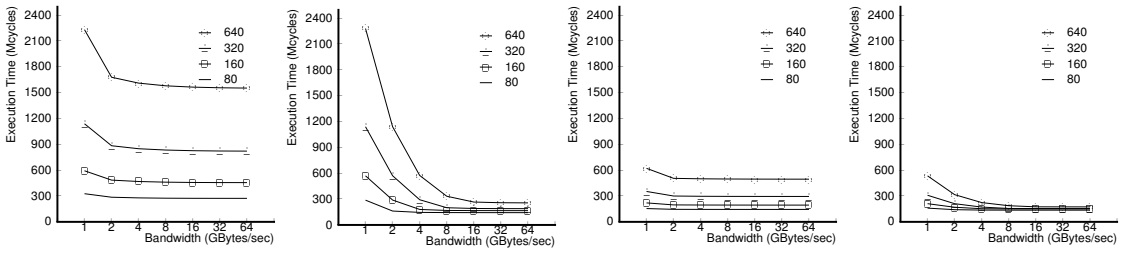


RB Orig RB Pref RB Opt RB Opt+Pref

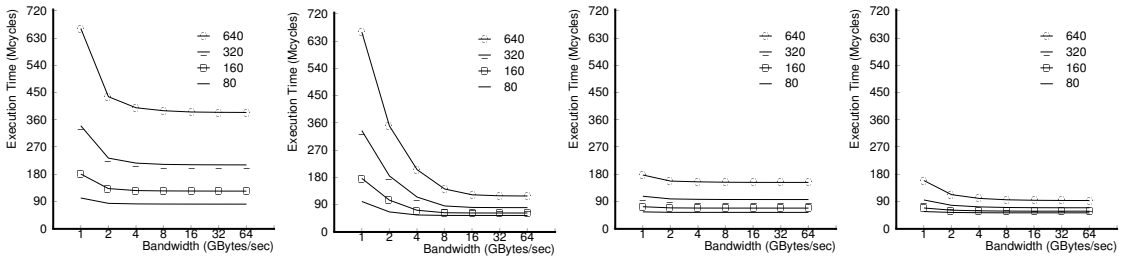
Figure 6.4: Execution time under both memory bandwidth and latency scaling for affine array benchmarks with no optimizations (Orig), with software prefetching (Pref), with locality optimization (Opt), and with combined optimizations (Opt+Pref).



IRREG Orig IRREG Pref IRREG Opt IRREG Opt+Pref

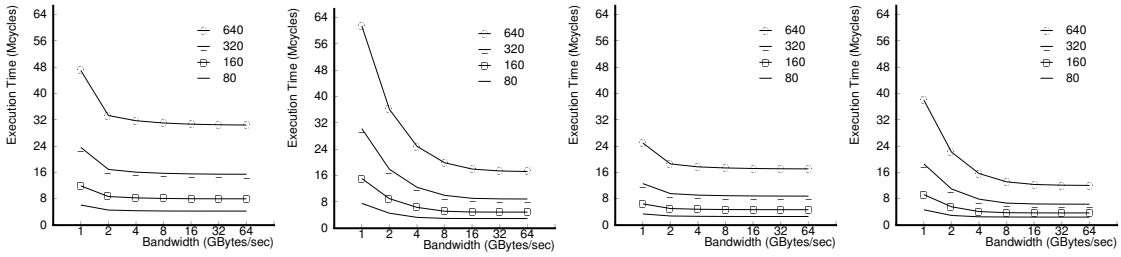


MOLDYN Orig MOLDYN Pref MOLDYN Opt MOLDYN Opt+Pref

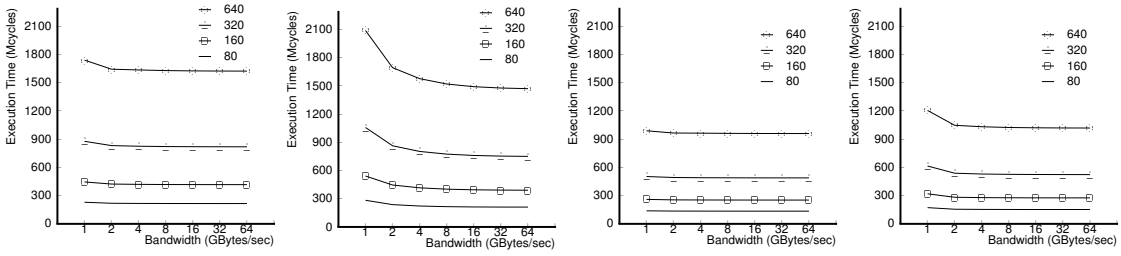


NBF Orig NBF Pref NBF Opt NBF Opt+Pref

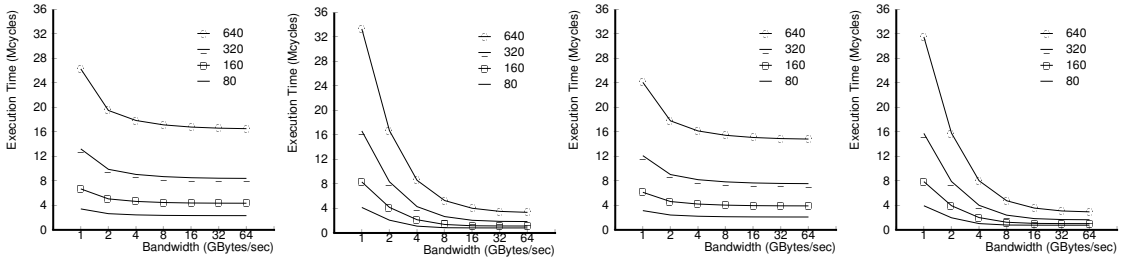
Figure 6.5: Execution time under both memory bandwidth and latency scaling for indexed array and pointer-chasing benchmarks with no optimizations (Orig), with software prefetching (Pref), with locality optimization (Opt), and with combined optimizations (Opt+Pref).



HEALTH Orig HEALTH Pref HEALTH Opt HEALTH Opt+Pref



MST Orig MST Pref MST Opt MST Opt+Pref



EM3D Orig EM3D Pref EM3D Opt EM3D Opt+Pref

Figure 6.6: 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).

however, at low memory bandwidths, the locality optimization graphs show better performance than prefetching even with high memory latency. Thus, the conclusion here is that software prefetching outperforms locality optimizations at high memory bandwidths, while locality optimizations outperform software prefetching at low memory bandwidths for all the memory latencies for all of the benchmarks. NBF at 80 cycles latency is the one exception since locality optimizations is slightly better than prefetching at all bandwidths.

Pointer-chasing benchmarks are different from the array-based benchmarks with varying latency as was the case with varying bandwidth while keeping latency fixed. For MST, locality optimization outperforms software prefetching at all memory latencies and bandwidths. For HEALTH, still software prefetching at high bandwidths outperforms locality optimizations. The same reasons given in Section 6.2 for the reduced effectiveness of software prefetching on pointer-based data structures explain locality optimization's performance advantage at higher memory latencies and lower bandwidths. Again, EM3D is an exception. EM3D performance with memory latency scaling is similar to affine and indexed array performance. The same reasons given in Section 6.2 apply. It is worthy to note that since the performance of the three pointer-chasing benchmarks differs, there is still room for research to study more pointer-chasing applications to better characterize their performance when software prefetching and locality optimizations are applied.

Table 6.5 shows the equi-performance bandwidths at different memory latencies. The crossover bandwidths grow with latency in general. Consequently,

Latency in Cycles	RedBlack	Jacobi	Matmult
80	9	8 , 40	16
160	9	11, 80	28
320	9	11, 156	52
640	9	11, 308	104

Table 6.6: Prefetch distances for the combined version of RedBlack, Jacobi and Matmult for the different latencies .

on future systems with high memory latencies, greater memory bandwidth will be required before software prefetching demonstrates a performance advantage over locality optimizations for all these benchmarks, *i.e.* latency reduction will still be important in the future and will provide more benefits than latency tolerance applied alone.

6.4 Combined Techniques

This section evaluates software prefetching and locality optimizations when they are combined naively, *i.e.* no tuning is applied to combine the techniques in the best possible way. The combined versions of all the benchmarks were generated in the following way.

First, for the affine array benchmarks, software prefetching is instrumented into the innermost tiled loops of each benchmark, *i.e.* the benchmark code is first tiled (locality optimized) and then software prefetching is applied to its inner most loops.

Table 6.6 shows the prefetch distances for the affine array benchmarks after applying software prefetching to the locality optimized innermost loops of the benchmarks to produce the naive combined version of the benchmarks at

Latency in Cycles	IRREG	MOLDYN	NBF
80	8 , 20 , 20 , 40	1, 1, 2	2
160	12, 40 , 40 , 80	2, 2, 3	4
320	24, 80 , 80 , 160	4, 4, 5	8
640	44, 160, 160, 319	7, 7, 9	16

Table 6.7: Prefetch distances for the combined version of IRREG, MOLDYN and NBF for the different latencies.

Latency in Cycles	HEALTH	MST	EM3D
80	8	3	2
160	16	3	3
320	32	3	6
640	16	3	11

Table 6.8: Prefetch distances for the combined versions of HEALTH, MST and EM3D for the different latencies.

different latencies. Note that for REDBLACK, the prefetch distance is fixed at 9 since the square tile size is 9x10 and the prefetch distances computed is larger than 9. For our algorithm to function properly, we must limit the prefetch distance to the smallest of the prefetch distance and the length of the tile in order to prevent prefetching beyond the current computation tile. This effect also occurs in the case of JACOBI for latencies higher than 80 cycles. Table 6.7 shows the prefetch distances for indexed array benchmarks after merging software prefetching and locality optimizations since they modify different parts of the benchmarks. Table 6.8 shows the prefetch distances used for the pointer-chasing benchmarks. The combined code is generated by merging the two optimizations as the case with indexed array benchmarks.

For the affine array benchmarks, tiling significantly reduces the number of iterations in the innermost loop because of the additional loop nests which tiling introduces. These loop modifications can be seen in Figure 5.2 Part(A). When

prefetching is applied to these short tiled loops, the software pipeline startup overhead incurred by prefetching, *i.e.* prologue loop prefetches becomes significant since they are executed more often than without tiling. In effect, this reduces the amount of memory latency that software prefetching would have hidden in the case that no tiling was applied. This effect is very clear in the high CPU overheads in the "Pref+Opt" versions of MATMULT and JACOBI in Figure 6.1. Combining inherits the merits and the demerits of the two techniques, namely, the overheads associated with both software prefetching and tiling. This reduces the combined code performance relative to software prefetching alone as shown in the previously mentioned figures.

For affine array benchmarks, the overheads roughly add. This is shown clearly in Figure 6.1. The maximum performance is often lower than the maximum performance of either technique alone. The combined technique, by virtue of the tiled loops providing short innermost loops giving prefetching less work to hide latency underneath, suffer more memory stalls. Also, the startup prefetches in the "prologue loop" will add more stalls since it will be executed once per innermost loop. Hence, this will be more visible at high memory latencies since the startup prefetches and memory stalls become more expensive.

For both indexed array and pointer-chasing benchmarks, software prefetching and locality optimizations modify different parts of the benchmark code. Software prefetching modifies the computation loops themselves while locality optimizations instrument the creation code semantics, or reorders data before entering the computation loop. Thus, the combined versions are generated

by merging modifications of software prefetching and locality optimization together. The results are shown in Figures 6.2, 6.3, 6.5 and 6.6 under "Pref+Opt" which is the rightmost graph in each of the benchmarks presented. The performance of the combined versions of the benchmarks for affine array benchmarks, indexed array benchmarks and EM3D is without question better than that of the original un-optimized version. This is not the case with MST and HEALTH for the reasons mentioned in Section 6.2. Yet, with these results especially for pointer-chasing applications there is still room for more investigation of other pointer-chasing applications to study how they perform and what differentiates them.

For indexed array codes, the combined overheads are similar to prefetching since the locality transformation codes have no measurable overhead as discussed earlier. The combined performance is superior at all bandwidths and latencies. The overhead is not higher than prefetching. There is no negative effects for the locality optimizations since data reordering doesn't change the loops at all; it only changes the order of the computations which doesn't affect software prefetching.

For pointer-chasing codes, except for EM3D, the combined techniques are worse than locality optimizations alone at low bandwidth since additional jump pointers consume precious memory bandwidth and increase overheads as well. For HEALTH, combined is the best at high bandwidth even though the overheads are high. For MST, the combined technique is worse than locality optimization alone because prefetching is completely ineffective for the extremely short lists.

Only overheads are added without hiding any memory latency. For EM3D, locality optimizations performs better than any other technique at low bandwidth. At higher bandwidth, combined is the best among all the techniques since it takes the advantages of latency tolerance from software prefetching and the better locality of CCMALLOC allocation.

For pointer-chasing benchmarks, combining always under-performs CCMALLOC memory allocation alone at low memory bandwidths. The jump pointers added for software prefetching and prefetch arrays required for pointer prefetching increase the demand for memory bandwidth, thus partially decreasing the reduced traffic benefits achieved by CCMALLOC memory allocation in the combined version. The combined version also under-performs CCMALLOC memory allocation at high memory bandwidths in MST. This was explained before in Section 6.2. 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. Thus, it is not such a good idea to apply combining for these applications. Only if the application has long lists compared to the prefetch distance computed for the computation loops, and there is enough memory bandwidth, then software prefetching with or without CCMALLOC will achieve performance gains.

The combined techniques inherit the overheads from both techniques, but enjoy the combined benefits of both traffic reduction and latency tolerance. From the performance figures, it is clear that at high bandwidth, the graphs follow the software prefetching performance. Although the combined techniques do not

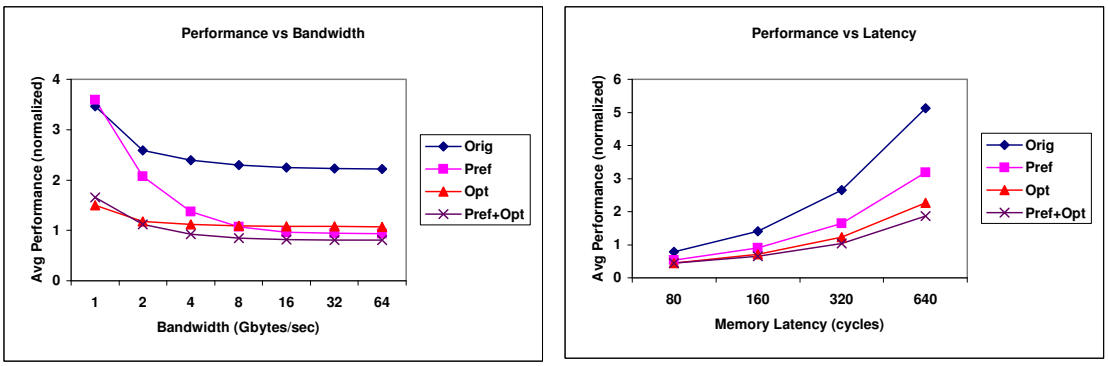


Figure 6.7: Comparing average performance for different versions of programs relative to memory bandwidth and latency. Performance is normalized relative to the original program with 1 Gbyte/sec bandwidth and 80 cycle latency.

outperform software prefetching alone, they are much better than locality optimization alone especially at high bandwidth. At low bandwidth, the combined techniques perform better than software prefetching alone because of the reduced traffic provided by locality optimizations.

The conclusion is that the combined techniques capture both the good and bad features of both techniques. On average, the combined codes perform better than any of the two techniques in isolation. Figure 6.7 shows this result. In this figure, the average performance of each version of the program is plotted relative to memory bandwidth and latency. Performance is normalized relative to the original program (with bandwidth of 1 Gbyte/sec and latency of 80 cycles; the lower-left point of the latency varyinggraphs in Figures 6.4, 6.5, and 6.6), then averaged over all programs for each memory bandwidth or latency. Two graphs are generated: one with varying bandwidth and one with varying latency.

The result in Figure 6.7 suggests that the combined techniques perform better than any of the two techniques alone on average and definitely better than

the original code over all bandwidths and latencies. In other words, the combined techniques are more robust to changes in memory system parameters. One observation that can be made from this data is that applying both techniques in concert achieves the best average application performance independent of memory parameters. This relieves the compiler from having to choose which technique to apply based on memory system parameters, which are usually not known to the compiler. This is particularly true if the compiler is generating code for different machines with different parameters.

Chapter 7

Algorithm Enhancements

Chapter 6 discussed the results for applying software prefetching and locality optimizations in isolation to the benchmark suite and the results of the naive combination in detail. This Chapter presents several enhancements to better combine the two techniques and to enhance software prefetching for array codes in the presence of conflict misses. First, tiling is enhanced to combine more effectively with software prefetching. Then, padding which is normally used to reduce conflicts in tiling is applied to software prefetching to avoid prefetch thrashing. Finally, CCMALLOC is used to reduce overhead in software prefetching for pointer-chasing data structures.

7.1 Enhancing Tiling for Software Prefetching

High startup overheads are noticed when tiling and software prefetching are combined naively as discussed in Section 6.4. Prefetching, when naively combined with tiling, loses part of its effectiveness due to the destructive interference introduced by the tiling algorithm which makes the tiled loops shorter than they were before applying tiling. The effect of these smaller innermost loops is to force

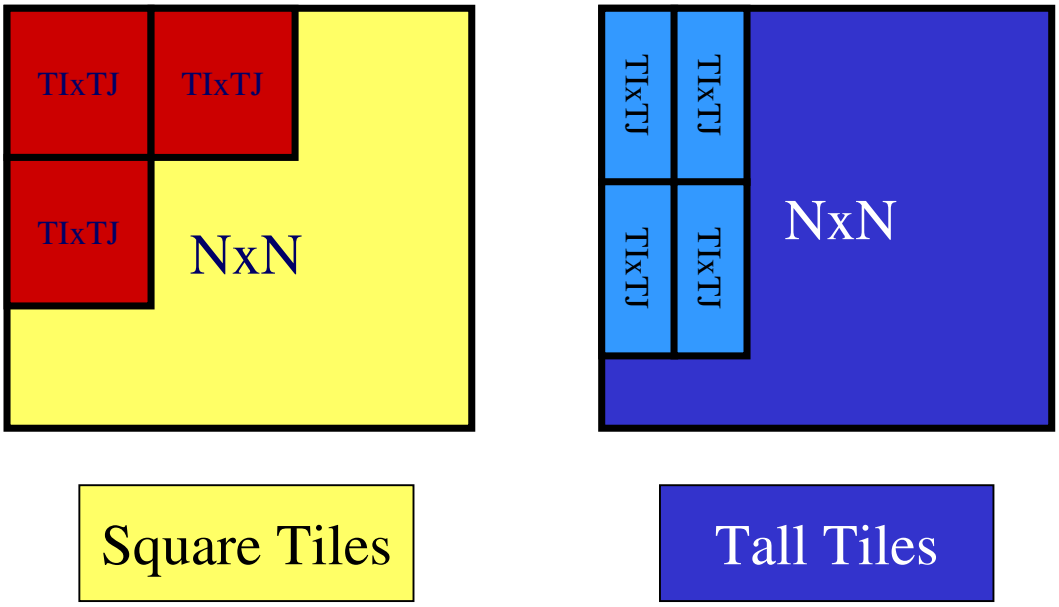


Figure 7.1: Two configurations one with square tiles and one with tall tiles.

more memory stalls to be exposed since there is not enough work to hide the latency underneath. Also, the overheads are increased since the prologue loop is getting executed more often due to the same reason mentioned above, the short tiled loops. Note that the number of prefetches executed is the same but how often the prologue loop gets executed increases. Using the pipeline analogy to explain prologue loops in Chapter 4, the pipeline length gets shorter so the same amount of work is performed on a shorter pipeline resulting in more frequent pipeline startup operations, and increased overhead.

The performance of software prefetching can be improved by biasing the tiling algorithm to select tiles that result in longer innermost loops, hence providing more work to hide latency underneath and reducing the overhead introduced by the prologue loop, *i.e.* countering the effects of short tiles.

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

Table 7.1: Tile sizes for square and tall-tile versions of the affine array benchmarks.

The tiling algorithm used to tile the affine array benchmarks is called the Euclidean GCD algorithm. It is explained in detail in [14, 44]. This algorithm generates a series of non-conflicting tile sizes that can be used to tile a particular program under a specific set of cache parameters.

Figure 7.1 shows two configurations of tile sizes: one is squarish and one is rectangular in shape. Tiles with a squarish aspect ratio typically achieve the best cache utilization. However, the algorithm can select taller tiles with greater height to width aspect ratio so that the problem of tiling and software prefetching can be alleviated. Such *tall tiles* will have more iterations in their innermost loops compared to square tiles. Thus, as mentioned before, two positive effects are achieved: one is reduction of startup overhead, and the other is more work in the innermost loops to hide the latency underneath. Table 7.1 shows the square and the tall tile sizes for the affine array benchmarks studied in this thesis.

Although selecting tall tiles increases software prefetching effectiveness, selecting tiles that are extremely tall has drawbacks. In the extreme case, having a tile size of $Y \times 1$ would result in mapping the problem back to the original computation order (recall in Figure 1.3 that the computation order was column by column, and that the size of each column was $N \times 1$). Thus, extremely tall tiles negates the benefits tiling introduces.

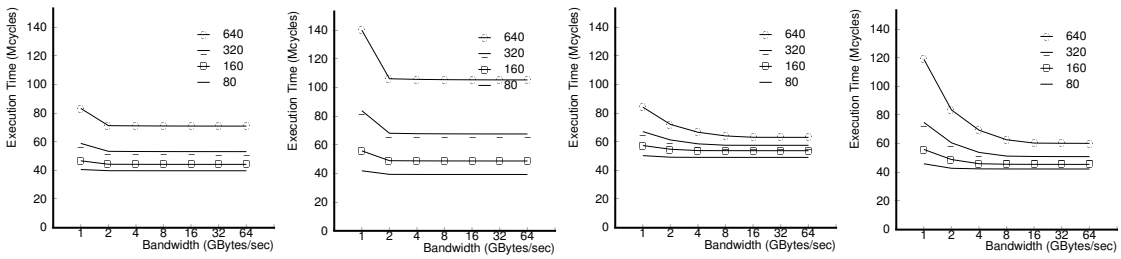
Latency in Cycles	RedBlack	Jacobi	Matmult
80	12	8 , 40	16
160	20	12, 80	28
320	31	24, 156	52
640	31	48, 308	104

Table 7.2: Prefetch distances for RedBlack, Jacobi and Matmult for the different latencies with tall tiles applied.

Table 7.2 shows the prefetch distances for the three affine array benchmarks. Each benchmark is instrumented with tall tiles and software prefetching. If the prefetch distance computed after the instrumentation with tall tiles exceeds the length of the tile, the prefetch distance is selected to be the minimum of the tile length and the computed prefetch distance. This ensures that software prefetching does not prefetch beyond the particular tile for which the computation is taking place.

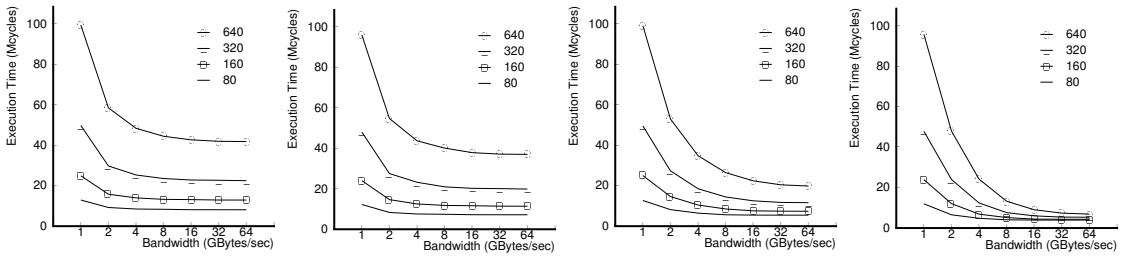
Figure 7.2 shows the results using tall tiles with and without prefetching for all the affine array benchmarks. Figure 7.2 compares square-tile versions of the benchmarks with tall-tile versions. Tall tiles and square tiles achieve similar performance when they are applied without software prefetching. However, when tall tiles are combined with software prefetching, the short-loop overheads suffered at high bandwidths are significantly reduced compared to square tiles. The performance with tall-tiles matches the performance of software prefetching alone from Figure 6.4. The conclusion of these performance results is that tall tiles fully exploit the benefits of software prefetching and tiling simultaneously and avoids the problems encountered due to the short innermost loops.

The combined tall-tile and software prefetching techniques retain the



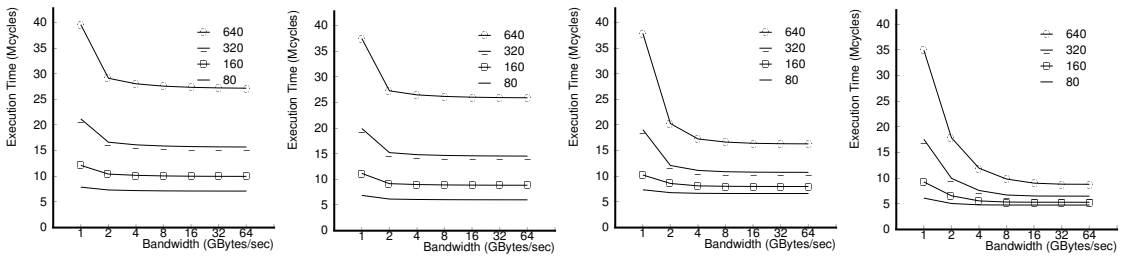
Square Tile | Tall Tile | Square Tile+Pref | Tall Tile+Pref

MATMULT



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

JACOBI



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

REDBLACK

Figure 7.2: Comparing square tiles and tall tiles with and without prefetching.

robustness benefit described in Section 6.4. Figure 7.2 shows that at low bandwidth, the performance tracks tiling performance alone, while at high bandwidth, the performance tracks software prefetching performance alone. Hence, the enhanced combined technique shows more robustness to variations in the memory system parameters since the two techniques have a better fit when the enhancement is applied.

7.2 Padding for Software Prefetching

Software prefetching works by hiding memory latency. Conflict misses on prefetched data, which can arise in pathological problem sizes that divide or nearly divide the cache size, generate conflict misses among prefetched data as illustrated in Figure 7.3. Such conflict misses essentially eliminate all the benefits of memory latency tolerance that software prefetching tries to achieve. Prefetched data gets knocked out from the cache before it is used due to the fact that some other data maps to the same location as the datum being prefetched.

Software prefetching for affine array codes with specific conditions such as power of 2 problem sizes and/or low L2 cache associativity requires applying array padding to alleviate the conflicts causing a mapping such as that suggested in the left half of Figure 7.3. This figure shows that with a particular problem size, some data gets mapped to the same cache line(s). Thus, conflicts arise. Array padding can avoid such conflicts [43, 44], even if the loops are tiled. Array padding helps eliminate further conflicts that tiling would not address. The compiler can perform array padding by inserting dummy variables or by

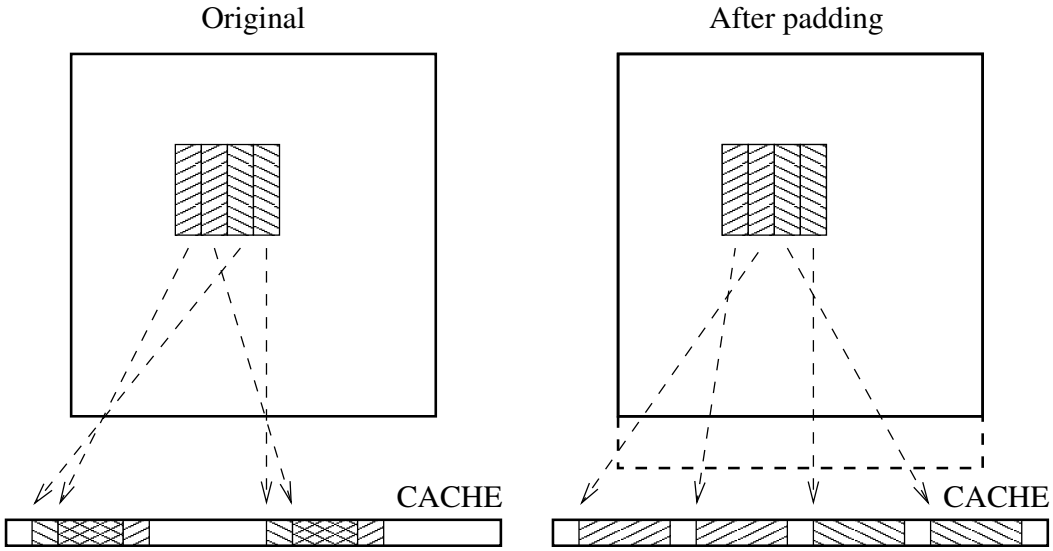


Figure 7.3: Layout of Data in the Cache before and after padding.

Latency in Cycles	RedBlack	Jacobi
80	12	8 , 36
160	24	16, 68
320	48	28, 136
640	96	56, 268

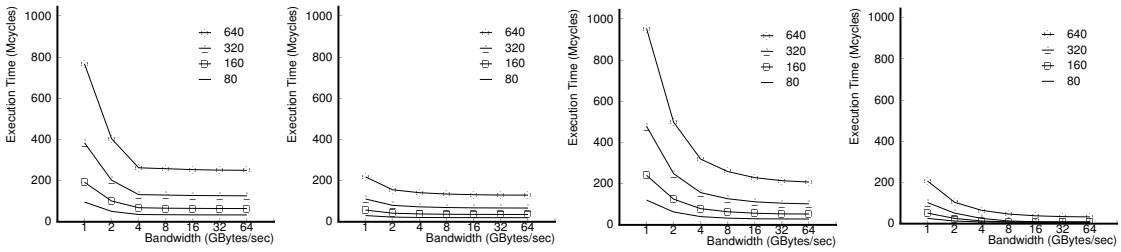
Table 7.3: The prefetch distances for the padded versions of RedBlack and Jacobi.

increasing the size of the leading array dimension.

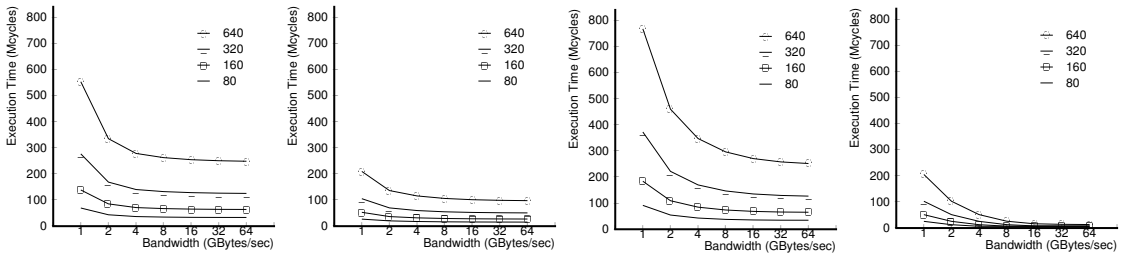
The appropriate amount of padding to apply to the array is computed as follows. First, the prefetch distance, computed by the prefetching algorithm as explained in Chapter 4, is treated as the "height" of a tile that should be kept in the cache for the whole computation, such that no conflicts occur to evict array elements within this tile. Before the first step the prefetch distance for the benchmark is computed as described in previous chapters. Table 7.3 reports the prefetch distances for Jacobi and RedBlack. Second, the compiler uses the

Euclidean GCD algorithm which was used in Section 7.1 to compute the tall tile size and to determine whether cache conflicts will occur within such a prefetch distance (tile height). Padding is introduced incrementally to the leading array dimension until the Euclidean GCD algorithm gives a conflict free tile size whose height is at least equal to or larger than the computed prefetch distance for the benchmark used [44, 45]. In other words, the algorithm computes a new padded problem size such that if tiling were applied to this particular problem size, conflict free tiles whose height is at least equal to the prefetch distance are produced by the Euclidean GCD algorithm. This ensures that prefetched data will not experience conflicts and will stay in the cache until the processor accesses them.

Versions of JACOBI and REDBLACK were created with and without both padding and prefetching. These codes were run on the same simulator configuration used before, but with a 2-way set associative L2 cache. The conflicts introduced in these benchmarks can be eliminated by using a 4-way set associative L2 cache for both benchmarks. However, this is not true for more complex benchmarks. Thus, the combination of these benchmarks with a 2-way set associative L2 cache permits a reasonable study of our padding for software prefetching technique to show significant conflict cache misses and enabling the study of improving software prefetching with padding. The problem size used for JACOBI is 256×256×8, while the problem size of REDBLACK is 256×256. Note that, power-of-two problem sizes occur frequently in multigrid codes. Based on the prefetch distance for each of the two benchmarks, applying the algorithm



JACOBI



REDBLACK

Figure 7.4: Padding for prefetching in Jacobi and RedBlack.

gives a padded problem size for JACOBI to be 313×256×8, and 313×256 for REDBLACK. The padding is done for the leading array dimension which has the effect of shifting the locations of all other columns of the arrays in the cache, as depicted in the right half for Figure 7.3.

Figure 7.4 shows the results for the two benchmarks. Figure 7.4 shows both JACOBI and REDBLACK experience many cache misses due to conflicts. This was verified by running the same original un-optimized code for the two applications with fully associative caches which showed that all the memory stalls went away and performance similar to that shown for "Orig+Pad" was observed. After padding the array according to the algorithm explained above, the experiments were run for four different versions of each benchmark: ("Orig" which represents the results for the un-optimized original code, "Orig+Pad" which represents the results for the original array with array padding, "Pref" which represents the results for the prefetching version of the code, and "Pref+Pad" which represents the results for the prefetched code optimized with array padding).

Looking at the first two graphs for REDBLACK and JACOBI, padding alone added to the original un-optimized code is capable of removing most of the conflict misses. Also, it is clear that prefetching when applied alone to both benchmarks provides zero benefit due to the conflicts. In fact, performance degrades at low bandwidth due to fetching more data because of the conflicts. Finally, let us look at the results after applying array padding to software prefetching. Array padding minimizes the cache conflicts seen in the original

code, allowing software prefetching to achieve better performance. It is clear that "Pref+Pad" outperforms every other version of the benchmarks.

7.3 Index Prefetching

Software prefetching for pointer-chasing codes as shown in Chapter 6 suffers large overheads due to the creation and management of the jump pointers. If CCMALLOC is used to allocate the list nodes, then there is no need to have jump pointers. Since CCMALLOC allocates list nodes in a linear fashion, the address of any link node can be computed simply by offsetting from the address of the list head instead of traversing all the list nodes sequentially. With CCMALLOC used in this manner, the pointer-chasing problem is alleviated and all accesses behave like affine array accesses.

Figure 5.5 shows how CCMALLOC would allocate the list nodes in memory. All the list nodes are contiguous in main memory, so referencing them will require just knowing the location of the first node and then indexing. Using the address of the first node and the node size, any node down the list can be reached without performing memory indirection.

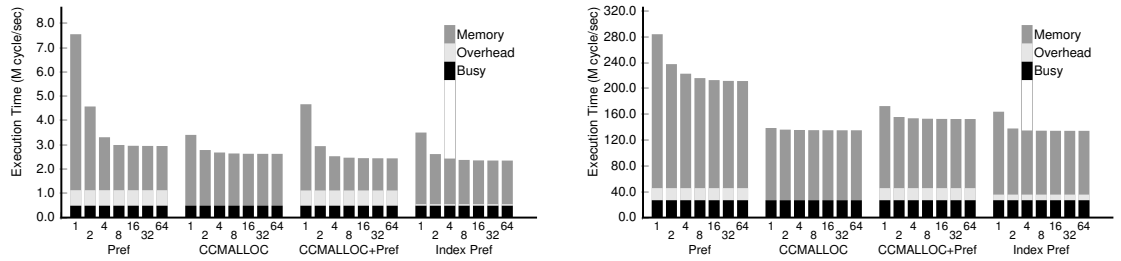
This approach was called *index prefetching* in [2]. The accesses use indexing as if they were accessing a static array. This technique was originally proposed by Luk and Mowry in [32]. They called it *data-linearization prefetching*. With index prefetching, the jump pointers become unnecessary, and removing them also removes all the overheads associated with jump pointer creation, management and maintenance. This benefit is applied only to the two

pointer-chasing benchmarks which exhibit bad performance with jump pointer prefetching: HEALTH and MST.

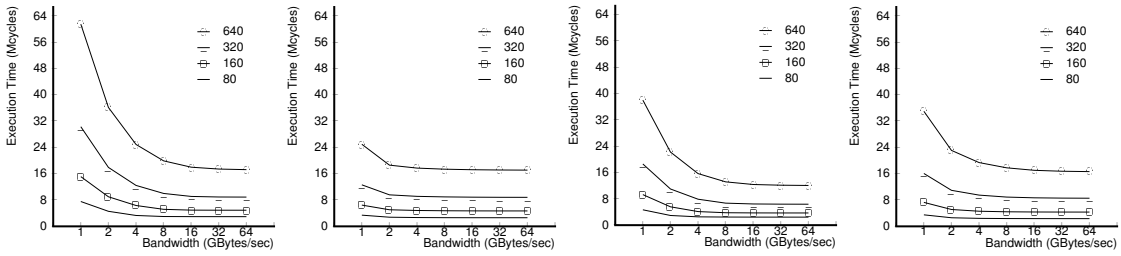
Index prefetching versions for HEALTH and MST are created and simulation results are shown in Figure 7.5 with both fixed latency and variable latency, as is done in Chapter 6. This figure shows the results for prefetch arrays (jump pointers), CCMALLOC memory allocation, combined optimizations, *i.e.* CCMALLOC and prefetching, and finally, index prefetching. The top two graphs show results with memory latency fixed at 80 cycles, and the remaining graphs show results with both latency and bandwidth scaling.

The figure shows that index prefetching eliminates most of the overheads incurred by jump pointer prefetching. Thus, index prefetching outperforms almost all versions at high memory bandwidths for both HEALTH and MST. Index prefetching is very close in performance to CCMALLOC alone. This is most obvious in MST. The reason for this is that the length of the list is not known and some unnecessary prefetches past the list length can be issued, causing both overhead and waste of memory bandwidth.

Index prefetching reduces software overheads but does not eliminate as much memory stalls as prefetch arrays, particularly in HEALTH. In HEALTH, many delete and insert operations occur. Even though CCMALLOC allocates lists linearly, frequent inserts and deletes randomize the layout of the list nodes. Hence, index prefetching loses its effectiveness because it tries to prefetch physically contiguous nodes which are no longer logically contiguous due to the deletes and inserts. Although index prefetching has lower overhead, prefetch

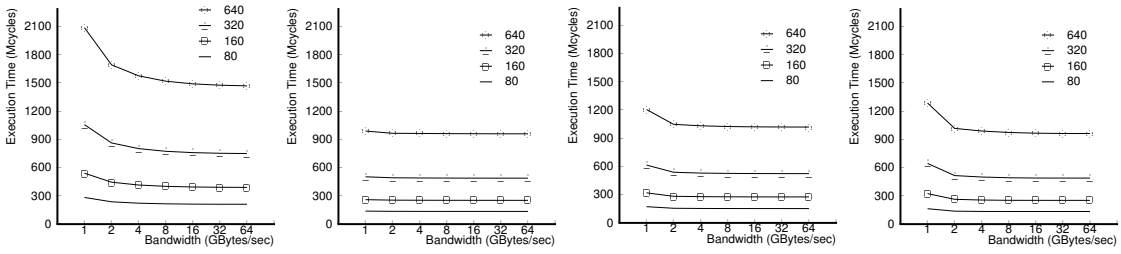


HEALTH MST



Pref CCMALLOC CCMALLOC+Pref Index Pref

HEALTH



Pref CCMALLOC CCMALLOC+Pref Index Pref

MST

Figure 7.5: 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.

arrays hide more memory latency than index prefetching because of the delete-insert problem. The increased memory stalls caused by the increased memory latency outweigh the benefits of reduced software overheads, making the naive combination of CCMALLOC and prefetch arrays the superior technique.

For MST, index prefetching performs very similar to CCMALLOC that it outperforms it with a very small margin at high memory latencies only since some of the latency is partially hidden. CCMALLOC does not have that advantage. CCMALLOC, in general, outperforms all other techniques when the lists are short since short lists prevent prefetching from being effective.

Chapter 8

Conclusion

Several conclusions can be drawn from this work. The main conclusions are as follows. First, the relative effectiveness of software prefetching and locality optimizations depends on how much memory bandwidth is available. In array-based benchmarks, software prefetching outperforms locality optimizations at high memory bandwidths, while locality optimizations outperform software prefetching at low memory bandwidths without exception. The equi-performance "crossover" bandwidth is around 2.4 GBytes/sec on today's memory systems, but the simulations show that this is going to increase as memory latencies increase in the future. However, for some types of pointer chasing applications, 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. For EM3D, the comparison between software prefetching and locality optimizations resembles the affine array benchmarks.

Second, combining software prefetching and locality optimizations inherits the merits of both techniques. Combining provides better performance than

either software prefetching or locality optimizations alone when memory latency is very high. Also, combining is less sensitive to changes in the memory system parameters than either software prefetching or locality optimization techniques applied in isolation. Moreover, naively combining techniques does not outperform the best performance of software prefetching and locality optimizations when applied alone at all bandwidths and latencies.

Finally, the combination of software prefetching and locality optimizations can be enhanced through better combining algorithms. First, for affine array benchmarks, selecting tall-tiles will reduce prefetching startup overheads, allowing combining to outperform software prefetching and locality optimizations applied in isolation especially at the points where one of the two techniques would win over the combined case. Second, padding is capable of removing conflicts on prefetched data in affine array benchmarks, and is essential for problem sizes that are close to multiples of the cache size. Third, for pointer-chasing benchmarks, using of index prefetching instead of naively combining prefetch arrays and CCMALLOC memory allocation can reduce prefetch overheads. However, index prefetching is not a panacea. Index prefetching does not perform well when large numbers of list nodes cannot be allocated contiguously due to fragment deletion and insertions as is the case in HEALTH, or when CCMALLOC allocation strategy is already doing well by itself thus reducing the need for further enhancements as is the case in MST. The only exception in such applications is EM3D which performs similar to affine array benchmarks using jump pointers prefetching.

Memory bandwidths of 1-4 Gbytes/sec are achievable in current commercial systems. The simulation results that match current memory and processor systems are towards the low end of our graphs. With faster and faster processors, the memory wall will grow which will impose higher requirements on memory bandwidth relative to current systems. Locality optimizations will be essential in these future systems since the available memory bandwidth will be more scarce. As the latency of memory systems increase, the relevant results from our experiments will be the upper curves in the variable latency figures.

In the future, processor-in-memory (PIM) architectures [3] if realized will increase memory bandwidth dramatically. For data residing on the PIM chip, the available memory bandwidth will resemble the higher end of our bandwidth ranges. Thus, this will enable software prefetching to provide significant performance gains and improve the performance of applications running on such processors. Locality optimizations for such systems will still be useful to reduce the time the processor spends going off-chip to bring data into the PIM chip.

Chapter 9

Future Work

The results in this thesis have been acquired using small kernels from larger applications. These kernels are important to optimize since they comprise a significant portion of the processing time in the larger problems. Thus, the question that needs to be answered is what is the effect of these techniques when applied to real applications?

An important area of future work is to perform our study using more realistic applications. Our locality optimizations were applied semi-automatically, only partially implemented in the compiler and run-time system. For these optimizations to be widely used, compilers must automatically incorporate them as much as possible. In addition, software prefetching was instrumented by hand in all of the loops for all the benchmarks. This is yet another task that should be performed automatically by compiler. Similarly, the enhancements we propose are also performed by hand. Another important direction for future work is to implement our algorithms in a compiler and to perform a similar study to see if our results hold.

Our goal of reducing the memory wall is essential for improving

performance on scientific, engineering, and commercial workloads. This thesis presents some insights that are likely to prove useful for improving the memory performance of these workloads on future memory system designs.

Bibliography

[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] Abdel-Hameed A. Badawy, Aneesh Agarwala, Donald Yeung, and Chau-Wen Tseng. Evaluating the Impact of memory system performance on software prefetching and locality optimizations. In *Proceedings of the 15th Annual International Conference on Supercomputing*, Sorrento, Italy, June 2001. ACM.

[3] Gordon Bell, Richard Sites, William Dally, David Ditzel, and Yale Patt. Architects Look to Processors of Future. *MICROPROCESSOR REPORT, MICRODESIGN RESOURCES*, VOL.10(NO.10), August 5, 1996.

[4] T. Bonk and U. Rüde. Performance analysis and optimization of numerically intensive programs. SFB Bericht 342/26/92 A, Institut für Informatik, TU München, November 1992.

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

[6] 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.

[7] David Callahan, Ken Kennedy, and Allan Porterfield. Software Prefetching. In *In Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, April 1991.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, San Diego, CA, May 1993. IEEE.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] John L. Hennessy and David A. Patterson. Computer Architecture A Quantitative Approach. *Morgan Kaufmann*, Second Edition:Chapter 1, 1996.

[24] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, Doug Carmean, Alan Kyker, and Patrice Roussel. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*, Q1, 2001.

[25] 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.

[26] 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 International Symposium on Computer Architecture*, pages 364–373, Seattle, WA, May 1990. ACM.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] Bert McComas and Van Smith. The War Escalates Athlon4 takes on Pentium4. *InQuest Market Research*, May 2001.

[34] 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.

[35] 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.

[36] 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.

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

[38] 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.

[39] Todd Mowry, Monica Lam, and Anoop 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.

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

[41] 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 24–33, Chicago, IL, May 1994. ACM.

[42] 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.

[43] 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.

[44] 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.

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

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] James R. Larus Satish Chandra and Anne Rogers. Where is time spent in messagepassing and sharedmemory programs. pages 61–73. ASPLOS VI, November San Jose, California, 1994.

[51] 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.

[52] L. Stals, U. Rüde, C. Weiss, and H. Hellwagner. Data local iterative methods for the efficient solution of partial differential equations. In *Proceedings of the Eighth Biennial Computational Techniques and Applications Conference*, Adelaide, Australia, September 1997.

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

[54] W. F. van Gunsteren and H. J. C. Berendsen. GROMOS: GROningen MOlecular Simulation software. Technical report, Laboratory of Physical Chemistry, University of Groningen, Nijenborgh,The Netherlands, 1988.

[55] C. Weiß, W. Karl, M. Kowarschik, and U. Rüde. Memory characteristics of iterative methods. In *Proceedings of SC'99*, Portland, OR, November 1999.

[56] 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.

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