EFFECTIVE COMPILE-TIME ANALYSIS FOR DATA PREFETCHING IN JAVA

A Dissertation Presented

by

BRENDON D. CAHOON

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 2002

Department of Computer Science

EFFECTIVE COMPILE-TIME ANALYSIS FOR DATA PREFETCHING IN JAVA

A Dissertation Presented

by

BRENDON D. CAHOON

Approved as to style and content by:

Kathryn S. McKinley, Chair

J. Eliot B. Moss, Member

Charles C. Weems, Member

Russell G. Tessier, Member

W. Bruce Croft, Department Chair
Department of Computer Science

To my family, especially my wife Laura

ACKNOWLEDGMENTS

I am deeply indebted to my advisor, Kathryn McKinley, for providing an enjoyable atmosphere for doing challenging research. I took her first class at UMass, and she has been stuck with me ever since. She has been an outstanding advisor and I am fortunate to have worked with her. I hope I am able to pass on her principles.

I thank Eliot Moss and Chip Weems for their leadership of our research group. They are very knowledgeable, and my discussions with them have improved this work significantly. They also served as members of my committee, and I appreciate their helpful comments and suggestions. Thanks also to Russ Tessier, my external committee member, for his careful reading of my dissertation and providing valuable feedback.

A special thanks goes to Fred Green and Arthur Chou, two excellent teachers that introduced me to research and encouraged me to go to graduate school.

I wish to thank all the members of the ALI research group over the years, especially Steve Blackburn, Jim Burrill, Steve Dropsho, Sharad Singhai, and Zhenlin Wang. Thanks also to many other friends in Amherst who made my time there enjoyable including Alan B., Matt K., Alan K., Ron P., Dan R., Matt S., and John W.

I have spent the last three years at the University of Texas. I would like to say thanks to J.C. Browne and Calvin Lin for hosting me while at UT. I have been fortunate to make many friends while at UT, including Emery Berger, Rich Cardone, Sam Guyer, Xianglong Huang, Daniel Jimenez, Ram Mettu, and Phoebe Weidmann. I could always count on them for stimulating discussions, some of which were actually about research.

I am very grateful to my family, especially my parents, for their support over the years. They are the ones that have made this work possible. Finally, I want to thank Laura for her support, patience, and love.

ABSTRACT

EFFECTIVE COMPILE-TIME ANALYSIS FOR DATA PREFETCHING IN JAVA

SEPTEMBER 2002

BRENDON D. CAHOON

B.A., CLARK UNIVERSITY

M.S., UNIVERSITY OF MASSACHUSETTS, AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Kathryn S. McKinley

The memory hierarchy in modern architectures continues to be a major performance bottleneck. Many existing techniques for improving memory performance focus on Fortran and C programs, but memory latency is also a barrier to achieving high performance in object-oriented languages. Existing software techniques are inadequate for exposing optimization opportunities in object-oriented programs. One key problem is the use of high-level programming abstractions which make analysis difficult. Another challenge is that programmers use a variety of data structures, including arrays and linked structures, so optimizations must work on a broad range of programs. We develop a new unified data-flow analysis for identifying accesses to arrays and linked structures called recurrence analysis. Prior approaches that identify these access patterns are ad hoc, or treat arrays and linked structures independently. The data-flow analysis is intra- and inter-procedural, which is important in Java programs that use encapsulation to hide implementation details.

We show Java programs that traverse arrays and linked structure have poor memory performance. We use compiler-inserted data prefetching to improve memory performance in these types of programs. The goal of prefetching is to hide latency by bringing data into the cache prior to a program's use of the data. We use our recurrence analysis to identify prefetching opportunities in Java programs. We develop a new algorithm for prefetching arrays, and we evaluate several methods for prefetching objects in linked structures. Since garbage collection is an integral part of Java, we evaluate the impact of a copying garbage collector on prefetching. We demonstrate how to improve the memory performance of the collector itself by using prefetching. This dissertation shows that a unified whole-program compiler analysis is effective in discovering prefetching opportunities in Java programs that traverse arrays and linked structures. Compiler-inserted data prefetching improves the memory performance even in the presence of object-oriented features that complicate analysis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

We develop an effective compile-time analysis that discovers and exploits data prefetch opportunities to improve memory performance in Java programs that use arrays and linked structures.

Increases in modern processor speed continue to outpace advances in memory speed resulting in an underutilization of hardware resources due to memory bottlenecks. Schemes for reducing or tolerating memory latency are necessary to achieve high performance in modern computer systems. Most commercial architectures use multiple level cache hierarchies to alleviate memory bottlenecks. Caches improve memory performance by allowing quick access to a small amount of frequently used data. Caches work because programs exhibit locality of reference. Unfortunately, increases in data set sizes and the increasing disparity between the processor and memory speeds limit the performance of many workloads.

Researchers have performed a significant amount of work investigating techniques, including compiler support, to improve the *effectiveness* of caches. These include data and code transformations to improve the data reuse in the cache. Compile-time program transformations require complex static analysis in order to determine when transformations are profitable and legal. Compilers must be conservative when making optimization decisions and will not transform a program if sufficient information is either unavailable or not computable.

Another technique for improving memory performance is data prefetching which attempts to *tolerate* cache latency. The goal of prefetching is to bring data into the cache

prior to the use of that data. The key to effective prefetching is to determine *what* to prefetch and *when* to issue a prefetch. Determining prefetching opportunities may be done by hardware or software techniques. Hardware prefetching detects run-time access patterns using additional hardware resources to determine appropriate data to prefetch. Software methods require compiler support to generate additional instructions for prefetching data. Software data prefetching has several advantages over hardware-only schemes. The hardware complexity of implementing a prefetch instruction is much less than the complexity of a complete hardware prefetching implementation. A software prefetch instruction is more flexible. Programs can choose exactly when to issue prefetches and what to prefetch. For example, a hardware mechanism may prefetch floating point data only, whereas software methods can prefetch data of any type. Compilers provide a method to generate software prefetches automatically. Hardware methods typically ignore program structure, which results in an increase in memory traffic by issuing superfluous prefetches.

Most existing techniques for improving memory performance using prefetching focus on Fortran and C programs. Memory latency is a barrier to achieving high performance in Java programs as well. The software engineering benefits of object-oriented languages encourage programmers to use Java to implement a wide variety of programs, including those that require high performance. Traditional techniques for improving memory performance are difficult to apply to object-oriented languages. Software engineering practices make compile-time analysis of object-oriented programs difficult. For example, object-oriented programming encourages the use of encapsulation and small methods, both of which complicate compile-time analysis. Overcoming the challenges that software engineering practices introduce requires whole program analysis.

Programmers frequently use arrays and linked structures in Java programs. For example, the underlying data structure for several Java core library classes, such as `java.util.Stack`, is an array. New versions of Java include library support for container classes that use linked structures, such as `java.util.LinkedList`. We believe that it

is important for compilers to analyze and optimize both arrays and linked structures. Prior techniques focus on optimizing one or the other, but not both.

To improve the memory performance of object-oriented programs, we investigate software controlled data prefetching to improve memory performance by tolerating cache miss latency. We develop and implement a new data-flow analysis to identify traversal patterns in arrays and linked structures. The analysis is unique because it presents a single framework for identifying prefetching opportunities in array-based and pointer-based codes. We believe the data-flow analysis will be useful in other domains, such as data layout optimization. We describe and evaluate a compiler implementation of a new compiler technique for prefetching arrays, and three methods for prefetching linked structures in Java. Our unified framework is able to identify array and linked structure traversals that occur across method boundaries. We demonstrate that our new array prefetching technique is able to improve memory performance significantly. Prefetching linked structures in Java programs is effective also, but there is still room for improvement.

Java uses garbage collection to manage dynamic memory allocation automatically. Since Java requires garbage collection, we examine the impact of garbage collection on prefetching. We investigate generational copying garbage collectors specifically. Since a copying collector reorganizes data, there is synergy between prefetching and the collector. We investigate the potential for using the collector to improve prefetching. We also show that our copying collector has poor memory performance, so we evaluate the effectiveness of adding prefetch instructions to improve the performance of the collector itself.

We organize the rest of this section as follows. Section 1.1 introduces array prefetching. Section 1.2 introduces linked-structure prefetching. We summarize our unified data-flow analysis that detects prefetch opportunities for both arrays and linked structures. Section 1.3 describes the organization of the dissertation. Finally, we summarize our contributions in Section 1.4.

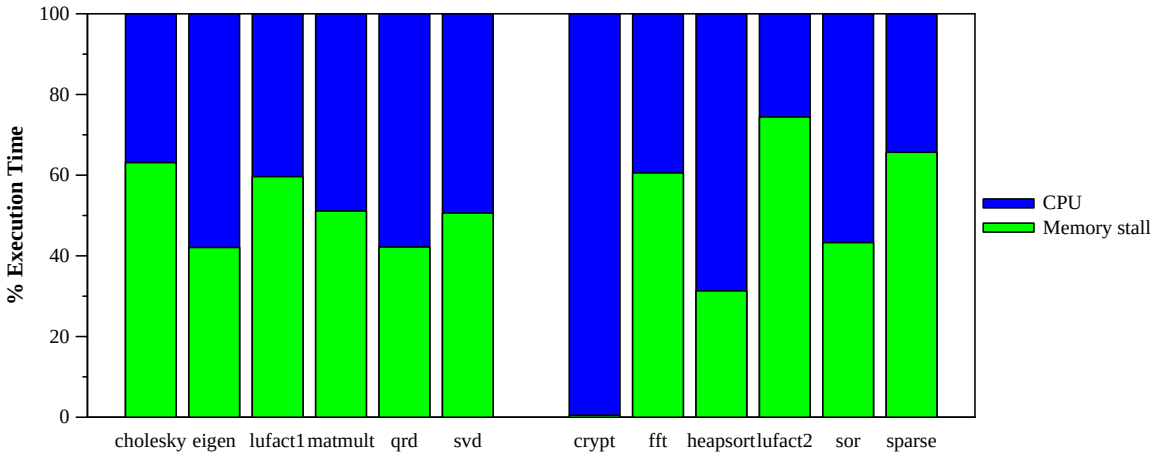


Figure 1.1. Memory Penalty in Array-based Java Programs

1.1 Prefetching Arrays

Programmers are using Java increasingly to solve programming problems that require high performance, including those involving matrix computations. Poor memory efficiency limits the performance of Java programs just as it does for C and Fortran. Over half the programs in Figure 1.1 spend more than 50% of time waiting for memory on a simulated out-of-order superscalar processor. We obtain these measurements by compiling the programs using Vortex [34], an ahead-of-time compiler, and running them on RSIM [84], an execution driven simulator. Figure 1.1 illustrates that there is significant room for improvement in these Java programs.

Traditional approaches for improving memory performance in array-based applications use loop transformations, such as tiling and unrolling [64, 73]. Implementing loop transformations in Java compilers is challenging due to the semantics of Java arrays and exceptions [9]. Java multidimensional arrays present challenges because the language specifies them as arrays-of-arrays. As a result, it is not possible to compute the address of any element directly. In a true multidimensional array, such as in Fortran, it is possible to compute the address of any element relative to the start of the array. The Java language specification requires precise exceptions, which means that all statements appearing before an exception must complete, and that the result of any statement appearing after an exception cannot

appear to have completed [43]. Optimizations must be careful not to violate this property so compilers often do not transform code that occurs in exception handlers.

We develop a simple, yet effective method for prefetching array references that contain induction variables in the index expression. An induction variable is incremented or decremented by the same value during each loop iteration. We detect the induction variables using an analysis that is able to detect general recurrences, including those involving linked structures. We formalize the recurrence analysis as a data-flow problem. Prior approaches are ad hoc, or focus on either arrays or linked structures, but cannot detect both.

We evaluate array prefetching using benchmark programs from the Jama library [46] and the Java Grande benchmark suite [14]. Our results show that our simple prefetching implementation is very effective on array-based Java programs on an aggressive out-of-order processor. Prefetching reduces the execution time by a geometric mean[1] of 23%, and the largest reduction is 58%. We see large improvements on several kernels, include matrix multiplication, LU factorization, SOR, and Cholesky factorization. In SOR, prefetching eliminates all memory stalls and reduces execution time by 46%. Performance degrades in one program, FFT, because of a large number of conflict misses caused by a power of 2 data layout and access of a large 1-D array that make prefetching counterproductive.

We augment these results with a case study of matrix multiplication to explore the utility of additional loop transformations to schedule prefetches more carefully in the spirit of the previous work by Mowry et al. [79]. We find that prefetching on modern architectures is less sensitive to precise scheduling via loop transformations, but loop transformations may provide further improvements in some cases. The additional functional units and out-of-order execution in modern processors are able to hide the cost of superfluous prefetch instructions.

[1]We use the geometric mean because we compute the mean of normalized execution times.

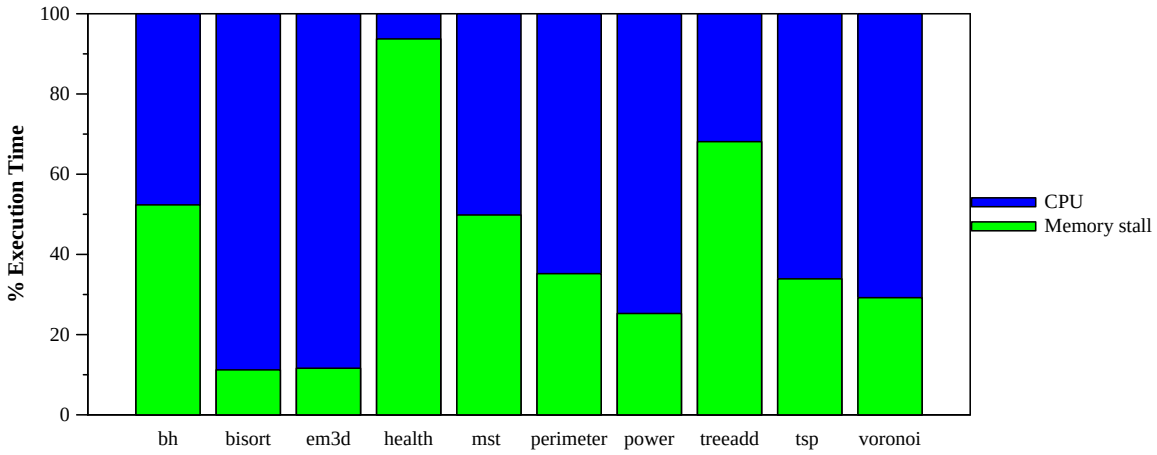


Figure 1.2. Memory Penalty in Java Programs Containing Linked Structures

Our technique is much simpler and faster than existing array software prefetching techniques because it does not require array dependence testing or loop transformations. These characteristics make it suitable for a just-in-time (JIT) compiler, but we leave that evaluation for future work.

1.2 Prefetching Linked Structures

The memory penalty can also be high for object-oriented programs that frequently traverse linked data structures. Figure 1.2 illustrates the percentage of time spent servicing memory requests in an object-oriented Java implementation of the Olden benchmark suite [17]. We compile the programs using Vortex, an ahead-of-time compiler. Memory stalls account for 15% to 95% of the execution time running on RSIM.

Prefetching linked structures is difficult because distinct dynamically allocated objects are not necessarily contiguous in memory, and the access patterns in memory may be unpredictable or erratic. Given an object `o`, we know the address of objects that `o` references, and we cannot prefetch other objects without following pointer chains. Recent pointer prefetching work considers C programs only [66, 70, 90, 56, 101]. Object-oriented Java programs pose additional analysis challenges because they mostly allocate data dynamically, contain frequent method invocations, and often implement loops with recursion.

Linked structure traversals are similar to induction variables. A statement in each loop iteration updates an object by the same field expression, *e.g.*, `o = o.next`. A simple extension to the data-flow analysis for discovering induction variables enables the recurrence analysis to recognize linked structures also. Thus we can use the same unified analysis to discover prefetch opportunities in linked structures and arrays.

Our results show that compile-time prefetching is effective on object-oriented programs that contain linked structures. We find that object-oriented programs often cross procedure boundaries during linked structure traversals. The recurrence analysis is successful in detecting most traversal patterns in the presence of encapsulation and recursion. Our compiler generates prefetch instructions wherever the program traverses a linked structure. We implement three prefetch techniques: greedy, jump-pointer, and stride prefetching, which reduce run time by a geometric mean of 5%, 10%, and 9%, respectively. Greedy prefetching inserts prefetches for directly connected objects. Jump-pointer prefetching uses a compiler-added field to prefetch objects further away in a linked structure. Stride prefetching inserts a prefetch for *n* bytes ahead or behind the current object in a linked structure. The largest reduction is 53%, which occurs with stride prefetching. Even with prefetching, memory latency is still a problem, so future work should combine other techniques with prefetching.

1.3 Organization of Dissertation

We organize the remainder of this dissertation as follows. In Chapter 2, we describe background material and discuss the related work. We present a basic overview of the memory hierarchy, and several methods for improving memory performance. We also present the foundations for the static analysis techniques that we use in our compiler. At the end of the chapter, we discuss the related work in data prefetching separately.

Chapter 3 describes our recurrence analysis. The analysis discovers loop induction variables and linked structure traversals. We first define our intraprocedural analysis and

then our interprocedural analysis. We present extensions to the basic analysis to handle assignment of data-flow information to object fields and arrays. We also describe two other data-flow analyses that improve our prefetching techniques. These analyses compute array sizes and determine which object fields are shared or unshared.

Chapter 4 presents the prefetch algorithms. We first describe our array prefetching implementation. Our prefetch algorithm does not require loop transformations, or expensive data dependence analysis. Then, we describe greedy, jump-pointer, and stride prefetching for linked structures. We show how we use the recurrence analysis to determine *what* to prefetch, and show how the different prefetch algorithms determine *when* to generate prefetch instructions. Chapter 4 also describes the details of our compiler implementation.

In Chapter 5, we evaluate the prefetch algorithms. We compile Java programs using Vortex [34], an ahead of time compiler that produces SPARC assembly, with and without prefetching, to compare the performance benefits directly. We run the programs on RSIM [84], a simulator for an aggressive out-of-order superscalar processor, to obtain detailed performance statistics. We evaluate Java programs from the Olden [17], Jama [46], and Java Grande [14] benchmark suites.

In Chapter 6, we turn on garbage collection and evaluate the effect of garbage collection on prefetching. We run experiments using different heap sizes, and discuss the effect of garbage collection on prefetching. We show that memory performance during the garbage collection phase is very poor. To improve the performance of the collector, we add prefetch instructions at different steps during the collection algorithm. We show that prefetching can help improve the memory performance of garbage collection as well.

Finally, Chapter 7 provides a summary of the contributions in this dissertation. We also discuss directions for future work.

1.4 Summary of Contributions

We make the following contributions in this dissertation:

1. We develop a new method for detecting recurrences in programs. We detect recurrences in linked structures and indices of array references. Our approach uses interprocedural and intraprocedural data-flow analysis. We do not require explicit definition-use chains or for the program to be in static single assignment (SSA) form. Our analysis unifies the discovery of recurrences in linked structures and arrays. We apply the analysis to compiler-generated data prefetching, but we believe that compilers can use the analysis in other domains, such as data layout optimization.

2. We develop a new technique for prefetching arrays. We prefetch arrays that use induction variables. We do not require array dependence analysis or loop transformations. We evaluate prefetching on a set of scientific Java programs. Our array prefetching technique reduces the execution time by more than 15% in 6 of the 12 programs from the Jama library and Java Grande benchmark suite.

3. We implement a new compiler technique for linked-structure prefetching in Java programs. We implement three prefetching algorithms: greedy prefetching, jump-pointer prefetching, and stride prefetching. We find that interprocedural analysis is necessary to discover many of the important linked structure accesses. Our results show that jump-pointer prefetching is able to achieve the largest performance improvements, but may degrade performance if the compiler is not careful when creating the jump-pointers. Stride prefetching produces results similar to jump-pointer prefetching, but the results depend on the layout of the linked structures. Greedy prefetching produces the smallest improvements, but does not increase the execution time in any of the programs.

4. We evaluate the effect of prefetching on garbage collection. The Vortex run-time system uses a generational copying garbage collector. The data reorganization that the collector performs potentially affects the performance of the linked-structure prefetching methods. We quantify the effect using the Olden benchmarks. We also show

that memory performance during garbage collection is consistently poor. We add prefetch instructions to the collection algorithm, and show that prefetching can reduce the execution time of the collector.

We believe that it is important for compilers to analyze and optimize both arrays and linked structures in Java programs. We develop a unified whole-program data-flow analysis for identifying recurrences and inserting prefetches in Java programs. We show that our data prefetching algorithm is effective in improving the memory performance of Java programs.

CHAPTER 2

BACKGROUND AND RELATED WORK

In this chapter, we provide background material and discuss related work. We organize the chapter as follows. In Section 2.1, we describe the memory subsystem of many modern computer architectures. A main focus of this dissertation is to improve performance by reducing the memory penalty in programs. Section 2.2 discusses existing techniques for improving memory performance. We describe methods for tolerating latency and program transformations to utilize the cache more effectively. Our approach uses static analysis to reason about programs and obtain information to optimize programs. The specific type of static analysis is data-flow analysis. Section 2.3 describes the foundations of interprocedural and intraprocedural data-flow analysis. We develop a data-flow analysis that generalizes the detection of induction variables. We present related work for induction variable analysis in Section 2.4. Section 2.5 summarizes the compiler infrastructure that we use to implement and evaluate our optimization techniques. The specific technique we use to improve memory performance is data prefetching. We discuss the related work for prefetching in Section 2.6.

2.1 The Memory Hierarchy

Over the last few decades, a substantial amount of research has focused on cache design and improving their effectiveness [97]. There are several reasons that researchers continue to investigate new techniques to improve cache effectiveness. The gap between processor and memory speed continues to grow, so it is increasingly important to develop methods

to reduce the impact of the growing gap. We continue to see programs that spend a large fraction of time waiting for memory.

Modern computer architectures contain deep memory hierarchies to achieve high performance. The memory hierarchies contain multiple levels of cache that maintain recently accessed data. The ability to keep data in a cache is crucial to achieving high performance. There is a delicate balance between the cache size and access time. The access time to the cache at level l is faster than the cache at level $l+1$, but the cache size of level l is smaller than the cache size of level $l+1$. As processor speed continues to increase, first level caches must remain small in order to achieve one or two cycle access times.

Caches improve performance by taking advantage of data locality, which is the property that programs tend to access the same memory location or nearby locations frequently within a short time period. The two general classifications of data locality are *temporal* and *spatial* locality. Temporal locality occurs when one or more statements reference the same data at different times, and spatial locality occurs when one or more statements reference nearby memory locations.

A cache is divided into fixed sized blocks, called *cache lines*. The cache lines are grouped into sets. The number of cache lines in a set specifies the *associativity* of the cache. A cache divided into sets of size n is *n-way set associative*, and when n is 1, the cache is *direct-mapped*. A cache line is associated with a specific set, but it may be located anywhere in the set. Most caches use a least recently used (LRU) policy to determine which cache line to evict when the set is full.

Hill and Smith categorize a cache miss as compulsory, conflict, or capacity [47]. A *compulsory* miss is the first access to a cache line. A *capacity* miss occurs when the cache size is too small to hold all the cache lines referenced by a program. With sufficient capacity, a *conflict* miss occurs when multiple cache lines are mapped to the same set in the cache, and the program subsequently references an evicted line.

2.2 Improving Memory Performance

There are many techniques for improving memory performance. These techniques can be categorized as techniques for tolerating cache latency or improving cache utilization.

2.2.1 Tolerating Latency

One method to improve memory performance is to tolerate cache miss latencies. The goal of latency tolerating techniques is to perform useful work between the time when a program requests data and the time when the program uses the data. In a very simple in-order processor, the request and use occur at the same time, and the processor stalls until the memory subsystem transfers the data from memory to a register. Modern processors are able to separate the tasks of requesting and using data. The difficulty is finding enough work to perform between the request and use.

We discuss three techniques for tolerating cache latency. These include *prefetching*, the focus of this dissertation, *multithreading*, and *out-of-order* execution.

2.2.1.1 Prefetching

Most high performance architectures contain several simple hardware mechanisms for hiding memory hierarchy access costs. Early cache designs allowed only a single outstanding memory access to occur. Thus, all memory accesses stalled the processor until completed. Kroft introduced lockup-free caches [62] to enable multiple concurrent memory accesses. Lockup-free caches permit *non-blocking* loads that do not stall the processor until a future instruction references the data. Lockup-free caches require a mechanism, such as miss status handling registers (MSHRs), to maintain information about pending loads. A processor limits the number of allowable pending loads, and stalls when the maximum number of loads are outstanding.

Prefetching is a hardware or software technique for tolerating cache latency by providing a mechanism to separate the request and use of data explicitly. A software approach explicitly inserts prefetch instructions into the instruction stream to perform data prefetch-

ing. In a hardware approach, the processor contains a mechanism to automatically prefetch data without the use of extra instructions.

A prefetch initiates a transfer of data in the memory hierarchy prior to the demand request. A prefetch instruction is similar to a non-blocking load. The main difference is that a prefetch instruction typically does not cause an exception if the address is invalid. Another difference is that a prefetch instruction does not load the data into a register. Instead, the prefetch moves data closer to the processor in the memory hierarchy.

A simple form of hardware prefetching is long cache lines. If an object is smaller than a cache line, then adjacent objects are also brought into the cache when a miss occurs. Long cache lines often improve the performance of programs that have spatial locality. Long cache lines are not always effective because not all programs exhibit spatial locality. The main disadvantage of long cache lines is the increase demand for available bandwidth to memory. In a multiprocessor, long cache lines increase false sharing, which occurs when two processors require separate objects that reside on the same cache line. We discuss more complex prefetch methods in Section 2.6.

An effective prefetch method must determine what to prefetch and when to issue the prefetch. We evaluate prefetching effectiveness by categorizing dynamic prefetches as follows:

- The data in a *useful* prefetch arrives on time and is accessed by the program.
- The latency of a *late* prefetch is only partially hidden because a request for the cache line occurs while the memory system is still retrieving the cache line.
- The cache replaces an *early* prefetch before the cache line is used. If the cache line is never accessed, the prefetch is early.
- An *unnecessary* prefetch hits in the L1 cache, or is coalesced into an MSHR.

2.2.1.2 Multithreading

Multithreading is a technique for tolerating latency by switching contexts to a pending thread when the processor stalls [98, 2, 107]. To improve memory performance, a processor switches to a thread that can execute instructions when a memory stall occurs in the current thread. One advantage of multithreading over prefetching is that it is not necessary to determine what to prefetch and when to issue the request. A multithreaded architecture simply switches to another context whenever a cache miss occurs. The main disadvantage is that the program must contain enough parallelism so that when a cache miss occurs, another context is able to execute. Another disadvantage is that hardware implementations require complex architectural changes to support multiple contexts, which places additional pressure on hardware resources.

Recently, several researchers have investigated methods to use additional threads in a multithreaded system to prefetch data [36, 91, 28, 69, 99]. These methods implement new hardware, and some also use software support, to create separate threads that speculatively run ahead of the main thread and prefetch data. The existing methods use different techniques for deciding when to create and how to manage the speculative threads.

2.2.1.3 Out-of-order Execution

Out-of-order (OOO) execution is a hardware technique for tolerating a small amount of memory latency. A processor with out-of-order execution is able to execute instructions when the operands become available rather than in the order that the program specifies. Out-of-order processors exploit instruction level parallelism by allowing other instructions to execute when an instruction stalls in the processor waiting for a resource. Out-of-order processors use a fixed-size instruction window from which instructions may be executed. In order to preserve program semantics, the processor retires the instructions in-order. The amount of latency that an out-of-order processor is able to tolerate depends upon the amount of instruction level parallelism (ILP) and the size of the instruction window. Most

high performance commercial processors support out-of-order execution including the Alpha 21264 [57], MIPS R10000 [121], and Intel Pentium [48].

2.2.2 Program Transformations

Developing code and data transformations to improve cache utilization is a very active area of research. Code transformations attempt to reorder the program instructions to utilize the data in the caches more effectively. Data transformations reorganize data layouts to increase reuse. These techniques attempt to improve the cache effectiveness by reusing data that is in the cache already. In contrast, data prefetching attempts to tolerate cache latency by moving data into the cache speculatively. The techniques we describe below often complement data prefetching.

A typical code transformation restructures the computation in a loop to improve the spatial or temporal locality of the program, moving reuse of the same or adjacent locations closer together in time [1, 64, 73]. Loop tiling is a classic locality optimization that works by transforming a loop nest so that array accesses reuse smaller blocks of data that fit into the cache. Compilers typically employ advanced static analysis techniques to determine when and how to perform a transformation. The static techniques perform data dependence analysis to determine the access patterns in programs [63]. The static analysis needs to determine when the transformation is legal. If the compiler is unable to determine the legality then it cannot perform the optimization. An advantage of data prefetching is that compilers can be more aggressive in determining opportunities because legality is not a requirement.

Data transformations attempt to co-locate data to improve spatial locality. Data transformations are often performed at run time or require profiling information to reorganize data. Calder, Krintz, John, and Austin present and evaluate an approach for cache conscious data placement [15]. They reduce cache conflicts by using profiling information to relocate objects. Chilimbi and Larus rearrange data at garbage collection time to improve data

locality in object-oriented programs [27]. Chilimbi, Hill, and Larus perform object reordering at allocation time to improve cache locality in C programs [25]. Chilimbi, Davidson, and Larus evaluate structure splitting and field reorganization to improve cache performance [24]. Truong, Bodin, and Seznec use program profiling to evaluate two data layout techniques, field reorganization and instance interleaving, to improve the cache behavior of dynamically allocated data in C programs [105]. Since they apply these transformations by hand, Truong also describes plans for automating the data layout techniques based upon profile information [104]. Kistler and Franz evaluate a profile-based optimization that reorders members in objects to improve spatial locality [59]. Franz and Kistler also propose physically splitting frequently and infrequently accessed members of objects to improve cache performance [38].

Other data transformations apply specifically to heap allocated data in a garbage collected environment. Moon describes a mostly depth-first copying garbage collection algorithm to improve the page locality of Lisp programs [75]. Stamos evaluates five static grouping garbage collection algorithms to improve the locality of objects [100]. Courts proposes a dynamic garbage collection algorithm for improving locality [30]. Wilson, Lam, and Moher propose and evaluate different static copying algorithms to improve locality in garbage collected systems [111]. They introduce hierarchical decomposition that combines breadth-first and depth-first traversals in a copying algorithm. The algorithm is similar to Moon's, but does not rescan any locations. Wilson, Lam, and Moher also empirically examine the cache performance of generational garbage collection [112]. Their results show that miss rates in garbage collected systems are not very high. Reinhold empirically examines the cache performance of garbage collected programs, and looks at both the mutator and the collector [88]. Boehm experiments with adding prefetching to a non-moving, mark-sweep garbage collector [12]. Boehm prefetches objects during the mark phase. Boehm shows that prefetching improves performance on a set of microbenchmarks running on a Pentium II and HP PA-RISC machine.

In this section, we describe many methods for improving memory performance. In this dissertation, we focus on software data prefetching to tolerate memory latency. Our approach is applicable to a wide range of programming styles, and does not require complex hardware mechanisms. We also do not require advanced code or data transformations. Compilers must be conservative when applying code or data transformations to ensure program correctness. Although software prefetching does require compiler support, the prefetch instructions do not affect the correctness of programs. We use data-flow analysis to discover prefetch opportunities. In the following section, we describe the foundations of data-flow analysis.

2.3 Data-Flow Analysis

A data-flow framework provides a generic mechanism for specifying a program analysis [58, 55]. Data-flow analysis is a pervasive program analysis technique in many compilers. In this section, we discuss the fundamentals behind data-flow analysis. We separately define data-flow analysis for use within a procedure and for a whole program. Nielson et al. present an excellent discussion of intraprocedural and interprocedural analysis [80].

2.3.1 Intraprocedural Data-Flow Analysis

In this section, we describe the basics behind intraprocedural data-flow analysis. An *intraprocedural* data-flow analysis operates on a single procedure and makes conservative assumptions about procedure calls and returns.

A monotone data-flow analysis framework consists of the following:

- A complete lattice, $(L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$. L is a set that defines the elements in the lattice. The $\sqsubseteq$ function defines a partial ordering of the elements in L. Thus $\sqsubseteq$ is reflexive, transitive, and anti-symmetric. For any $X \subseteq L$, $\sqcup X$ denotes the least upper bound of X. Formally, $\forall l \in L, \sqcup X \sqsubseteq l$ if and only if $X \sqsubseteq l$. The greatest lower bound, $\sqcap X$, is defined by replacing $\sqsubseteq$ with $\sqsupseteq$. The $\sqcup$ and $\sqcap$ functions are known as the *join* and *meet*

operations, respectively. The join and meet functions are idempotent, commutative, and associative. The elements $\perp$ and $\top$ are known as *bottom* and *top*, respectively. Thus $\perp = \sqcup\emptyset = \sqcap L$, and $\top = \sqcap\emptyset = \sqcup L$.

- A set F of monotone transfer functions over L. A function $f : L \to L'$ is monotonic if $x \sqsubseteq y$ implies $f(x) \sqsubseteq f(y)$. F is closed under composition, and contains the identity function. There is a function f for each statement in a procedure. Intuitively, f computes the data-flow information that captures the semantics from executing part of a program.

Solving an analysis problem using a data-flow framework requires a flow graph and a function that maps each node in the flow graph to a function in F. A flow graph is a representation of the control flow of a procedure. Formally, $G = (N, E, n_0)$ is a flow graph where N is the set of nodes that represent the statements in a procedure, E is the set of edges that represent possible control flow, and n_0 is the start node of the procedure. The set of edges is a subset of $N \times N$. An edge (n_i, n_j) indicates that control flow may leave node n_i and enter n_j.

We denote the function in F associated with node $n \in N$ by f_n. The data-flow instance yields the following equations for an analysis:

$$\text{Analysis}_{in}(n_j) = \bigsqcup_{(n_i,n_j)\in E} \text{Analysis}_{out}(n_i) \tag{2.1}$$

$$\text{Analysis}_{out}(n_j) = \text{f}_{n_j}(\text{Analysis}_{in}(n_j)) \tag{2.2}$$

Equations 2.1 and 2.2 define a *forward* analysis. Equation 2.1 represents the data-flow information upon entering a statement in the procedure, and Equation 2.2 represents the data-flow information when exiting a statement. A *backward* analysis begins with the exit node and processes the statements in reverse order. To define a backward analysis, we

change Equation 2.1 so that $(n_j, n_i) \in E$. We also need to swap the meaning of *out* and *in* so that Equation 2.2 represents the data-flow information upon entering a statement, and Equation 2.2 represents the data-flow information when exiting.

An analysis may be *flow-sensitive* or *flow-insensitive*. In a flow-sensitive analysis, the order of statements in the procedure matter. A flow-insensitive analysis ignores control flow within a procedure. In general, a flow-sensitive analysis is more accurate, but usually takes longer to perform.

2.3.2 Interprocedural Data-Flow Analysis

An *interprocedural* data-flow analysis takes procedure calls and returns into account during the analysis. The interprocedural analysis extends the definition of an intraprocedural flow graph G to include procedure calls and returns. The interprocedural flow graph contains a node for each procedure in a program. An edge connects two nodes if there is a calling relationship between the procedures. Formally, $IPG = (N', S, E', m_0)$ where N' is the set of procedures in a program, S is the set of call site labels, E' is a set of labeled edges representing procedure calls, and m_0 is the main procedure. The set of edges is a subset of $N' \times S \times N'$. The call site label is necessary to distinguish between multiple calls from one procedure to the same target procedure.

An interprocedural analysis operates on the IPG and uses Equations 2.1 and 2.2. In addition, the interprocedural analysis contains two transfer functions for each call. One transfer function is for the call, f_c, and the other is for the return, f_r. The callee also contains two transfer functions: one for the start of the procedure, f_s, and one for the end of the procedure, f_e.

At a procedure call, the transfer function f_c creates a new context and the initial data-flow information for analyzing the callee. Upon return, the transfer function f_r restores the caller's data-flow information and adds new information from the callee for the return.

An interprocedural analysis may be *context-sensitive* or *context-insensitive*. In a context-insensitive analysis, the data-flow information for a procedure is a combination of the information from all call sites. The framework analyzes each procedure only once using the combined information. In a context-sensitive analysis, the framework analyzes target procedures in each distinct calling context.

At a call site with multiple potential callees, *e.g.*, due to a virtual method call, the interprocedural framework applies the join operator to the results of the analysis for each possible callee.

2.4 Induction Variable Analysis

In this section, we describe induction variable analysis, and we survey the related work. We develop a general recurrence detection analysis in this dissertation that subsumes induction variable analysis.

An induction variable is incremented or decremented by the same value during each loop iteration. An example of an induction variable is the expression `i = i + c` occurring in a loop. During each iteration of the loop, the variable `i` is incremented by a loop invariant value, `c`. Traditional algorithms for finding induction variables are either loop-based [4] or use static single assignment (SSA) form [31].

The original use for induction variable detection was operator strength reduction [67, 4]. The initial algorithms typically require the compiler to compute reaching definitions and loop invariant expressions. The algorithms are conservative and find simple linear induction variables. The PTRAN compiler uses an optimistic approach, and assumes variables in loops are induction variables until proven otherwise [6]. Gerlek, Stoltz, and Wolfe present a demand driven SSA approach for detecting general induction variables by identifying strongly connected components in the SSA graph [40]. Gerlek et al. present a lattice for classifying different types of induction variables. They detect a wide range of induction variables including linear, polynomial, exponential, periodic, and wrap-around. Haghighat

and Polychronopoulos also categorize different types of induction variables for use in parallelizing compilers [44]. Ammarguellat and Harrison describe an abstract interpretation technique for detecting general recurrence relations, which includes induction variables [8]. The approach requires a set of patterns, which they call templates, that describe the recurrences.

Wu, Cohen, and Padua describe a data-flow analysis for discovering loop induction variables [114, 115]. The analysis computes whether a variable increases or decreases along a given execution path, and the minimum and maximum change in value. The authors compute closed form expressions from the distance intervals to perform array dependence testing. Wu et al.'s induction analysis does not compute information about linked structures. We present a different data-flow analysis for identifying induction variables, and we use the analysis to discover both induction variables and linked structure traversals. We also use the analysis for prefetching rather than array dependence testing.

2.5 Vortex: A Compiler Infrastructure

This dissertation describes several new compiler analyses and optimizations, and presents empirical results. We implement and evaluate these new techniques in Vortex, an existing compiler infrastructure. Vortex is an optimizing compiler for object-oriented languages developed at the University of Washington [34]. We briefly describe Vortex in this section. In Section 4.5, we describe Vortex in more detail, and we discuss our extensions to support prefetching.

Vortex supports several object-oriented languages including Cecil, Java, C++, Modula-3, and Smalltalk. The compiler itself is written in Cecil. The implementation work in this dissertation uses the Java front-end. At a high level, Vortex performs the following steps:

1. Convert object-oriented program to an intermediate representation.

2. Perform interprocedural optimization on the complete program. This step is optional and performed only when the user specifies an interprocedural optimization.

3. Perform intraprocedural optimization. The backbone of the Vortex compiler is a flexible data-flow analysis framework.

4. Generate SPARC assembly code.

Vortex does not operate directly on Java source files. Instead, Vortex converts Java class files (byte codes) to a high-level internal representation. The intermediate language represents high-level object-oriented features. Vortex performs all analysis and optimization on the intermediate representation of the program. Vortex represents the control flow and individual statements as a graph. The nodes in the graph represent operations, and the arcs between the nodes indicate either data or control flow. During the compilation process, Vortex performs several conversions on the intermediate language to convert high-level operations to low-level operations. After each conversion, Vortex applies different analyses and optimizations. The output of the compiler is either C or assembly language.

The high-level form closely matches the original program structure. For example, Vortex contains high-level operators for object creation and method calls. The low-level form more closely matches the machine level. The low level no longer represents the object-oriented features. Instead, the graph nodes in the low-level representation are almost a one-to-one match with the assembly instructions.

A central part of the Vortex optimization infrastructure is a general iterative data-flow analysis framework [18, 65]. The framework is parameterized by the properties that we describe in Section 2.3. An important feature of the analysis framework is the ability to compose, or combine, several analyses so that they run together. Each analysis is able to query the results of another analysis when running. Running multiple analyses at the same time potentially improves precision by eliminating phase ordering problems. This feature

also improves code reuse by making it easy to incorporate the results of other analyses into a new analysis, and reduces compilation time.

2.6 Prefetching: Related Work

In this section, we survey existing data prefetching research with a focus on compiler support. We also refer the reader to a thorough survey of data prefetching techniques for scientific programs by VanderWiel and Lilja [110]. The article presents an overview of existing techniques for software data prefetching, hardware data prefetching, combinations of software and hardware prefetching, and prefetching for multiprocessors.

Sections 2.6.1 and 2.6.2 present related work for array-based software and hardware prefetching, respectively. For completeness, we briefly describe prefetching in multiprocessors in Section 2.6.3, although we do not investigate prefetching for multiprocessors.

In Section 2.6.4, we describe Luk and Mowry's prefetching algorithms for linked data structures. Since this work most closely relates to ours, we describe it separately. We discuss other software and hardware approaches for prefetching linked structures in Section 2.6.5.

2.6.1 Array Prefetching in Software

In this section, we describe previous research that uses software approaches for prefetching arrays. Much of the research in data prefetching has focused on data prefetching for array-based scientific programs. The array prefetching techniques generate prefetch instructions for array references that will likely occur in future loop iterations.

Callahan, Kennedy, and Porterfield present one of the first descriptions and evaluations of software prefetching [16]. They use a very simple algorithm to add non-blocking prefetch instructions to twelve array-based Fortran programs. The algorithm prefetches array references one loop iteration before they are needed. The results show that prefetching improves miss rates, but the overhead of the prefetch instructions may be too large to im-

prove execution times. The authors suggest a few changes to reduce the overheads to make prefetching profitable.

Klaiber and Levy describe an algorithm for software controlled data prefetching that holds the prefetched data in a separate fully-associative buffer instead of the cache [60]. The algorithm works by inserting a prefetch for an array element one or more iterations before the actual load of the datum. Results show that prefetching improves performance on the Livermore Loops benchmarks using average time per memory reference as the metric. The algorithm is most effective on array-based scientific codes, but it also slightly improves performance on two non-numeric programs. Klaiber and Levy also indicate that their algorithm causes little or no increase in bandwidth utilization.

Chen, Mahlke, Chang, and Hwu compare the performance of software prefetching into the cache verses a special prefetch buffer [22]. They do not specifically target either regular or irregular access patterns. Instead their algorithm adds prefetches for as many data loads as possible when they are able to completely hide the memory latency. Their simulation results on a superscalar processor suggests that a prefetch buffer is more effective than a larger cache. One drawback of the research is that Chen et al. use very small (1K or 2K) caches in their evaluation.

Yamada, Gyllenhaal, Haab, and Hwu combine data relocation and block prefetching to improve data cache performance [118]. The hardware support consists of five special instructions to perform data relocation and prefetching. They use a compiler to transform loop nests in scientific programs and add the special instructions. The non-blocking instructions compress and preload arrays into sequential cache locations. The compiler also uses standard cache improvement techniques such as loop unrolling and tiling. Simulation results show improvements in cache utilization and execution speed.

Mowry, Lam, and Gupta describe and evaluate compiler techniques for adding prefetching to array-based codes [79, 78]. This paper is one of the first that reports execution times for compiler inserted prefetching. The algorithm works on affine array accesses within

scientific codes. The algorithm significantly improves performance by as much as a factor of 2. They also show that their algorithm is better than indiscriminate prefetching. The algorithm involves two steps. First the algorithm performs locality analysis to determine array accesses that are likely to miss in the cache. Then, the algorithm uses loop splitting to isolate predicated cache misses, and uses software pipelining to schedule prefetch instructions.

In his dissertation, Selvidge presents profile-guided software data prefetching as a scheduling algorithm [94]. A compiler, called c-flat (Compiler For LAtency Tolerance), uses profile information to identify regular and irregular data reference streams. In Section 2.6.5, we discuss Selvidge's work in the context of prefetching linked data structures. C-flat also works on array-based codes, including indirection arrays. Most of the benefits that Selvidge reports are due to prefetching array elements.

McIntosh extends Mowry's work by focusing on the compiler support necessary for software prefetching [72]. He develops several new compiler techniques to eliminate useless prefetches and to improve prefetch scheduling for array-based codes. McIntosh develops a new technique for detecting cross-loop reuse that provides useful information for improving software prefetching. Cross-loop reuse summarizes data accesses that occur between loop nests as opposed to within a single loop nest, *i.e.*, intra-loop reuse.

Two reports evaluate software prefetching on commercial processors using the HP PA-8000 and the PowerPC. Santhanam, Gornish, and Hsu evaluate software prefetching on the HP PA-8000, a 4-way superscalar processor [93]. The prefetch algorithm concentrates on array references occurring within the innermost loops. Santhanam et al. discuss implementation details and present results showing a 26% speedup on the SPECfp95 benchmark suite. Bernstein, Cohen, Freund, and Maydan describe a compiler implementation for data prefetching on the PowerPC architecture [11]. Bernstein et al. follow Mowry's approach but the only transformation they apply is loop unrolling. Bernstein et al. provide actual

execution times for the SPECfp92 benchmarks and Nasa7 kernels. Improvements occur on only three of the fourteen SPECfp92 programs and six of the seven Nasa7 kernels.

We propose an array prefetch algorithm that is easy to implement and is effective on array-based Java programs. Our algorithm does not require array locality analysis or loop transformations, which prior techniques use. None of the prior methods focus on Java, which implements multidimensional arrays as arrays-of-arrays.

2.6.2 Array Prefetching in Hardware

Hardware prefetching schemes add prefetching functionality without explicit programmer or compiler assistance. The main benefit of hardware schemes is the ability to run existing program binaries, which enables prefetching without recompiling the program. Most hardware mechanisms prefetch only array reference streams. Several of the techniques we describe below require some software support to attain performance improvements.

Smith investigates a simple cache prefetching algorithm called one block lookahead[96, 97]. When a program references cache line *i*, the one block lookahead scheme fetches the next cache line, *i + 1*. Smith shows that one block lookahead is successful in lowering miss rates.

Jouppi proposes and evaluates stream buffers as a mechanism to prefetch data into a separate area from the cache [54]. When a reference misses in the cache, the processor first checks the stream buffer. The stream buffer is a simple FIFO queue; when the processor removes an item from the head of the queue, the stream buffer fetches a new successive address. Palacharla and Kessler extend stream buffers and present a more detailed evaluation [85]. The extensions include a filter to reduce the bandwidth requirements and the ability to prefetch non-unit strides. They conclude that stream buffers work well on regular, scientific codes but not as well on irregular codes.

Baer and Chen propose a purely hardware prefetching scheme for scientific programs[10, 19]. The scheme predicts the execution stream and preloads references with arbitrary con-

stant strides. The hardware mechanism includes a reference prediction table (RPT) and a look-ahead program counter (LA-PC). The RPT maintains state about load and store instructions such as the previous address encountered and the stride value. If an entry appears in the RPT when the LA-PC encounters a load or store, then the hardware predicts the next address to be loaded based upon the previous address and stride value. The mechanism works only for loads and stores with regular accesses. Experiments show reductions in the number of cycles per instruction (CPI) for scientific programs.

Baer and Chen compare the effectiveness of their hardware mechanism to a non-blocking cache [21]. They find that prefetching outperforms a non-blocking cache, in general. They also propose a hybrid scheme that uses a non-blocking cache and hardware prefetching that results in further performance improvements. Chen and Baer conclude that a good optimizer and scheduler are necessary to obtain good results for a non-blocking cache. In later work, Chen briefly describes a user programmable prefetch controller called Hare [20]. The prefetch engine program signals the processor to begin prefetching. Chen discusses compiler support for the prefetch engine, including locality analysis to identify arrays to prefetch. An evaluation on four programs shows memory access time improvements.

VanderWiel and Lilja develop a decoupled prefetching mechanism consisting of an external data prefetch controller (DPC) that uses a small program to control prefetching [109]. At run time, the processor and DPC work separately yet cooperate to perform prefetching. A compiler creates the prefetch program while compiling the original program. The compiler annotates the compiled program to activate the DPC at appropriate points. VanderWiel and Lilja compare the DPC to Chen and Baer's RPT prefetching mechanism and software prefetching [10]. They show execution time improvements over both these schemes on scientific programs.

Lin, Reinhardt, and Burger propose and evaluate a hardware prefetch mechanism that prefetches blocks of data nearby recent misses [37]. The technique does not focus on arrays

explicitly, but does take advantage of programs with spatial locality. The goal of the work is to improve the effectiveness of the L2 cache without degrading performance in programs that are bandwidth intensive. The hardware prefetch engine prefetches data into the L2 cache only when there are idle cycles on the memory channel. The prefetched data has a low replacement priority in the cache. Results show that the hardware prefetch mechanism improves the performance significantly in 10 of the 26 SPEC benchmarks.

Hardware prefetching methods require complex additional hardware. There is a large variation in functionality among the architectures that contain hardware prefetch mechanisms. Some architectures prefetch into the 1st level cache (*e.g.*, the POWER4 [103]), some prefetch into the 2nd level cache (*e.g.*, the Pentium 4 [51]), others prefetch into a special buffer (*e.g.*, the UltraSPARC III [102]), and some prefetch floating point data only (*e.g.*, UltraSPARC III). A software prefetch mechanism requires less complexity. Also, software prefetching increases flexibility by allowing the compiler to determine what and when to prefetch.

2.6.3 Array Prefetching on Multiprocessors

Although we do not evaluate prefetching schemes on multiprocessors, several researchers have investigated prefetching of array-based codes on multiprocessors.

Fu and Patel evaluate two hardware prefetching schemes on a vector multiprocessor system [39]. Mowry and Gupta evaluate software prefetching for array-based programs on shared-memory multiprocessors [77, 78]. Gornish, Granston, and Veidenbaum implement prefetching for shared-memory multiprocessors [42]. Dahlgren, Dubois, and Stenstrom evaluate sequential hardware prefetching and stride prefetching on a shared-memory multiprocessor [32, 33]. In his thesis, Gornish compares software and hardware prefetching, and presents an integrated prefetching scheme for multiprocessors [41]. Zhang and Torrellas describe techniques for prefetching pointer-based programs on multiprocessors using a scheme that is similar to greedy prefetching [123]. Tullsen and Eggers evaluate compiler

assisted software prefetching on shared-memory multiprocessors [106]. Ranganathan, Pai, Abdel-Shafi, and Adve examine the effectiveness of software prefetching for scientific programs on a shared-memory multiprocessor built with modern ILP processors [87].

2.6.4 Prefetching Linked Structures: Luk and Mowry

Luk and Mowry develop three prefetching schemes for recursive data structures (RDS) [68, 70, 71]. These include *greedy* prefetching, *history-pointer* prefetching, and *data-linearization*. In the initial work, Luk and Mowry have a compiler implementation for greedy prefetching only. But, in his dissertation, Luk implements and evaluates all three techniques [68]. They use the Olden benchmarks to evaluate and compare the performance of prefetching RDSs [17]. Their experiments show that greedy prefetching can increase performance by as much as 45%. Results also show that greedy prefetching always performs as well or better than SPAID, another non-numeric prefetch technique that we discuss below. Luk and Mowry use a very simple alias analysis and very little locality analysis to determine what and when to prefetch. They also show that improving the locality analysis also improves performance.

The prefetching algorithm uses type declarations to discover recursive linked data structures and control flow to recognize linked structure traversals. They define a recursive data structure (RDS) as a record type containing at least one reference that points either directly or indirectly to itself. The compiler looks at loops and recursive procedure calls to determine where programs access RDSs. The compiler inserts the appropriate prefetch instruction when traversing the RDS depending on the prefetch algorithm. For jump-pointer prefetching, the default is to update the jump-pointer during traversals. The compiler relies on the user to identify memory allocation sites to add jump-pointers at the allocation point.

Our contributions over this previous work include a new intra and interprocedural data-flow analysis for discovering objects to prefetch, and an evaluation on a suite of Java programs. Luk and Mowry do not perform interprocedural analysis, but they do detect

self-recursive calls. Our analysis works in the presence of virtual method calls, and when data-flow facts are assigned to object fields and arrays. We also detect indirect recurrent reference variables. We developed our implementation of jump-pointer prefetching simultaneously with Luk. We also develop a recurrence analysis that is able to detect both linked structure and array traversals. We use the same analysis to drive the prefetch algorithms for arrays and linked structures.

2.6.5 Other Linked-Structure Prefetching Techniques

In this section, we describe existing techniques for software and hardware prefetching of pointer-based programs. Some of the techniques also apply to prefetching irregular array accesses.

Harrison and Mehrotra add an indirect reference buffer (IRB) to the cache to perform hardware prefetching on programs with pointers and indirect array references [45, 74]. The IRB is able to prefetch regular array references as well. The IRB consists of a recurrence recognition unit and a prefetch unit that cooperate to detect recurrent address sequences and generate prefetches based upon the reference stream pattern. For linked list traversals, the IRB is a hardware implementation of a greedy prefetching algorithm that prefetches the next element in a linked structure. Most of the loads involved in recurrent address sequences exhibit either linear patterns or a combination of linear and indirect patterns. Although they show improvements when using an idealized model (infinite IRB and zero latency prefetch), they do not see improvements when using a realistic model because their benchmarks already exhibit good cache performance.

SPAID (speculatively prefetching anticipated interprocedural dereferences) is a compile-time algorithm for prefetching data pointer arguments to function calls [66]. Using a simple heuristic to prefetch function arguments, they show cache miss improvements, but not execution time results. They use small C and C++ benchmarks and a statistical cache model instead of a cycle-by-cycle simulation. Results show that SPAID achieves the best

results when prefetching one argument at a call. Unfortunately, this approach is limited by the amount of latency it is able to tolerate. Luk and Mowry show that greedy prefetching is a more effective algorithm.

Joseph and Grunwald use a Markov predictor hardware mechanism to prefetch data into a special buffer [53]. The Markov predictor records the cache miss address stream at run time using a probabilistic transition table. Upon a cache miss, the prefetch mechanism looks up the address in the table to prefetch a value with a high probability of also missing. Joseph and Grunwald evaluate the effectiveness of Markov prefetching using commercial workloads that mostly contain unstructured references (*i.e.*, non-scientific programs). They also compare Markov prefetching to stream buffers (*e.g.*, [54]) and stride prefetching (*e.g.*, [21]). The Markov prefetcher generates the greatest number of useful prefetches, but also increases the bandwidth consumed more than the other methods. As with most hardware prefetching mechanisms, the Markov prefetcher requires a training period before it can issue prefetches. Another drawback of the Markov prefetcher is the amount of memory that is necessary to store the table. The Markov predictor uses 1 MB for the predication table in the experiments.

Roth and Sohi introduce a hardware mechanism called dependence-based prefetching for prefetching linked data structures [89]. The hardware mechanism recognizes recurrent pointer accesses by identifying producer-consumer instruction pairs. Hardware mechanisms identify loads that produce addresses and instructions that consume those addresses. The hardware uses the producer-consumer information to issue prefetch requests. The dependence-based prefetch mechanism achieves speedups of up to 25% using the Olden benchmarks although most improvements are much smaller. Although the approach successfully predicts linked structure traversals, it requires several complex hardware mechanisms. Dependence-based prefetching is able to prefetch a wider variety of linked structures than mechanisms such as the IRB.

In later work, Roth and Sohi discuss jump-pointer prefetching for tolerating memory latencies for linked data structures [90]. Roth and Sohi present four schemes for jump-pointer prefetching that can be implemented in software, hardware, or a combination of the two. The schemes are *queue*, *full*, *chain*, and *root* jumping. They use the four different versions for specific data structure instances. Queue jumping is applicable on simple linked structures that contain nodes of the same type. In full jumping, each node may contain multiple jump-pointers, which prefetch nodes of different types. Full jumping is useful on generic data structures in which each node contains a pointer to another node of a different type. Roth and Sohi use the term "backbone and ribs" to refer to this type of structure. Chain jumping achieves the benefits of full jumping, but it uses only a single jump-pointer. At the beginning of a loop, the hardware prefetches the backbone node using the jump-pointer and, at the end of the loop, the hardware prefetches the rib node using the natural pointer. Finally, root jumping uses the existing pointers for prefetching, but attempts to prefetch the next element in the linked structure during each loop iteration. Roth and Sohi run experiments evaluating their jump-pointer prefetching schemes using the entire Olden benchmark suite. They implement the software schemes by hand.

Rubin, Bernstein, and Rodeh combine data reorganization and prefetching of recursive data structures [92]. They create virtual cache lines (VCLs) that group dynamically allocated objects with spatial locality. The design of VCLs supports efficient insertion and deletion operations by allocating a small amount of extra space on each VCL. The amount of extra space may be parameterized to improve performance. Experiments show that VCLs improve performance when repeatedly searching linked lists. Using VCLs also improves performance in programs with insertion and deletion operations. Rubin et al. also apply Luk and Mowry's greedy prefetching algorithm to VCLs and run experiments on the PowerPC 604e. Rather than prefetching individual elements in a linked list, Rubin et al. prefetch VCL elements. The size of each VCL depends upon the cache line size and the number of allowable outstanding prefetches. Results show that prefetching VCLs is

better than prefetching individual elements when the amount of work performed on each element is very small. It is difficult to assess the full benefit of VCLs because they perform experiments on a single toy example that involves repeated scans of a linked list.

Karlsson, Dahlgren, and Stenstrom describe a technique called prefetch arrays for prefetching linked data structures [56]. Their focus is on prefetching short linked data structure, such as lists in hash tables or trees when the traversal path is unknown. The technique works by creating an array of jump-pointers that are prefetched during each iteration or just prior to a loop. They present a software solution only, and a combined software and hardware solution. The authors identify prefetching opportunities and add prefetch instructions to programs by hand. Karlsson et al. present results using the Olden benchmarks on a single issue, in-order processor. The techniques they describe appear to be heavily dependent upon specific programming idioms and are difficult to apply automatically.

Selvidge discusses prefetching linked lists as well as arrays in his dissertation [94]. Selvidge uses profiling information to discover prefetching opportunities and uses the information in a compiler to insert prefetch instructions during the scheduling phase. The algorithm works by matching specific patterns in the strongly connected components of a data-flow graph. For example, the compiler contains a pattern for matching specific simple linked list traversals. Selvidge's prefetching technique uses multiple prefetch instructions during each iteration to prefetch the next element in the linked list. None of the benchmarks contain enough linked list traversals to show any benefit from prefetching.

Ozawa, Kimura, and Nishizaki discuss a technique for preloading in non-numeric programs [81]. Preloading is a form of prefetching where data is loaded into a register instead of the cache. Preloading attempts to place data in a register far enough in advance to hide the latency of a cache miss. Ozawa et al. classify load instructions into two categories: list access and stride access that correspond to traversing a linked list and an array, respectively. The authors propose several effective scheduling heuristics that move loads of list/stride accesses across basic blocks to increase the distance between a load and a use.

The preloading heuristic slightly increases code size and the number of spilled registers, but they show execution time improvements in most of the SPEC92 benchmarks.

Kohout, Choi, Kim, and Yeung propose and evaluate a programmable prefetch engine for prefetching linked structures [61]. The technique prefetches a single linked list sequentially, but attempts to prefetch multiple lists simultaneously. For this technique to be effective, the compiler or programmer must identify independent linked structures. The programmable prefetch engines uses the compiler or programmer information to issue prefetches. Kohout et al. evaluate their prefetch technique on the Olden benchmarks and several of the SPEC CPU2000 benchmarks, and show significant improvements. They also compare their results to jump-pointer prefetching and prefetch arrays.

Stoutchinin et al. develop and evaluate a new algorithm for prefetching linked structures based upon the idea of induction pointers [101]. They identify linked structure traversals in a loop through pointer load instructions that are updated by a constant offset in each iteration. The prefetch algorithm generates prefetches only when sufficient bandwidth is available. The technique relies on the run-time system to allocate objects a constant distance apart. They implement the prefetch algorithm in the SGI MIPSpro compiler, and evaluate the effectiveness using SPEC CINT95 and SPEC CPU2000 benchmarks on the MIPS R10000 architecture. Prefetching improves performance in three of the ten benchmarks by 15% to 35%.

Chilimbi and Hirzel design and evaluate a dynamic prefetching scheme that uses on-line profile information to discover prefetch opportunities [26]. The prefetch technique works in several phases. First, a low-overhead profile phase gathers data reference streams. After profiling, the run-time system analyzes the data streams to determine prefetch opportunities and dynamically generates code to add prefetch instructions to the program. The program executes the prefetch instructions for a period of time, and then the run-time system starts the profiling phase again. Initial results on a few of the memory bound SPEC CPU2000 benchmarks show performance improvements.

Wu et al. use profiling to identify prefetching opportunities in programs with irregular accesses [117]. Their insight is that irregular programs contain a large number of loads with near constant strides. The compiler uses the profile information about loads with constant strides to generate prefetch instructions. Wu et al. show that the profile information is stable across input sets and that the profile overhead is low. They evaluate the prefetch technique on the SPEC CPU2000 programs, and they show large improvements in three of the programs. Wu et al. improve the profile information to identify more effective prefetch opportunities [116]. They show large improvements for a few SPEC CPU2000 programs, and a 7% average improvement on all the programs.

Recently, several researchers have proposed techniques that initiate prefetch requests lower in the memory hierarchy and push the data up the memory hierarchy. The *push* model is different from traditional *pull* model that initiates requests from the the top of the memory hierarchy to the lower levels. Zhang et al. present an initial evaluation of a prefetch scheme for pointer-based structures that prefetches at the memory controller [122]. The technique uses programmer intervention to identify linked structures, and special hardware to determine when to initiate prefetches. Yang and Lebeck evaluate a push method that adds a programmable prefetch engine to each level of the memory hierarchy [119, 120]. They use software support to identify linked structures and to generate programs for the prefetch engine to execute. Hughes and Adve also evaluate a programmable prefetch engine at the memory level [50]. The architecture contains special instructions to identify linked structures and the fields involved in traversals. Content-aware data prefetching is a hardware mechanism that examines objects in the memory subsystem, and attempts to identify pointers that need to be prefetched [29]. The hardware detects values within objects that are likely to be pointers and issues prefetch requests for the pointers.

As we show in this section, many researchers have investigated prefetching techniques. Prior research does not present adequate solutions for software data prefetching in Java programs. Object-oriented languages promote software engineering practices that make

compile-time analysis difficult. We propose whole program analysis to discover prefetch opportunities across method boundaries. Java programmers frequently use arrays as well as linked structures. We develop a unified framework that generates prefetches for both types of data structures.

CHAPTER 3

DATA-FLOW ANALYSIS FOR IDENTIFYING RECURRENCES

Programs often iterate over data structures such as arrays and linked structures. Traditional approaches typically use ad hoc methods to detect these common traversal patterns, and existing approaches focus on either arrays or linked structures, but not both.

In this chapter, we describe our data-flow analysis for identifying recurrences in programs. We describe and implement an analysis that unifies the discovery of loop induction variables and linked structure traversals.

Our analysis, called *recurrence analysis*, contains an intraprocedural component and an interprocedural component. The intraprocedural algorithm finds recurrent variables that occur in loops, and the interprocedural algorithm finds recurrent variables that occur across function calls.

3.1 Loop Induction Variables

An *induction* variable is incremented or decremented by the same value during each loop iteration. An example of an induction variable is the expression `i=i+c` occurring in a loop, as shown in Figure 3.1 (a). During each iteration of the loop, the variable `i` is incremented by a loop invariant value, `c`.

There are several classifications of induction variables [40]. A *linear* induction variable changes by the addition or subtraction of a loop invariant value in every iteration. A *polynomial* induction variable changes by a linear induction variable using addition or subtraction in each iteration. A variable that changes by the addition or subtraction of a polynomial induction variable produces a polynomial of a higher degree. An *exponential* induction

```
int i = 0;
while (i<n) {
  sum += arr[i];
  i=i+1;
}
```

(a) Array Traversal

```
List o = getList();
while (o != null)  {
  sum += o.value();
  o=o.next;
}
```

(b) Linked-Structure Traversal

Figure 3.1. Similarities Between Array and Linked-Structure Traversals

variable changes by the multiplication of a loop invariant expression in each iteration. Our analysis discovers each of these types of induction variables.

3.2 Linked Data Structures

We identify regular traversals of a linked data structure by a *recurrent* update to a pointer variable. A recurrent update is a field assignment of the form `o = o.next` that appears within a loop or recursive call, as shown in Figure 3.1 (b). Each execution of the assignment updates the pointer variable with a new object of the same type, either directly or indirectly through one or more intermediate variables.

3.3 A Unified Analysis

In this section, we present our unified analysis that discovers both loop induction variables and linked structure traversals. An induction variable and linked-structure traversal are examples of general recurrences. Figure 3.1 illustrates the similarities in the code sequences for an array and linked-structure traversal.

The recurrences contain similar patterns. The loop in Figure 3.1 (a) updates variable `i` by incrementing the value by 1. The loop in Figure 3.1 (b) updates object `o` by referencing the next element in the list. We propose a unified recurrence analysis that detects both of these traversal patterns.

In the remainder of this section, we describe our basic intraprocedural algorithm. We follow with extensions to handle object fields and arrays that contain recurrent variables and for indirectly recurrent variables. Then we briefly describe our interprocedural analysis.

3.3.1 Basic intraprocedural analysis

Intraprocedural recurrence analysis discovers the field assignments that are recurrent due to loops. Our analysis is similar to reaching definitions analysis combined with computing definition-use chains for field references [4]. We discover recurrences using a unified forward data-flow analysis.

We define the following sets in our data-flow analysis. Let V be the set of variables in a method, F be the set of object fields, E be the set of binary expressions, FE be the set of object fields and binary expressions, *i.e.*, $FE = F \cup E$, S be the set of statements in the method, and *RS* be the recurrent status that we describe below. The basic analysis information is a set of tuples:

$$R \subseteq \mathcal{P}(V \times FE \times S \times RS)$$

The tuple contains an object field name or binary expression (FE) to improve precision by reducing the number of recurrent variables that the analysis discovers. For example, if a program traverses a doubly linked list in one direction, we improve the precision of the analysis and the effectiveness of prefetching by recording the specific field involved in the traversal.

We use the statement number (S) to handle the case properly when there are two field assignments that occur outside a loop or recursive call. For example, if the sequence `o=o.next; o=o.next` is not in a looping construct, the analysis should not mark `o` as a recurrent variable.

The recurrent status (*RS*) indicates when a program uses a variable to traverse a linked data structure or as an induction variable. Let $rs \in RS = \{\texttt{nr}, \texttt{pr}, \texttt{r}\}$. We order the elements

of *RS* such that `nr` $\prec$ `pr` $\prec$ `r`. The $\prec$ operator forms a lattice for the elements of *RS*. We define the element values as follows:

Not recurrent (`nr`). The initial value that indicates a variable is not updated by the same expression, *i.e.*, it is not involved in a traversal.

Possibly recurrent (`pr`). The first time we process a field reference use or binary expression it is potentially recurrent.

Recurrent (`r`). This value indicates that a variable is an induction variable or involved in a linked-structure traversal.

We informally describe the meaning of the recurrent status element values using a linked-structure example. The first time the analysis processes a loop, an object occurring on the left hand side of a pointer field assignment becomes possibly recurrent, *e.g.*, `t = o.next`. On the second iteration of the analysis, the object on the left hand side becomes recurrent if the base object of the field reference, *i.e.*, `o`, is possibly recurrent. If the base object is not recurrent then `t`'s value remains the same.

We define a function *RA* that maps program statements to the analysis information, $RA : s \rightarrow R$, where $s \in S$. The data-flow equations for recurrence analysis are:

$$\begin{aligned} RA_{in}(s) &= \bigsqcup_{p \in pred(s)} RA_{out}(p) \\ RA_{out}(s) &= (RA_{in}(s) \setminus \text{KILL}_{RA}(s, RA_{in}(s))) \\ &\quad \sqcup \text{GEN}_{RA}(s, RA_{in}(s)) \end{aligned}$$

Given tuples, $t_1=(v_1, fe_1, s_1, rs_1)$ and $t_2=(v_2, fe_2, s_2, rs_2)$, we define the join operation, $t_1 \sqcup t_2$, as follows. If ($v_1=v_2 \wedge fe_1=fe_2 \wedge s_1=s_2$) then $t_1 \sqcup t_2 = (v_1, fe_1, s_1, rs_1 \sqcup rs_2)$. Otherwise, $t_1 \sqcup t_2 = \{t_1, t_2\}$. Given our ordering of the elements $rs \in RS$, $rs \sqcup$ `nr` $= rs$, `pr` $\sqcup$ `pr` = `pr`, and $rs \sqcup$ `r` = `r`.

An iterative data-flow solving algorithm takes d + 3 iterations to solve our data-flow equations, where d is the loop connectiveness[1] of the control flow graph [55].

We define the GEN_{RA} and KILL_{RA} functions as follows:

$$\text{GEN}_{RA}, \text{KILL}_{RA} : S \times R \rightarrow R$$

At the initial statement, init(S), we initialize the function RA_{in} to $\{(v, \emptyset, \emptyset, \texttt{nr}) | v \in V\}$

The interesting program statements for the analysis include field loads and assignments. We describe the details of our GEN and KILL functions for each interesting program statement below. In the following function definitions, f' $\in F$, e' $\in E$, fe' $\in FE$, s' $\in S$, and rs' $\in RS$.

`o = p.f_s` A field assignment at statement `s` may create a recurrent update when it occurs in a loop. Informally, the expression causes a recurrent update when the value assigned to `o` is propagated to `p`, the base object on the right-hand side. The canonical example is `o = o.next` in a loop with no other assignments to `o`. The KILL_{RA} and GEN_{RA} functions for a field assignment are:

$$\begin{aligned} \text{KILL}_{RA}(\text{o=p.f}_\text{s}, R) &= \{(\text{o,f,s,}\texttt{pr}), (\text{o,}\emptyset,\emptyset,\texttt{nr})\} \\ \text{GEN}_{RA}(\text{o=p.f}_\text{s}, R) &= \begin{cases} \{(\text{o,f,s,}\texttt{pr})\} & : \text{if } (\text{p,}\emptyset,\emptyset,\texttt{nr}) \in R \\ \{(\text{o,f,s,}\texttt{r})\} & : \text{if } (\text{p,f,s,}\texttt{pr}) \in R \end{cases} \end{aligned}$$

The *first* time the analysis processes a field assignment, it creates a tuple containing `o` with the `pr` recurrent status. If the field assignment occurs within a loop, then the data-flow analysis does not reach a fixed point due to the change in data-flow information. The analysis repeatedly processes all the statements in the loop until reaching a fixed point.

[1]The loop connectiveness of a control flow graph G, with respect to its depth first spanning tree, is the largest number of back edges found in any cycle-free path of G [55]

The *second* time the analysis processes a field assignment, if there exists a tuple containing p with the recurrent status pr, then there is no intervening assignment to p. In this case, the analysis creates a tuple containing o with the r recurrent status.

`v = j` *op* c_s An integer binary expression at statement s may create an induction variable when it occurs in a loop. Informally, the expression is an induction variable when the value assigned to v is propagated to j, the variable on the right-hand side. The canonical example is j=j+1 in a loop with no other assignments to j. The KILL_{RA} and GEN_{RA} functions for a binary expression are:

$$\text{KILL}_{RA}(\text{v=j op c}_s, R) = \{(\text{v,j op c,s,pr}), (\text{v},\emptyset,\emptyset,\text{nr})\}$$

$$\text{GEN}_{RA}(\text{v=j op c}_s, R) = \begin{cases} \{(\text{v,j op c,s,pr})\} & : \text{if (j},\emptyset,\emptyset,\text{nr}) \in R \\ \{(\text{v,j op c,s,r})\} & : \text{if (j, j op c,s,pr}) \in R \end{cases}$$

The actions for a binary expression are similar to those for a field expression. The *first* time the analysis processes a binary expression, it creates a tuple containing v, the expression j+1, and the pr recurrent status. If the binary expression occurs within a loop, then the data-flow analysis does not reach a fixed point due to the change in data-flow information. The analysis repeatedly processes all the statements in the loop until reaching a fixed point.

The *second* time the analysis processes a binary expression, if there exists a tuple containing j with the recurrent status possibly recurrent, then there is no intervening assignment to j. In this case, that analysis creates a tuple containing v, the expression j+1, with the recurrent induction status.

Binary expressions require an additional GEN and KILL function for the *operands*. Propagating information about the operands enables the analysis to create complex induction variable expressions, such as mutual induction variables. Section 3.3.2 presents an example using mutual induction variables.

$$\text{KILL}_{RA}(\text{v=j op c}_s, R) = \{(\text{v,e' op c,l,rs'}) | \text{l} \neq \text{s}\}$$
$$\text{GEN}_{RA}(\text{v=j op c}_s, R) = \{(\text{v,e' op c,l,rs'}) | \text{ (j,e',l,rs')} \in R \wedge \text{l} \neq \text{s}\}$$

`u=v` A variable assignment expression copies the recurrence information from `v` to `u`. For each tuple containing a variable `v`, we create a new tuple containing `u` with the same field or expression, statement, and recurrent status as `v`. We kill the old information associated with `u`. The KILL_{RA} and GEN_{RA} functions for an assignment are:

$$\text{KILL}_{RA}(\text{u=v}, R) = \{(\text{u,fe',s',rs'})\}$$
$$\text{GEN}_{RA}(\text{u=v}, R) = \{(\text{u,fe',s',rs'}) | \text{ (v,fe',s',rs')} \in R\}$$

`u=expr` Any other assignment to a variable kills the analysis information for `u`. Our analysis sets the recurrent status of any tuple containing `u` to not recurrent (`nr`). The KILL_{RA} and GEN_{RA} functions for all other assignments are:

$$\text{KILL}_{RA}(\text{u=}expr, R) = \{(\text{u,fe',s',rs'})\}$$
$$\text{GEN}_{RA}(\text{u=}expr, R) = \{(\text{u,}\emptyset,\emptyset,\texttt{nr}) | (\text{u,fe',s',rs'}) \in R\}$$

3.3.2 Intraprocedural Examples

In this section, we illustrate the intraprocedural recurrence analysis using a few examples. In each example, we show the sets $RA_{in}(s)$ and $RA_{out}(s)$ for each interesting statement. We show how the information changes during each iteration of the data-flow analysis.

Figure 3.2 shows a simple loop that iterates over a singly linked list. The recurrence analysis detects the linked list traversal that occurs at line 4. In the first iteration, the recurrent status for `t` become possibly recurrent at line 4. At line 5, the analysis copies the

```
1   o = createList();
2   while (o != null) {
3     o.compute();
4     t = o.next;
5     o = t;
6   }
```

stmt	RA	Iteration 1	Iteration 2	Iteration 3
2	in	(o,∅,∅,nr), (t,∅,∅,nr)	(o,next,4,pr), (t,next,4,pr)	(o,next,4,r), (t,next,4,r)
	out	(o,∅,∅,nr), (t,∅,∅,nr)	(o,next,4,pr), (t,next,4,pr)	(o,next,4,r), (t,next,4,r)
4	in	(o,∅,∅,nr), (t,∅,∅,nr)	(o,next,4,pr), (t,next,4,pr)	(o,next,4,r), (t,next,4,r)
	out	(o,∅,∅,nr), **(t,next,4,pr)**	(o,next,4,pr), **(t,next,4,r)**	(o,next,4,r), (t,next,4,r)
5	in	(o,∅,∅,nr), (t,next,4,pr)	(o,next,4,pr), (t,next,4,r)	(o,next,4,r), (t,next,4,r)
	out	**(o,next,4,pr)**, (t,next,4,pr)	**(o,next,4,r)**, (t,next,4,r)	(o,next,4,r), (t,next,4,r)

Figure 3.2. Recurrence Analysis Example: Traversing a List

data-flow information from `t` to `o`. In the second iteration, since the status of `o` is possibly recurrent, the recurrent status of `t` becomes recurrent at line 4. At line 5, the analysis copies the recurrent status from `t` to `o`. In the third iteration, the data-flow information does not change, which means the analysis has reached a fixed point and is done. At the end of the loop, both `o` and `t` are recurrent due to the `next` field at line 4.

Figure 3.3 shows a loop that iterates over an array. The recurrence analysis detects the array traversal that occurs at line 5. The example is analogous to the linked list example in Figure 3.2. Instead of propagating the `next` field, the analysis propagates the expression `i+1`. In the first iteration, the analysis computes that both `i` and `j` are possibly recurrent due to the expression `i+1`. In the second iteration, the analysis computes that `i` and `j` are recurrent because the program does not redefine the variables using different values between loop iterations.

Figure 3.4 illustrates an example that kills the data-flow information during each loop iteration. Because the intraprocedural analysis does not know the recurrent status information from the call to `newList()`, the analysis kills the recurrence information for `o` at line 6.

```
1   int sum = 0;
2   int i = 0;
3   while (i < n) {
4     sum += A[i];
5     j = i + 1;
6     i = j;
7   }
```

stmt	RA	Iteration 1	Iteration 2	Iteration 3
3	in	(i,∅,∅,nr), (j,∅,∅,nr)	(i,i+1,5,pr), (j,i+1,5,pr)	(i,i+1,5,r), (j,i+1,5,r)
	out	(i,∅,∅,nr), (j,∅,∅,nr)	(i,i+1,5,pr), (j,i+1,5,pr)	(i,i+1,5,r), (j,i+1,5,r)
5	in	(i,∅,∅,nr), (j,∅,∅,nr)	(i,i+1,5,pr), (j,i+1,5,pr)	(i,i+1,5,r), (j,i+1,5,r)
	out	(i,∅,∅,nr), **(j,i+1,5,pr)**	(i,i+1,5,pr), **(j,i+1,5,r)**	(i,i+1,5,r), (j,i+1,5,r)
6	in	(i,∅,∅,nr), (j,i+1,5,pr)	(i,i+1,5,pr), (j,i+1,5,r)	(i,i+1,5,r), (j,i+1,5,r)
	out	**(i,i+1,5,pr)**, (j,i+1,5,pr)	**(i,i+1,5,r)**, (j,i+1,5,r)	(i,i+1,5,r), (j,i+1,5,r)

Figure 3.3. Recurrence Analysis Example: Traversing an Array

```
1   o = createList();
2   while (o != null) {
3     o.compute();
4     o = o.next;
5     // perform some computation on o
6     o = newList();
7   }
```

stmt	RA	Iteration 1	Iteration 2
2	in	(o,∅,∅,nr)	(o,∅,∅,nr)
	out	(o,∅,∅,nr)	(o,∅,∅,nr)
4	in	(o,∅,∅,nr)	(o,∅,∅,nr)
	out	**(o,next,4,pr)**	**(o,next,4,pr)**
6	in	(o,next,4,pr)	(o,next,4,pr)
	out	**(o,∅,∅,nr)**	**(o,∅,∅,nr)**

Figure 3.4. Recurrence Analysis Example: Kill Data-Flow Information

```
1   o = createList();
2   while (o != null) {
3     o.compute();
4     if (o.someCondition())
5       o = o.next;
6     if (o.someCondition())
7       o = o.next;
8   }
```

stmt	RA	Iteration 1	Iteration 2	Iteration 3
2	in	(o,∅,∅,nr)	(o,next,5,pr), (o,next,7,pr)	(o,next,5,r), (o,next,7,r)
	out	(o,∅,∅,nr)	(o,next,5,pr), (o,next,7,pr)	(o,next,5,r), (o,next,7,r)
5	in	(o,∅,∅,nr)	(o,next,5,pr), (o,next,7,pr)	(o,next,5,r), (o,next,7,r)
	out	**(o,next,5,pr)**	**(o,next,5,r)**, (o,next,7,pr)	(o,next,5,r), (o,next,7,r)
7	in	(o,next,5,pr)	(o,next,5,r), (o,next,7,pr)	(o,next,5,r), (o,next,7,r)
	out	(o,next,5,pr), **(o,next,7,pr)**	(o,next,5,r), **(o,next,7,r)**	(o,next,5,r), (o,next,7,r)

Figure 3.5. Recurrence Analysis Example: Traversing a List Conditionally

In Figure 3.5, we illustrate the effect of conditional statements and multiple field references on the data-flow analysis. In the first iteration, `o` becomes possibly recurrent at line 5 and line 7 due to the access of the `next` field. In the second iteration, `o` becomes recurrent at lines 5 and 7. On the third iteration, nothing changes. The data-flow analysis merges the recurrence information at line 6 and line 8. During the second iteration, the merge at line 6 combines `(o,next,5,pr)` and `(o,next,5,r)` to produce `(o,next,5,r)`.

Figure 3.6 illustrates the how the data-flow analysis processes mutual induction variables. The analysis creates complex expressions to handle induction variables. In the example, `j`'s value depends upon `i` and `i`'s value depends upon `j`. Since the loop increments each variable by one, both variables increase by two in each iteration. Our recurrence analysis detects the mutual induction variables, and correctly computes the increment value. The interesting points occur at lines 5 and 6. For example, in the first iteration, at line 6, the analysis creates two new tuples. The first tuple, `(i,j+1,6,pr)`, indicates that variable `i` is assigned the value `j+1`. Since the analysis information contains a tuple for `j`, we create a second tuple, `(i,i+2,5,pr)`, which builds a complex expression using the expression from `j`'s tuple. Thus at line 6, the analysis represents `i` as `j + 1` and `i + 1`

```
1   int sum = 0;
2   int i = 0; int j = 0;
3   while (i < n) {
4     sum += A[i];
5     j = i + 1;
6     i = j + 1;
7   }
```

stmt	RA	Iteration 1	Iteration 2	Iteration 3
3	in	(i,∅,∅,nr), (j,∅,∅,nr)	(i,j+1,6,pr), (j,i+1,5,pr) (i,i+2,5,pr)	(i,j+1,6,r), (j,i+1,5,r) (i,i+2,5,r), (j,j+2,6,pr)
	out	(i,∅,∅,nr), (j,∅,∅,nr)	(i,j+1,6,pr), (j,i+1,5,pr) (i,i+2,5,pr)	(i,j+1,6,r), (j,i+1,5,r) (i,i+2,5,r), (j,j+2,6,pr)
5	in	(i,∅,∅,nr), (j,∅,∅,nr)	(i,j+1,6,pr), (j,i+1,5,pr) (i,i+2,5,pr)	(i,j+1,6,r), (j,i+1,5,r) (i,i+2,5,r), (j,j+2,6,pr)
	out	(i,∅,∅,nr), **(j,i+1,5,pr)**	(i,j+1,6,pr), **(j,i+1,5,r)** (i,i+2,5,pr),**(j,j+2,6,pr)**	(i,j+1,6,r), (j,i+1,5,r) (i,i+2,5,r), **(j,j+2,6,r)**
6	in	(i,∅,∅,nr), (j,i+1,5,pr)	(i,j+1,6,pr), (j,i+1,5,r) (i,i+2,5,pr), (j,j+2,6,pr)	(i,j+1,6,r), (j,i+1,5,r) (i,i+2,5,r), (j,j+2,6,r)
	out	**(i,j+1,6,pr)**, (j,i+1,5,pr) **(i,i+2,5,pr)**	**(i,j+1,6,r)**, (j,i+1,5,r) **(i,i+2,5,r)**, (j,j+2,6,pr)	(i,j+1,6,r), (j,i+1,5,r) (i,i+2,5,r), (j,j+2,6,r)

Figure 3.6. Recurrence Analysis Example: Complex Induction Variable

`+ 1`. After reaching a fixed point, the analysis indicates that `i` and `j` are recurrent and they increase by two in each iteration.

3.3.3 Interprocedural Algorithm

The interprocedural analysis finds recurrences that are due to recursive method calls and that cross method boundaries. The interprocedural analysis propagates data-flow information into the parameters of a method and from the return value.

The algorithm is a bidirectional context-sensitive traversal of the call graph. A context-sensitive algorithm enables the analysis phase to determine the fields used in recurrent object updates across recursive function calls. A context-insensitive algorithm cannot track the recurrence information from multiple call sites because the compiler analyzes each method only once. For example, in a program that traverses a binary tree using recursion, a context-sensitive analysis determines that the left and right children are recurrent fields. A

```
public ipRec() {
  int s = data;
  if (next != null) {
    s += next.ipRec();
  }
  return s;
}
```

(a) Recursion

```
public void ipIter(List l) {
  Enumeration e = l.elements();
  while (e.hasMoreElements()) {
    Node n = (Node)e.nextElement();
    n.calc();
  }
}
```

(b) Iteration with Encapsulation

Figure 3.7. Examples Showing Need for IP analysis

context-insensitive analysis analyzes the recursive method only once and will not determine that both children are recurrent fields.

A context-sensitive interprocedural algorithm can be quite expensive because each function may be analyzed multiple times. Our interprocedural analysis analyzes each function reached at each call site at most three times. Each call site may invoke multiple functions due to polymorphic function calls. In practice, a compiler should perform analysis to reduce the number of potential methods reachable at each call site. For example, our compiler uses 0-CFA interprocedural class analysis to reduce the call graph size [95].

There are two distinct classes of interprocedural recurrences. The first is due to a recursive method call. The second is due to iteration combined with encapsulation. We show an example using recursion in Figure 3.7 (a), and an example using iteration with encapsulation in Figure 3.7 (b). Example (a) traverses a linked list by recursively calling itself with the next element in the list. Example (b) uses iteration, but calls another method to obtain the next element in the list. The recurrent field access is hidden in the call to `nextElement`.

A method definition has the form: $\mathtt{r = m(p_0,...,p_n)}$, where $\mathtt{p}_i$ is a formal parameter, and $\mathtt{r}$ is the return value. At a call site, the analysis creates a new set of tuples, R_m, for the callee. The analysis processes each argument, $\mathtt{a}_0$,...,$\mathtt{a}_n$, as an assignment of the recurrence information from the argument to the parameter, $\mathtt{p}_i = \mathtt{a}_i$.

```
1  method getNext() {
2    return next;
3  }
4
5  l = l.getNext();
6  l = l.getNext();
```

(a) Two Contexts - Not Recurrent

```
1  l = getList();
2  while (l != null) {
3    l.compute();
4    l = l.getNext();
5  }
```

(b) One Context - Recurrent

Figure 3.8. Using Calling Context Information

At a call site, the analysis also adds the recurrent field information to R_m for each of the argument's fields, $\mathtt{a}_i\mathtt{.f}$. After initializing R_m, we analyze the callee method with R_m using the intraprocedural analysis. Recursive calls cause the analysis to iterate until the data-flow information for each parameter reaches a fixed point.

We process a function return as an assignment of `r` to the value on the left hand side of the method call by copying the analysis information from R_m to R_c. The analysis uses the appropriate GEN and KILL function, which depends on whether the left hand side expression is a simple object, field reference, or array reference.

In our intraprocedural analysis, the data-flow information contains the statement number where the expression occurs. We augment the statement number with context information to process the interprocedural information correctly. The context information distinguishes between the recurrence information in different calling contexts. When the analysis processes a return statement, we prepend context information to the statement number. We illustrate why the context information is necessary in Figure 3.8.

In Figure 3.8 (a), the analysis should not indicate that `l` is recurrent. Without context information, the analysis will compute that `l` is recurrent. At line 5, the analysis assigns the result of the call to `getNext()` to `l`. The analysis information computed at line 2 from the first call site is `(l,next,2,pr)`. The analysis creates the tuple `(l,next,2,pr)` at line 5 because of the assignment. The second call site causes the analysis to create the

tuple `(l,next,2,r)` at line 2. Then the analysis create the tuple `(l,next,2,r)` at line 6. At the end of processing the code sequence, the analysis indicates that `l` is recurrent.

We avoid spurious recurrences by adding context information to the data-flow information. Upon the return from `getNext()` at line 5, the analysis prepends the statement number with context information. We use the call site number as the context information. In this example, the analysis creates the tuple `(l,next,1.2,pr)` after the first call site. When the analysis processes the second call site, it filters the data-flow information and removes tuples that contain invalid contexts. In this example, the analysis removes the tuple `(l,next,1.2,pr)` from the callee's information because the contexts are different. After processing the second call site, the data-flow information includes the tuples `(l,next,1.2,pr)` and `(l,next,2.2,pr)`. If this second sequence occurs within a loop, then the analysis processes the statements again, and correctly indicates that the two call sites cause recurrences.

3.3.4 Interprocedural Example

In this section, we illustrate an example of interprocedural recurrence analysis. Figure 3.9 shows the steps of the interprocedural recurrence analysis using an example with recursion. The method recursively calls itself with the `left` and `right` children. The method contains an implicit parameter, `this`, that is the current node of the tree.

The tables in Figure 3.9 show the data-flow information that the analysis computes during each visit of the method. The method contains two call sites; call site 1 at line 3, and call site 2 at line 4. The subscript in the table header indicates the call site number. The second and third visits in the analysis are for call site 1, and the fourth and fifth visits are for call site 2.

On the first visit, the analysis computes that the `left` and `right` fields become possibly recurrent. The second visit occurs at line 3. The analysis prepends the call site id to the

```
1   int treeAdd() { // this is an implicit parameter
2     int total = value;
3     if (left != null) total += left.treeAdd();
4     if (right != null) total += right.treeAdd();
5     return total;
6   }
```

stmt	RA	First Visit	Second Visit$_1$	Third Visit$_1$
1	in	(this,∅,∅,nr)	**(this,left,1.3,pr)**	(this,left,1.3,r)
	out	(this,∅,∅,nr)	(this,left,1.3,pr)	(this,left,1.3,r)
3	in	(this,∅,∅,nr)	(this,left,1.3,pr)	(this,left,1.3,r)
	out	**(this,left,3,pr)**	**this,left,1.3,r)**	(this,left,1.3,r)
4	in	(this,left,3,,pr)	(this,left,1.3,r)	(this,left,1.3,r)
	out	(this,left,3,pr)	(this,left,1.3,r)	(this,left,1.3,r)
		(this,right,4,pr)	(this,right,4,pr)	(this,right,4,pr)

stmt	RA	Fourth Visit$_2$	Fifth Visit$_2$
1	in	**(this,left,2.3,pr)**, **(this,right,2.4,pr)**	(this,left,2.3,r), (this,right,2.4,r)
	out	(this,left,2.3,pr), (this,right,2.4,pr)	(this,left,2.3,r), (this,right,2.4,r)
3	in	**(this,left,2.3,r)**, (this,right,2.4,pr)	(this,left,2.3,r), (this,right,2.4,r)
	out	(this,left,2.3,r), (this,right,2.4,pr)	(this,left,2.3,r), (this,right,2.4,r)
4	in	(this,left,2.3,r), (this,right,2.4,r)	(this,left,2.3,r), (this,right,2.4,r)
	out	(this,left,2.3,r), **(this,right,2.4,r)**	(this,left,2.3,r), (this,right,2.4,r)

Figure 3.9. IP Recurrence Analysis Example: Recursion

statement number. On the second visit, the analysis computes that the `left` field becomes `recurrent` at line 3. On the third visit, the analysis information does not change.

The fourth visit occurs at line 4 for the `right` field. At this point, the analysis propagates the information for both the `right` and `left` fields. The analysis prepends the call site number, 2, to the statement number. At line 3, the analysis computes that the `left` field becomes recurrent. When the analysis attempts to analyze the method because of the call at line 3, it determines that it has already processed this same input on the third visit, and does not need to process it again. When trying to compute if the analysis has already seen an input, the comparison ignores the call site number. Finally, at line 4, the analysis computes that `right` becomes recurrent.

On the fifth visit, both the `left` and `right` children are recurrent, and the analysis information does not change. When the analysis processes the call sites again, it determines that is has already seen the inputs and the recursive calls finish.

3.3.5 Object Fields and Arrays

In the this section, we describe extensions to our basic analysis that improve the precision of the results. In the basic analysis, we assume that the left hand side expression is a simple variable. We improve the analysis by tracking the recurrence information of variables assigned to object fields and array elements also. For example, in the code sequence in Figure 3.10, the analysis of Section 3.3.1 does not indicate that object `o` is recurrent because the analysis does not propagate the recurrence information to `temp.f`. This sequence occurs in Java programs that use the `Enumeration` class to traverse linked lists when inlining is enabled.

To improve the analysis, we extend the data-flow tuple to include field references and arrays. Let VFA be the set of variables, field references, and arrays. We define:

$$R' \subseteq \mathcal{P}(VFA \times FE \times S \times RS)$$

```
while (temp.f != null) {
  o = temp.f;
  o.compute();
  t = o.next;
  temp.f = t;
}
```

Figure 3.10. Assigning Recurrence Information to a Field

We also define a new analysis function, $RA' : s \rightarrow R'$, where $s \in S$.

For object fields, we associate the analysis information with the field name, and the analysis ignores the specific base object instance. We prepend the field name with its class name to avoid ambiguity between fields from different classes. We can potentially improve the precision by tracking the base object of the field reference, but that increases the analysis complexity cost. We treat arrays as monolithic objects in our analysis, *i.e.*, as an assignment or use of the whole array.

We define the $\text{GEN}_{RA'}$ and $\text{KILL}_{RA'}$ functions to include the same definitions as GEN_{RA} and KILL_{RA} with the following extensions:

`p.f=o, a[j]=o` Create data-flow information for a field or array reference. The $\text{GEN}_{RA'}$ and $\text{KILL}_{RA'}$ functions are similar to a pointer variable assignment.

$$\begin{aligned}
\text{KILL}_{RA'}(\text{p.f=o}, R') &= \{\text{(f,fe',s',rs')}\} \\
\text{GEN}_{RA'}(\text{p.f=o}, R') &= \{\text{(f,fe',s',rs')} | \text{(o,fe',s',rs')} \in R'\} \\
\text{KILL}_{RA'}(\text{a[j]=o}, R') &= \{\text{(a,fe',s',rs')}\} \\
\text{GEN}_{RA'}(\text{a[j]=o}, R') &= \{\text{(a,fe',s',rs')} | \text{(o,fe',s',rs')} \in R'\}
\end{aligned}$$

`o=p.f, o=a[j]` For any tuple containing `p.f` or `a`, we create a new tuple containing `o` which includes the field, statement, and recurrent status. The $\text{GEN}_{RA'}$ and $\text{KILL}_{RA'}$ functions are:

$$
\begin{aligned}
\text{KILL}_{RA'}(\text{o=p.f}, R') &= \{\text{(o,fe',s',rs')}\} \\
\text{GEN}_{RA'}(\text{o=p.f}, R') &= \{\text{(o,fe',s',rs')} | \text{(f,fe',s',rs')} \in R'\} \\
\text{KILL}_{RA'}(\text{o=a[j]}, R') &= \{\text{(o,fe',s',rs')}\} \\
\text{GEN}_{RA'}(\text{o=a[j]}, R') &= \{\text{(o,fe',s',rs')} | \text{(a,fe',s',rs')} \in R'\}
\end{aligned}
$$

`p.f=`*`expr`*`, a[j]=`*`expr`* Any other assignment to a field or array kills the data-flow information for `p.f` or `a`. The $\text{GEN}_{RA'}$ and $\text{KILL}_{RA'}$ functions are:

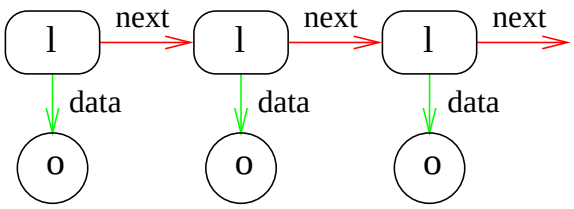
$$
\begin{aligned}
\text{KILL}_{RA'}(\text{p.f=}expr, R') &= \{\text{(f,fe',s',rs')}\} \\
\text{GEN}_{RA'}(\text{p.f=}expr, R') &= \{\text{(f,}\emptyset,\emptyset,\texttt{nr)} | \text{(o,fe',s',rs')} \in R'\} \\
\text{KILL}_{RA'}(\text{a[j]=}expr, R') &= \{\text{(a,fe',s',rs')}\} \\
\text{GEN}_{RA'}(\text{a[j]=}expr, R') &= \{\text{(a,}\emptyset,\emptyset,\texttt{nr)} | \text{(o,fe',s',rs')} \in R'\}
\end{aligned}
$$

3.3.6 Indirect Recurrent Variables

An *indirect* recurrent variable is an unshared object that is referenced by a recurrent variable, but is not recurrent itself. An object is *shared* if it may be referenced by multiple objects. In contrast, an object is *unshared* if it may be referenced by at most one other object. An example of an indirect recurrent variable occurs in a traversal of a generic linked list, where the data elements are separate objects from the list nodes. In Figure 3.11, `l` is a recurrent variable for a linked list traversal, and both `l` and `o` are unshared. In this example, `o` is also an indirect recurrent variable because it is unshared and it is referenced by a recurrent variable.

Both `l` and `o` are candidates for prefetching because each iteration of the loop accesses a new list node and a new data element. We do not want to prefetch shared objects because each iteration may access the same data element, which results in wasteful prefetches.

We must first classify objects as shared or unshared to compute the set of indirect recurrent variables. We use an approximation because statically classifying dynamically al-




```
while (l != null) {
  o = l.data;
  o.compute();
  l = l.next;
}
```

Figure 3.11. Example of Indirect Recurrent Variable

located objects exactly is not feasible. Our approximation classifies *class fields* as shared or unshared. We describe the shared object analysis in the following section.

3.4 Cooperating Analyses

We develop additional analyses to assist the recurrence analysis and prefetching optimizations. In this section, we describe the analyses and define the data-flow solutions. The cooperating analyses are:

Shared object analysis Compute which objects are referenced by at most one other object. We use this analysis to create jump-pointers.

Array size analysis Compute the size of all arrays, if possible. We use this analysis to generate prefetches for elements in arrays that contain recurrent objects.

3.4.1 Shared Object Analysis

Shared object analysis determines if multiple object instances may ever contain a field reference to the same object. We illustrate shared object analysis using Figure 3.12. Classes `A` and `C` contain a single field `f` that references an object of class `B`. In each instance of `A`, field `f` contains a reference to distinct objects of type `B`. In contrast, in each instance of `C`, field `f` contains a reference to the same object of type `B`.

Determining which fields reference a single object or multiple objects enables our recurrence analysis to detect an important type of linked structure in which data is not stored in the linked objects, but is a separate object that is referenced by the linked object.

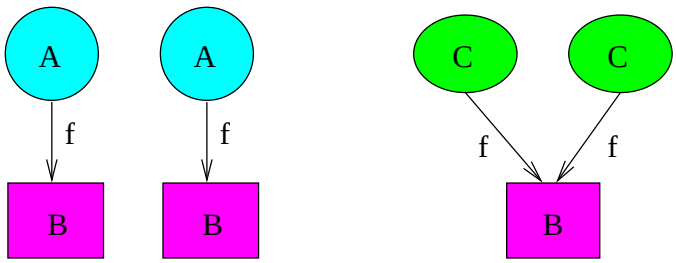


Figure 3.12. Object Sharing

We implement an interprocedural context-insensitive data-flow analysis to discover shared objects. Our analysis is similar to Aldrich et al.'s unshared field analysis for eliminating unnecessary synchronization [5], and Dolby's analysis for finding inlinable objects [35]. The main difference between our analysis and the prior approaches is the precision of the analysis. Dolby's analysis requires more precision since he uses the analysis to inline unshared objects. Both prior algorithms are context-sensitive, whereas ours is context-insensitive. It is possible to make our analysis more precise, but our application of the shared object analysis does not require more precision.

The analysis begins by assuming that all fields are unshared. The analysis maintains a mapping between program variables and field names. When processing an assignment of a variable to a field, the analysis creates an association between the variable and the field name. The analysis removes other existing associations between the field name and any different variable. If the variable appears on the right-hand side of multiple field store expressions, then the analysis associates the variable with each field name. When processing a field store, if there already exists an association between the variable and the field, then the field is shared.

The analysis also propagates the field information at assignments and field loads. At a function call, the analysis assigns the field information associated with each argument to each formal parameter. After processing the function call, the caller updates the analysis information with changes made in the callee by assigning the field information associated

with each formal parameter to each argument. The analysis computes aliases among the objects to determine if the object has previously been assigned to a field. All fields are identified as either shared or unshared at the end of this analysis.

The intraprocedural portion of the shared object analysis is a forward data-flow analysis problem. We define the following sets for our analysis. Let V be the set of variables in a method, F be the set of object fields, SF be the set of shared fields, and S be the set of statements. The basic analysis information is a tuple consisting of a mapping between variables and fields, and the set of fields that are shared:

$$SH \subseteq \mathcal{P}(V \times F \times SF)$$

We define an analysis function SA that maps program statements to the analysis information, $SA : s \rightarrow SH$, where $s \in S$. The data-flow equations for shared object analysis are:

$$\begin{aligned} SA_{in}(s) &= \bigsqcup_{p \in pred(s)} SA_{out}(p) \\ SA_{out}(s) &= (SA_{in}(s) \setminus \text{KILL}_{SA}(s, SA_{in}(s))) \\ &\quad \sqcup \text{GEN}_{SA}(s, SA_{in}(s)) \end{aligned}$$

At the initial statement, init(S), we initialize the function SA_{in} to $\{(v, \emptyset, \emptyset) | v \in V\}$. We define the GEN_{SA} and KILL_{SA} functions as GEN_{SA},$\text{KILL}_{SA} : S \times SH \rightarrow SH$. The statements which effect the analysis include assignments, field stores, and field loads. We describe the details of the GEN and KILL functions for each interesting program statement below. In the following definitions, v' $\in V$ and f' $\in F$.

`v = o.f` At a field load, we create a mapping between the variable on the left hand side and the field. A field load does not change the fields in the shared set.

The KILL_{SA} and GEN_{SA} functions for a field assignment are:

$$\begin{aligned} \text{KILL}_{SA}(\text{v=o.f}, SH) &= \{(\text{v,f'}, SF)\} \\ \text{GEN}_{SA}(\text{v=o.f}, SH) &= \{(\text{v,f}, SF)\} \end{aligned}$$

`v = u` For an assignment, we copy the data-flow information from `u` to `v`. For each tuple containing a variable `u`, we create a new tuple containing `v` with the same information as `u`. We kill the old information associated with `v`.

The KILL_{SA} and GEN_{SA} functions for an assignment are:

$$\begin{aligned} \text{KILL}_{SA}(\text{v=u}, SH) &= \{(\text{v,f}, SF)\} \\ \text{GEN}_{SA}(\text{v=u}, SH) &= \{(\text{v,f}, SF) \mid (\text{u,f}, SF) \in SH\} \end{aligned}$$

`v = expr` Any other assignment to a variable kills the variable/field mapping information for `v`. The assignment does not affect the shared field set.

$$\begin{aligned} \text{KILL}_{SA}(\text{v=}expr, SH) &= \{(\text{v,f}, SF)\} \\ \text{GEN}_{SA}(\text{o=}expr, SH) &= \{(\text{v,}\emptyset, SF)\} \end{aligned}$$

`o.f = v` A field store may create a shared field. A field is shared if the object on the right-hand side has been assigned to this field previously.

$$\text{KILL}_{SA}(\text{o.f=v}, SH) = \{(\text{v'},\text{f}, SF)\}$$

$$\text{GEN}_{SA}(\text{o.f=v}, SH) \;=\; \begin{cases} (\text{v,f}, SF), (\text{o'},\text{f}, SF) & : \text{ if aliases(o',o)} \\ (\text{v,f}, SF), (\text{v'},\text{f'}, SF \cup \text{f}) & : \text{ if } (\text{v,f}, SF) \in SH \end{cases}$$

The GEN function uses the aliases() function that returns true if the arguments are aliases. In our compiler implementation, we compute aliases using an existing value numbering algorithm.

The interprocedural analysis is bidirectional and context-insensitive. At a call site to method `m`, the analysis creates a new set of tuples, C_m, for the callee. The analysis processes each argument, a_0,...,a_n, as an assignment of the shared object information from the argument to the formal parameter, $f_i = a_i$. Upon return from a method, the analysis must propagate data-flow information from the formal parameters to the arguments, *i.e.*, $a_i = f_i$.

3.4.2 Array Size Analysis

The compiler must know the array sizes to generate prefetches when an array contains recurrent object references. Unfortunately, Java creates all arrays dynamically, and the array size is not known by simply examining the array declaration. We develop an analysis to determine the size of arrays by examining the statements in a program rather than just looking at the declarations.

Programmers use arrays to represent linked structures that may contain multiple recursive connections. Figure 3.13 shows the class definition and use of a tree with eight children. The `count` method recursively calls itself with each of the children. To generate the correct number of prefetches, we need to know the array size. Our array size analysis computes that the array size is a compile-time constant and the value is eight.

We develop a new data-flow analysis to compute the array sizes. Our analysis must be run interprocedurally to obtain meaningful results, but we divide the analysis into intraprocedural and interprocedural components. We first describe the intraprocedural analysis,

```
class OctTree {
  int       data;
  OctTree[] children;

  OctTree(int d) {
    data = d;
    children = new OctTree[8];
  }

  int count() {
    int c = data;
    for (int i=0; i<children.length(); i++) {
      if (children[i] != NULL) {
        c += children[i].count();
      }
    }
    return c;
  }
}
```

Figure 3.13. Using an Array to Represent an Oct-tree

and present the extensions for dealing with method calls. Our analysis is closely related to constant propagation.

The intraprocedural analysis determines the array sizes by analyzing the allocation expressions and propagates the size information to field stores whose type is an array. We define a forward data-flow analysis for the intraprocedural problem. The lattice in the array analysis is very similar to the constant propagation lattice.

We define the following sets for the array size analysis. Let VF be the set of variables and class fields, Z be the set of integers, and T be the set of array types. We need to extend Z to include the top element of the lattice which indicates the size of the array is unknown. We define the set $Z' = Z \cup \{\top\}$. Let $s \in S$ be the set of statements in the CFG.

The basic analysis information is:

$$C \subseteq \mathcal{P}(V \times Z' \times T)$$

We define a function AS that maps program statements to the analysis information, $AS : s \to C$, where $s \in S$.

The data-flow equations for array size analysis are:

$$\begin{aligned} AS_{in}(s) &= \bigsqcup_{p \in pred(s)} AS_{out}(p) \\ AS_{out}(s) &= (AS_{in}(s) \setminus \text{KILL}_{AS}(s, AS_{in}(s))) \\ &\quad \sqcup \text{GEN}_{AS}(s, AS_{in}(s)) \end{aligned}$$

We define the GEN_{AS} and KILL_{AS} functions as GEN_{AS},$\text{KILL}_{AS} : S \times C \to C$.

The statements that affect the analysis include array creation statements, assignments, and field stores. We describe the details of the GEN and KILL functions for each interesting program statement below. In the following function definitions, c $\in Z'$, and t $\in T$.

`v = new_array(n,T)` An array creation statement. The function `new_array` creates an array of size `n` of type `T`. If the size of the array is a compile-time constant, we

propagate the size information to the LHS. Otherwise, we indicate that the size of the array is unknown.

The KILL_{AS} and GEN_{AS} functions for an array creation statement are:

$$\begin{aligned} \text{KILL}_{AS}(\text{v=new\_array(n,T)}, C) &= \{(\text{v,c,T})\} \\ \text{GEN}_{AS}(\text{v=new\_arrray(n,T)}, C) &= \begin{cases} \{(\text{v,n,T})\} & : \text{if n} \in \text{Z} \\ \{(\text{v,}\top\text{,T})\} & : \text{otherwise} \end{cases} \end{aligned}$$

`v = u` For an assignment, we copy the array size information from `u` to `v`. For each tuple containing a variable `u`, we create a new tuple containing `v` with the same size and type information as `u`. We kill the old information associated with `v`.

The KILL_{AS} and GEN_{AS} functions for an assignment are:

$$\begin{aligned} \text{KILL}_{AS}(\text{v=u}, C) &= \{(\text{v,c,t})\} \\ \text{GEN}_{AS}(\text{v=u}, C) &= \{(\text{v,c,t}) \mid (\text{u,c,t}) \in C\} \end{aligned}$$

`o.f = v` Create data-flow information for an assignment of an array object to an object field. When `f` is an array reference, propagate the array size information from `v` to `f`. If there is no tuple with `f`, then create a new tuple containing `f` with the same size and type information as `v`. If there is another tuple with `f`, then the array size and type must be the same, otherwise we create a tuple with an undefined size and type.

The KILL_{AS} and GEN_{AS} for a field definition are:

$$\begin{aligned} \text{KILL}_{AS}(\text{o.f=v}, C) &= \{(\text{f,c,t})\} \\ \text{GEN}_{AS}(\text{o.f=v}, C) &= \begin{cases} (\text{f,}\top\text{,}\emptyset) & : \text{if } (\text{v,c,t}) \in C \wedge (\text{f,c',t}) \in C \wedge \text{c'} \neq \text{c} \\ (\text{f,c,t}) & : \text{if } (\text{v,c,t}) \in C \end{cases} \end{aligned}$$

The interprocedural analysis is bidirectional. The analysis propagates the data-flow information from the method arguments to the formal parameters, and propagates the method return value from the callee to the caller. At a call site to method `m`, the analysis creates a new set of tuples, C_m, for the callee. The analysis processes each argument, $\texttt{a}_0$,...,$\texttt{a}_n$, as an assignment of the array size information from each arguments to each formal parameter, $\texttt{f}_i = \texttt{a}_i$. The analysis processes a function return as an assignment of `r` to the value on the left hand side of the method call by copying the analysis information from C_m to C_c. The analysis uses the appropriate GEN and KILL function, which depends on whether the left hand side expression is a field store or variable.

After analyzing the methods in Figure 3.13, the analysis contains the tuple, (`children`, 8, `OctTree`) which indicates that the `children` array contains 8 elements.

3.5 Chapter Summary

In this chapter, we describe a new data-flow analysis for identifying recurrences in programs. Prior approaches are typically ad hoc, require explicit use-def chains, or focus on either arrays or linked structures. Our recurrence analysis recognizes induction variables and linked structure traversals. We show that the two common traversal idioms are closely related, which we exploit to create a unified analysis. The analysis contains an intraprocedural component to discover recurrences due to loops. The intraprocedural analysis efficient enough to implement in a just-in-time (JIT) compiler that contains a data-flow analysis framework. The analysis is also interprocedural, which enables the compiler to discover recurrences that are due to recursion or that occur across method calls. Since the interprocedural analysis is context-sensitive, it is not suitable for a JIT compiler. We need to investigate techniques for reducing the cost of the interprocedural analysis. Our analysis is able to propagate data-flow information that is assigned to object fields and array elements.

We present two additional analyses, shared object analysis and array size analysis, that assist the recurrence analysis and prefetch optimizations. Shared object analysis statically computes which objects are referenced by at most one other object. We use the shared object analysis to detect indirect recurrent objects, which are objects that are not recurrent, but are referenced by a recurrent object via an unshared field. The array size analysis computes the size of arrays when possible. Java creates all arrays dynamically, and the array size is not known by examining the declaration only. The analysis performs interprocedural constant propagation to compute array sizes. Since both shared object analysis and array analysis are interprocedural, it is uncertain whether they are suitable for JIT compilers. However, the algorithms are context-insensitive, which reduces the complexity cost.

In the next chapter, we show how to use the recurrence analysis to identify prefetch opportunities in arrays and linked structures. Computing recurrences is also applicable to other domains besides prefetching, such as data layout and code optimizations on linked structures.

CHAPTER 4

PREFETCH TECHNIQUES

Effective software data prefetching requires methods to determine *what* to prefetch and *when* to generate a prefetch instruction. The previous chapter presents a new technique for identifying what to prefetch. In this chapter, we discuss several algorithms that determine when to generate a prefetch.

The Java core library contains classes that use arrays and linked structures. Through using of these core classes, Java programs frequently access both arrays and linked structures that result in cache misses. To improve the memory performance of Java programs, we need to use algorithms that are able to prefetch both types of data structures.

In this chapter, we describe the implementation of an array prefetch technique and three algorithms for prefetching linked structures. The linked-structure algorithms are greedy prefetching, jump-pointer prefetching, and stride prefetching.

In the next section, we describe our novel array prefetch algorithm. Section 4.2 describes greedy prefetching. In Section 4.3, we present a compiler implementation of jump-pointer prefetching. We describe stride prefetching in Section 4.4. Finally, Section 4.5 discusses the compiler implementation of the recurrence analysis and prefetch algorithms. We show that the prefetch algorithms are quite similar, and we are able to share a large amount of code among the prefetch implementations.

4.1 Array Prefetching

In this section, we describe our algorithm to insert array prefetch instructions. The prefetch algorithm must identify an array access pattern and insert a prefetch for an element

```
for (int i=0; i<n; i++) {
  prefetch(&arr[i+d]);
  sum += arr[i];
}
```

Figure 4.1. Simple Index Expression

that will be accessed in the future. We illustrate a simple array prefetching example in Figure 4.1. During each iteration, the program references the `i`th element, and we prefetch element `i+d`, where `d` is the prefetch distance. Prefetching is most effective when the prefetch distance value, `d`, is large enough to move the `i+d`th array element into the L1 cache before `d` iterations of the loop.

We first describe Mowry et al.'s array prefetch algorithm, which is the most common algorithm that compilers use in practice. Mowry et al. developed and evaluated the algorithm on in-order uniprocessor architectures and multiprocessors. Our insight is that most modern processors are out-of-order architectures and often do not fully utilize the functional units in the processor. We describe a simpler prefetch algorithm that does not require locality analysis or loop transformations. We believe that our algorithm is suitable for a just-in-time (JIT) compiler because it requires a data-flow framework only. Although our evaluation uses an ahead-of-time compiler, several existing JIT compilers support data-flow methods including HotSpot and the Jikes RVM (*i.e.*, Jalapeño) [86, 7].

4.1.1 Mowry's Prefetch Algorithm

Compilers that contain support for prefetching typically base their implementation on Mowry et al.'s prefetch algorithm [11, 93]. Generating prefetch instructions using Mowry et al.'s algorithm requires several steps [79].

1. The compiler performs locality analysis on the array references in a loop to approximate the cache misses.

2. The compiler performs loop unrolling and loop peeling to prefetch the specific references causing cache misses.

3. The compiler attempts to improve prefetch effectiveness by performing software pipelining on loops.

Step 1 requires array dependence analysis to identify the locality relationships between array references within a loop. The dependence information enables the compiler to categorize the types of reuse and locality that occur within a loop. For each reference, reuse analysis determines if the reference contains temporal, spatial, or group reuse. The specific type of reuse guides the loop transformations. Data reuse results in locality only if the data remains in cache. Locality analysis approximates the iteration space of a loop to determine the references that might remain in the cache. Mowry et al. use Wolf and Lam's data locality analysis to determine the reuse relationship for array references [113]. Based upon the locality analysis, Mowry et al. compute a prefetch predicate, which is a function that returns true whenever a reference might suffer a cache miss. Whenever the predicate indicates true, the compiler needs to generate a prefetch.

Steps 2 and 3 of the prefetch algorithm are responsible for scheduling prefetch instructions according to the prefetch predicates. The goal of the second step is to reduce the cost of a dynamic prefetch instruction. Mowry et al. use loop peeling, loop unrolling, and strip mining to isolate the loop iterations that satisfy a prefetch predicate. Loop transformations improve prefetch effectiveness by prefetching the first array elements prior to starting the loop, eliminating prefetches that hit in the L1 cache, and eliminating useless prefetches past the end of the last array element.

Loop peeling removes one or two iterations from a loop so that they are executed prior to entering the loop. Loop unrolling makes additional copies of the code in the loop, and executes several iterations of the original code in a single iteration of the unrolled loop. Loop unrolling reduces the number of unnecessary prefetches by unrolling the loop according to the cache line size.

```
for (int i=0; i<n; i++) {
  sum += arr[i];
}
```

Figure 4.2. Original Loop

```
for (int i=0; i<n-3; i=i+4) {
  sum += arr[i];
  sum += arr[i];
  sum += arr[i];
  sum += arr[i];
}
```

Figure 4.3. Unrolled Loop

```
for (int i=0; i<10; i=i+4) {
  prefetch(&arr[i]);
}
int i=0;
for (; i<n-3; i=i+4) {
  prefetch(&arr[i+10]);
  sum += arr[i];
  sum += arr[i+1];
  sum += arr[i+2];
  sum += arr[i+3];
}
for (; i<n; i++) {
  sum += arr[i];
}
```

Figure 4.4. Loop After Transformations

Figures 4.2, 4.3, and 4.4 illustrate the steps of Mowry's algorithm using a simple example. If four array elements fit on a cache line, then the locality analysis in Step 1 determines that every fourth access of the array is a cache miss. Figure 4.3 shows the code after Step 2 performs loop unrolling so that each prefetch operation brings in a different cache line. Figure 4.4 shows the code after Step 3 performs software pipelining. Software pipelining prefetches the first few elements in the arrays prior to entering the loop. In this example the prefetch distance is 10 array elements.

Although our example is very simple, the analysis and transformations that the compiler needs to perform are complex. In loops with control flow, inner loops, and multiple array references, it is easy to imagine cases when the compiler is unable to compute precise information that is necessary for the prefetch algorithm.

4.1.2 Our Prefetch Algorithm

Our prefetch algorithm does not perform locality analysis or loop transformations, which reduces the complexity of our approach. Our results in Chapter 5 suggest that loop transformations are not required to achieve significant performance improvements with prefetching. We take advantage of available instruction level parallelism (ILP) in modern

```
for  (int i=0; i<n; i++ ) {
  prefetch(&arr[2*(i+d)]);
  sum += arr[2*i];
}
```

Figure 4.5. Complex Index Expression

```
for  (int i=0; i<n; i++ ) {
  prefetch(&arr[i+d]);
  t = arr[i+(d/2)];
  prefetch(t);
  sum += arr[i].value;
}
```

Figure 4.6. Array of Objects

processors to reduce the effect of unnecessary prefetches. When a processor has available ILP, an unnecessary prefetch is very cheap, and the cost is much less than the benefit from prefetching useful data.

Our algorithm generates a prefetch instruction for array references that contain a linear induction variable in the index expression. The compiler generates the prefetch only if the array reference is enclosed in the loop that creates the induction variable.

Our algorithm generates prefetches for array elements and objects referenced by array elements, if appropriate. Prior prefetching algorithms focus on Fortran arrays and prefetch array elements only. In Java, arrays may contain object references as well as primitive types, such as `double`. For an array of objects, we want to hide the latency of accessing the array element and the object. Figure 4.6 illustrates array object prefetching. The first prefetch instruction is for the array element, and the second prefetch is for the object. The second prefetch must load an array element to get the address of the object. To ensure the array element is in cache, the algorithm must load an array element that has already been prefetched. The prefetch distance for the object is half the distance of the prefetch distance for the array element.

The algorithm allows only linear induction variables because they generate arithmetic sequences. Since the induction variable value changes by the same loop invariant expression in every iteration, the prefetch distance remains the same during each iteration. Polynomial and exponential induction variables generate geometric progressions. To prefetch array references with geometric progressions effectively, a new prefetch distance needs to be computed during each iteration, and the distance depends upon the loop index value.

```
1  let I = IS_out(exit(S)); // exit(S) is the last statement
2  for each assignment, t = arr[v]
3    if (v, e, s,i) ∈ I
4    let l = set of statements in current loop
5    if s ∈ l and is_linear(e)
6      let le = linear(e)
7      let c = increment/decrement value of e
8      let d = prefetch distance * c
9      generate prefetch (&arr[v + d])
10     if array of objects
11       let o = arr[v + d/2]
12       generate prefetch (o)
```

Figure 4.7. Array Scheduling Algorithm

An array index expression may contain other terms besides the induction variable. For example, in Figure 4.5 the array index expression is `2*i`, and the induction variable is `i`. We generate a prefetch in this example because the induction variable is linear. The compiler generates code to add the prefetch distance to `i` before the multiplication.

The pseudo-code in Figure 4.7 summarizes our prefetch scheduling algorithm. The algorithm examines each array load instruction and checks if the index expression is a linear induction variable. The function `is_linear(e)` returns true if either the expression `e` or a subexpression of `e` is a linear induction variable. The function `linear(e)` returns the linear (sub)expression in `e`. The increment/decrement value of `e` is the amount that the expression changes during each iteration. The loop invariant value may be a compile-time or run-time constant. If the loop invariant value is a run-time constant, then the compiler may need to generate code to compute the value. The algorithm always generates a prefetch for an array element regardless of the type of the array. If the array contains references to objects, we generate a prefetch for an object.

We eliminate redundant prefetches using a simple common subexpression (CSE) analysis. Most compilers implement CSE, so one can leverage the existing analysis to eliminate redundant prefetches. A prefetch is redundant if the compiler has already generated a prefetch for the *cache line* that contains the data. We illustrate redundant prefetches in

```
for (int i=0; i<n; i++) {
  prefetch(&arr[i+d]);
  prefetch(&arr[i+1+d]); // redundant
  sum += arr[i] + arr[i+1];
  prefetch(&arr[i+d]); // redundant
  foo(arr[i]);
}
```

Figure 4.8. Redundant Prefetch Example

Figure 4.8. Our algorithm generates a prefetch instruction for each array reference. If a loop accesses the same array element multiple times in the same iteration of a loop, our algorithm generates a prefetch instruction for each of the references. In Figure 4.8, our algorithm generates 3 prefetch instructions. Only the first prefetch is useful. The last two are redundant because they prefetch the same cache line as the first. The CSE phase eliminates all but the first prefetch. The algorithm eliminates only prefetches that are redundant in the same loop iteration.

The CSE analysis eliminates redundant prefetches using the same mechanism for eliminating redundant load instructions. The CSE analysis associates a value with each load expression. When processing a load instruction, the CSE analysis records the value. If a subsequent load instruction contains the same value, the CSE analysis removes the load. The CSE analysis must be conservative and so must invalidate the load values due to intervening store instructions or changes in control flow.

We use the existing CSE optimization in the compiler, but eliminating redundant prefetches is simpler than the standard CSE analysis. The largest impact comes from eliminating prefetches that are redundant in the same loop iteration. Rather than tracking values for all expressions and having to deal with control flow, we restrict the analysis to prefetch instructions only. Furthermore, we invalidate the analysis information when following the back edge of a loop.

```
class SList {
  int   data;
  SList next;
  int sum() {
    prefetch(next);
    if (next != null)
      return data + next.sum();
    return data;
  }
}
```

Figure 4.9. Prefetching a Singly Linked List

```
class Tree {
  int  data;
  Tree left;
  Tree right;
  int sum() {
    prefetch(left);
    prefetch(right);
    int s = data;
    if (left != null)
      s += left.sum();
    if (right != null)
      s += right.sum();
    return s;
  }
}
```

Figure 4.10. Prefetching a Binary Tree

4.2 Greedy Prefetching

In this section, we describe the greedy prefetching algorithm that prefetches directly connected objects in linked structures. The goal of greedy prefetching is to hide the latency of accessing future elements in a linked structure traversal. The greedy prefetching algorithm consists of two steps.

1. Identify linked structure traversals

2. Schedule prefetches for fields involved in linked structure traversals

We use the recurrence analysis from Chapter 3 to discover the linked structure traversals. The recurrence analysis also discovers the fields involved in the traversal. The scheduling part of the algorithm inserts prefetch instructions for each set of recurrent field references as early as possible in the program. The number of prefetch instructions that the algorithm generates depends upon the object size and cache line size. The compiler inserts multiple prefetches if the object size is larger than the cache line size, and one prefetch otherwise.

We illustrate an example of greedy prefetching in Figure 4.9 using a singly linked list. The class `SList` contains a `sum` method that adds the elements in the list. Greedy pre-

fetching inserts a prefetch for the `next` field prior to performing computation on the current object. This example shows the main disadvantage of greedy prefetching, which is that it can prefetch only the `next` object in the list. The technique cannot prefetch arbitrary objects because the address of only directly connected objects is known. If the cost of the addition and function call is less than the cost of a memory access, then greedy prefetching only partially hides the read latency of accessing the `next` object.

Achieving the full benefits of prefetching requires that the computation time between the prefetch and use of the object be greater than or equal to the memory access time in order to hide the latency completely. Greedy prefetching is most effective on linked structures that traverse multiple fields within an object. For example, Figure 4.10 shows a `sum` method for a binary tree that performs a depth first traversal using the `left` and `right` fields. Greedy prefetching inserts prefetches for both the `left` and `right` children. Although the prefetches only partially hide the latency of accessing the `left` object, the prefetches may completely hide the latency of accessing the `right` object. The prefetches for the objects at the top of a tree may be useless if the tree is very large, but this occurs infrequently because half the objects are at the leaf nodes.

4.2.1 Intraprocedural Greedy Prefetch Scheduling

The scheduling phase computes which recurrent objects to prefetch and where to insert the prefetch instructions. The algorithm is greedy because we do not perform any analysis to determine if an object is already in the cache, and we try to prefetch as much as possible.

For each recurrent object at each program point, we generate a prefetch for its recurrent fields when the object is not null. The scheduler computes the set of non-null objects using information from the program structure. The scheduler uses a data-flow analysis that computes which objects are null and not-null. The default is to assume an object may be null. Certain program statements establish that an object is not null, and the data-

```
while (o != null) {
  prefetch(o.next);
  prefetch(t.next); // no generated, redundant
  o.compute();
  t = o.next();
  o = t;
}
```

Figure 4.11. Redundant Greedy Prefetch Example

flow analysis propagates the information through the program. The following statements establish that an object is not null:

- An object allocation site. The object on the left-hand side is not null.
- An object comparison to `null`. The object is not null along the false path following the comparison.
- An object field reference. The base object of the field reference is not null, otherwise the reference causes a fault.
- A method call. The first argument is not null following a method call, otherwise a fault occurs.
- Start of a method. The first parameter, *i.e.*, the `this` object, is not null upon entering a method.

For example, a loop that traverses a linked list typically compares the current head element to null at the start of the loop. In this case, the data-flow analysis computes that the head element is not null, and the scheduler can generate a prefetch for the recurrent field in the list.

The scheduler uses alias analysis information to generate a single prefetch for groups of aliased recurrent objects. For example, Figure 4.11 shows a loop with two recurrences, `t` and `o`, that are aliases of each other. The greedy prefetching algorithm only generates

```
1  let R = RA_out(exit(m)); // exit(m) is the last statement
2  for each assignment, o = expr, at statement s:
3    if o is not null // uses the null/not-null analysis
4      for each tuple (o,f,s,r) ∈ R
5        // generate multiple prefetches for large objects
6        c = 0; size = sizeof(o);
7        while (c < size) {
8          generate prefetch (o.f+c)
9          c += cache line size;
10       }
11       remove (o,f,s,r) from R
12       for each p that is an alias of o
13         remove (p,f,s,r) from R
```

Figure 4.12. Intraprocedural Greedy Prefetch Scheduling Algorithm

a single prefetch instruction in this example. The pseudo-code in Figure 4.12 summarizes our intraprocedural scheduling process.

If the size of the object is greater than the cache line size, then the compiler inserts multiple prefetches in order to prefetch the entire object. A command line option specifies the cache line size.

The scheduling phase inserts prefetches for all the individual elements of an array, if the array contains recurrent objects and the size of the array is a small compile-time constant. Computing the size of a Java array is not trivial since Java programs allocate arrays dynamically at run time. Many programs allocate arrays of the same type using compile-time constants, which makes it possible for the compiler to determine the size of an array. When performing interprocedural analysis, the compiler analyzes all the array allocation sites, and computes the set of constant size arrays of the same type and size, as we describe in Section 3.4.2. Figure 3.13 shows an example of a program that uses an array to represent a tree with eight children. In Figure 4.13, we show the `count` method after performing greedy prefetching.

```
int count() {
  int c = data;
  prefetch(children[0]);
  prefetch(children[1]);
  prefetch(children[2]);
  prefetch(children[3]);
  prefetch(children[4]);
  prefetch(children[5]);
  prefetch(children[6]);
  prefetch(children[7]);
  for (int i=0; i<children.length(); i++) {
    if (children[i] != NULL) {
      c += children[i].count();
    }
  }
  return c;
}
```

Figure 4.13. Greedy Prefetching on an Oct-tree

4.2.2 Interprocedural Greedy Prefetch Scheduling

The greedy prefetching algorithm uses an interprocedural scheduling phase to generate prefetches for recurrent parameters. As long as we perform interprocedural recurrence analysis, we can extend the intraprocedural algorithm by adding the recurrent parameters. Let RA_{intra} be the recurrence information computed by the intraprocedural analysis, and R_m be the recurrence information for formal parameters in method m. To generate prefetches for interprocedural recurrences, we change the first line in Figure 4.12 to `let R` = $RA_{intra} \cup R_m$.

Extending the intraprocedural algorithm to include recurrent parameters may result in unnecessary prefetch instructions. We illustrate this problem with an example in Figure 4.14. The program traverses a linked list using recursion and calls `compute()` for each object in the list. The interprocedural recurrence analysis identifies the `next` field in the `this` object in method `sum` as recurrent. Since `sum` calls `compute` with the `this` object, the recurrence analysis also identifies the `this` object in `compute` as recurrent. As Figure 4.14 shows, the intraprocedural scheduling algorithm generates a prefetch instruc-

```
int sum() {
  prefetch(next);
  if (next != null)
    return compute() + next.sum();
  return compute();
}
int compute() {
  prefetch(next); // redundant!
  return data * 2;
}
```

Figure 4.14. Naive Interprocedural Prefetch Scheduling

tion in `sum` and `compute`. The prefetch in `compute` is redundant because it prefetches the same object as the prefetch in `sum`.

The redundant prefetches are due to recurrent objects passed as parameters from a recursive method to another method. We minimize redundant prefetches as follows. The interprocedural scheduling algorithm performs a single, in-order pass over the call graph to schedule recurrent parameters as high as possible in the call graph. The scheduler does not insert a prefetch of a recurrent parameter when the scheduler inserts a prefetch for the parameter in a calling method. Using the example in Figure 4.14, the interprocedural scheduling algorithm does not schedule a prefetch for the `this` object in `compute` because it schedules a prefetch for the same object in the caller, `sum`.

Another source of useless prefetches occurs in non-recursive methods due to method overriding. When a program contains several implementations of a method that has a recurrent parameter, but only one of the implementations is recursive, our analysis indicates that the parameter is recurrent in all the implementations. We eliminate this source of useless prefetches by not generating a prefetch for a field of a recurrent parameter if the callee does not reference the field.

4.3 Jump-Pointer Prefetching

In this section, we discuss the design and implementation of compile-time jump-pointer prefetching. Jump-pointers are a flexible mechanism for prefetching linked data structures because the technique can prefetch arbitrary objects, not just directly connected objects. Jump-pointer prefetching is potentially able to tolerate any amount of latency by varying the prefetch distance between two objects.

Jump-pointer prefetching adds information to an object to indicate which object to prefetch. The target object does not need to be directly connected to the source object. Instead, we add a new prefetch field to the source object, and we generate code to initialize the jump-pointer at run time and to use the jump-pointer for prefetching. The jump-pointer is effective when the creation and access order of the data structure are similar. Figure 4.15 illustrates jump-pointer prefetching for a binary tree. Each tree node contains a jump-pointer to a tree node two links away. Thus we issue a prefetch for node 3 when the program accesses node 1. The number of links depends upon the amount of latency that needs to be hidden. In this example, the program accesses the nodes using a depth first traversal starting with the left child. In the picture, the program accesses the nodes in increasing order from node 1 to node 7. If the program also creates the tree top-down starting with the left child, then we add the jump-pointers in the forward direction, from a lower numbered node to a higher numbered node. We show forward jump-pointers in Figure 4.15. If the program creates the nodes top-down starting with the right child, then we add jump-pointers in the reverse direction, from the higher numbered node to the lower numbered node. Unfortunately, if the program creates the tree bottom-up, then we cannot create effective jump-pointers for top-down traversals.

Greedy prefetching restricts the amount of latency tolerance by prefetching direct links only, but does not require an additional field to store the jump-pointer. Jump-pointer prefetching may also reduce the number of prefetches, yet still remain effective. In Figure 4.15 for example, greedy prefetching adds two prefetches for each node reference, but jump-

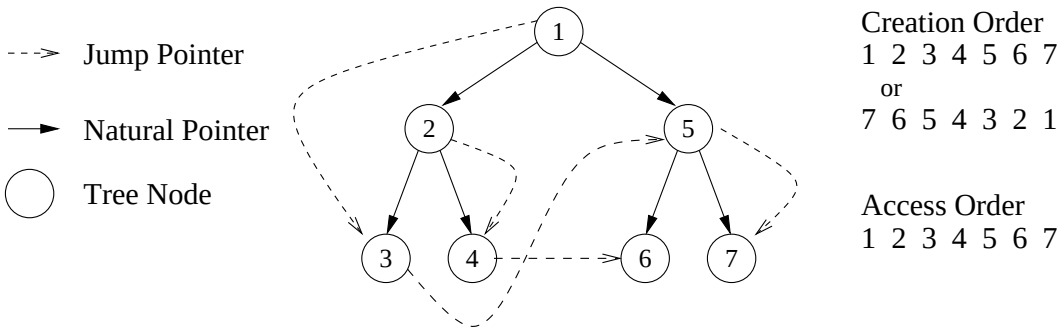


Figure 4.15. Jump-Pointer Prefetching: Binary Tree Traversal

pointer prefetching adds only one. Furthermore, jump-pointer prefetching does not prefetch `null` objects at the leaf nodes.

Our compiler automates jump-pointer prefetching by inserting code to initialize and update the jump-pointers as well as inserting prefetch instructions at appropriate places in the program. The jump-pointer prefetching scheduling algorithm consists of three steps:

1. Identify linked-structure traversals
2. Schedule prefetches for objects containing jump-pointers
3. Insert the code to create the jump-pointers

Just as with greedy prefetching, we use the recurrence analysis from Chapter 3 to discover linked-structure traversals. The second step uses the scheduling algorithm in Figure 4.12 with a couple of minor changes. Instead of generating a prefetch for each recurrent field at line 8, the compiler generates a prefetch for the jump-pointer field, *i.e.*, `prefetch (o.prefetch)`. The third step is the major difference between jump-pointer and greedy prefetching, and we describe it below.

4.3.1 Creating Jump-Pointers

The compiler creates jump-pointers either when an object is *created*, *e.g.*, using `new`, or when *traversing* a data structure. We use a compiler option to specify the choice. By

```
class Tree
{
  int  value;
  Tree left;
  Tree right;
  Tree prefetch;
  static Tree[] jumpQueue;
  static int jumpIndex; // used in example
  static Tree queuePtr; // used in implementation
}
```

Figure 4.16. Binary Tree Class Definition with Jump-Pointer Field

```
Tree createTree(int l)
{
  if (l == 0) return null;
  else {
    Tree n = new Tree();
    jumpObj = jumpQueue[jumpIndex];
    jumpObj.prefetch = n;
    jumpQueue[jumpIndex++ % size] = n;
    Tree left = createTree(l-1);
    Tree right = createTree(l-1);
    n.left = left;
    n.right = right;
    return n;
  }
}
```

Figure 4.17. Inserting Jump-Pointers for a Binary Tree

default, our compiler builds jump-pointers at the object creation site. In our current implementation, the compiler adds only one jump-pointer field to a recurrent object. We do not create jump-pointers when linked structures are updated, unless the update occurs while traversing the linked structure. The effectiveness of jump-pointer prefetching depends on when and where the compiler creates the jump-pointers. We discuss each choice in detail below.

Figure 4.16 shows the extra field members that we add to each class that uses jump-pointers. The `prefetch` field is the jump-pointer. The initial value for the prefetch field

is `null`, although the field may be set to refer to itself instead. We add `jumpQueue`, `jumpIndex`, and `queuePtr` as *static* fields to assist with creating the jump-pointers. Note that in our implementation we use `queuePtr` and `jumpQueue` only. We use the `jumpIndex` field to illustrate the jump-pointer creation process. The fields `jumpQueue` and `jumpIndex` are static members of the `Tree` class that the class initialization method allocates and initializes. We initialize each entry in `jumpQueue` to a special dummy object. Figure 4.17 shows the code for initializing jump-pointers in a binary tree object at creation time. The circular queue, `jumpQueue`, maintains a reference to the last *n* objects allocated. The compiler uses a separate circular queue for each class that contains a jump-pointer.

When an object allocation occurs, the code creates a jump-pointer from the object at the head of `jumpQueue` to the new object. Then the code inserts the new object at the end of `jumpQueue`, and advances the circular queue index. Currently, our compiler creates jump-pointers from the `jumpQueue` object to the current object unless a command line option specifies the reverse direction. We also use a circular queue when the compiler updates the jump-pointers during a traversal.

The code sequence in Figure 4.18 replaces the use of `jumpQueue` and `jumpIndex` with `queuePtr`, a pointer to the current entry in `jumpQueue`. The class initialization method sets `queuePtr` to the start of `jumpQueue`. The jump-pointer creation code sequence is more efficient when it uses one static field member rather than two static field members. Since the SPARC uses two instructions to load or store a global variable, we need to minimize the number of references to global variables. An alternative approach is to use a global register to maintain the jump queue pointer instead of a static field member, but a program may have several jump queues.

Another efficiency factor is the queue size. The jump-pointer creation code is efficient only if the queue size is a power of two because the code sequence can use cheap bit

```
1  ; l0 contains the new node
2  ; l1 contains the jump queue ptr
3  ; jump queue contains 8 objects
4  ; prefetch field is located at offset 20
5  sethi %hi(queuePtr),%l1        ; load the jump queue ptr
6  ld    [%l1+%lo(queuePtr)],%l1  ; (two insts on SPARC)
7  ld    [%l1],%l2                ; load object from queue
8  st    %l0,[%l2+20]             ; create jump-pointer
9  st    %l2,[%l1+%g0]            ; store new obj. in queue
10 add   %l1,4,%l3                ; incr. queue ptr, and
11 and   %l3,31,%l3               ; wrap if at end of
12 and   %l1,-32,%l1              ; queue
13 or    %l1,%l3,%l1
14 sethi %hi(queuePtr),%l2        ; store the new queue ptr
15 st    %l1,[%l2+%lo(queuePtr)]  ; (two insts on SPARC)
```

Figure 4.18. Sparc Assembly for Creating Jump-Pointers

mask operations instead of an expensive division operation. Figure 4.18 shows the SPARC assembly code that the compiler generates when the queue size is a power of two.

Figure 4.18 creates the jump-pointer from the object in the jump queue to the newly allocated object at line 8. In lines 10 – 13, we increment the queue pointer to the next location. We use a series of bit operations to ensure that the pointer does not exceed the queue size.

Figure 4.19 shows two other possible code sequences to create the jump-pointers. The sequence in Figure 4.19 (a) uses an explicit check to test if the jump queue index needs to be reset to the start. This sequence is efficient when the size of the jump queue is not a power of 2. Otherwise, the sequence in Figure 4.17 is more efficient. The sequence in Figure 4.19 (b) uses separate variables for each element in the jump queue. This approach mimics a circular queue by copying objects between the variables in a last-in, first-out manner. This sequence is efficient only if the jump queue size is very small because the cost of the sequence is proportional to the queue size.

```
jumpObj = jumpQueue[jumpIndex++];
jumpObj.prefetch = n;
if (jumpIndex > queueSize) {
  jumpIndex = 0;
}
```

(a) An explicit check

```
jumpObj4.prefetch = n;
jumpObj4 = jumpObj3;
jumpObj3 = jumpObj2;
jumpObj2 = jumpObj1;
jumpObj1 = n;
```

(b) Separate variables

Figure 4.19. Creating Jump-Pointers

4.3.1.1 Object Creation

Adding jump-pointers during object creation is beneficial for data structures with regular access patterns that do not change frequently. This choice minimizes the run-time cost because the jump-pointers are created once. Unfortunately, it is not always possible to create effective jump-pointers at the creation site. For example, in Figure 4.15, the creation phase must be preorder, beginning with either the left or right subtree. If the program builds the tree bottom-up, then the jump-pointers will not be useful. Another problem occurs in programs that frequently update a linked structure that contains jump-pointers because the original jump-pointers no longer correspond to the original structure.

4.3.1.2 Traversal

Building jump-pointers during traversals is effective for programs that contain multiple instances of a linked structure that a program traverses frequently and may also update. Due to the overhead of maintaining the jump-pointer queue, this approach is less effective when programs do not change the linked structures, or when the traversal patterns change frequently, *e.g.*, traversing a list in one direction alternating with a traversal in the reverse direction. An advantage of initializing the jump-pointers during traversal is that the code to create jump-pointers appears locally with the prefetches, which means the compiler does not need knowledge of the entire program.

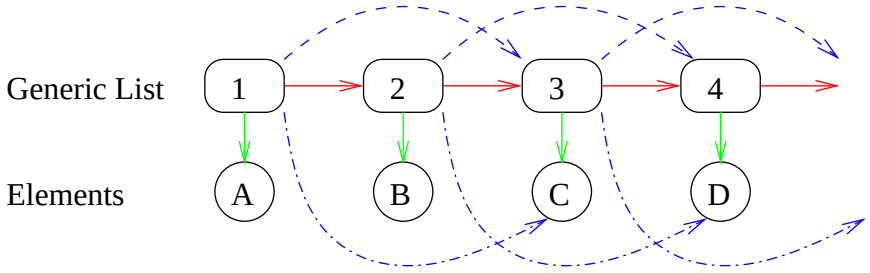


Figure 4.20. Example: Indirect Jump-Pointer

4.3.2 Indirect Jump-Pointers

Section 3.3.6 discusses our analysis for discovering indirect recurrent variables. An indirect recurrent variable is a unshared object that is referenced by a recurrent variable. We prefetch indirect recurrent variables by creating a second jump-pointer from the recurrent variable to the indirect recurrent variable. We illustrate indirect jump-pointers in Figure 4.20, which contains a generic linked list (the rectangles) with pointers to the list elements (the circles). If a program allocates the list objects in order, `A`, `B`, `C`, and `D`, then we add jump-pointers as illustrated (`1` to `C`, `2` to `D`, etc.). When the program traverses the linked list, we schedule prefetch instructions for the list and element jump-pointers. Greedy prefetching is unable to prefetch these objects effectively because there are no direct links between them.

4.3.3 Garbage Collection

Java uses garbage collection for automatic memory management instead of allowing the user to manage dynamically allocated objects. In this work, we use a generational copying garbage collector. Garbage collection has a significant impact on our jump-pointer prefetching implementation. Jump-pointer prefetching adds a field to each object that indicates the object to prefetch. The garbage collector needs to be aware of the field, and must handle the field specially. When the collector copies an object, it must update the jump-pointers to point to valid, preferably useful, objects. By updating the jump-pointers,

the collector can potentially improve the effectiveness of the jump-pointers. We discuss the relationship between the collector and prefetching in more detail in Chapter 6.

The garbage collector computes which objects are live and reclaims the rest of the objects. It then uses the reclaimed memory for future allocations. The collector computes the live objects by identifying an initial set of root objects as live, and then the collector traces all the objects reachable from the root objects by following the pointer fields of each reachable object. It is important that the collector does *not* trace the jump-pointer fields. If the only reference to an object is through the jump-pointer, then the collector should identify the object as dead (unreachable) and reclaim the space. A memory leak occurs if the collector identifies the referent of a jump-pointer as live. If a jump-pointer refers to a dead objects, then the collector must set the jump-pointer to refer to a live, preferably useful, object.

During a collection, a copying garbage collector moves all the live objects to a new region of memory. A copying garbage collector must update the jump-pointers to contain references to the objects in the new region. Otherwise, the jump-pointers become invalid because they point to unallocated data. We solve the jump-pointer problem by treating the collector as a traversal phase. We add code to the collector to re-initialize the jump-pointers using a jump-pointer queue. As the collector copies objects, it creates a jump-pointer from an object on the queue to the copied object, and then inserts the copied object into the queue.

4.4 Stride Prefetching

In this section, we discuss stride prefetching for linked structures. Stride prefetching generates a prefetch for an address *n* bytes ahead or behind the current object in a linked-structure traversal. Stride prefetching works when a linked structure is laid out in consecutive memory locations. Unlike greedy prefetching or jump-pointer prefetching, stride prefetching does not access any fields of the linked structure to perform a prefetch.

```
while (o != null) {
  o.compute();
  o = o.next();
  prefetch(o+64);
}
```

Figure 4.21. Example of Stride Prefetching

Stride prefetching is able to tolerate any amount of latency, but the program may never access the address that is prefetched. If the linked structure is not laid out in consecutive locations, then stride prefetching may potentially hurt performance by bringing in useless cache lines that may displace useful data. Stride prefetching exploits the characteristic that programs often co-locate objects in the same linked structure. In a garbage collected environment, stride prefetching is potentially more effective when the collector uses a copying algorithm that naturally groups together objects in the same linked structure.

The stride prefetch algorithm consists of two steps:

1. Identify linked structure traversals

2. Insert a prefetch for *n* bytes ahead (or behind)

We use the recurrence analysis from Chapter 3 to discover the linked structure traversals. The second step inserts a prefetch immediately after the field reference statement that performs the structure traversal.

Figure 4.21 illustrates stride prefetching for a linked list. In the example, the compiler generates a prefetch for the address 64 bytes ahead of the current object. By default, the algorithm generates a prefetch using a positive value. The prefetch may be more effective if the prefetch distance is a negative value. For example, if a program adds new objects to the beginning of a linked list, then the addresses of the objects in the list most likely decrease during a traversal, but this depends upon the allocator. When the compiler performs interprocedural analysis, it uses a heuristic to classify the memory order of the objects in the list. The heuristic examines how a program inserts a new object into a linked structure. If

the program assigns the new object to the linked structure, then the compiler uses a positive distance. If the program assigns the linked structure to the new object, then the compiler uses a negative distance.

4.5 Implementation in Vortex

We implement the recurrence analysis and prefetching algorithms in Vortex, an optimizing compiler for object-oriented programs. We briefly describe Vortex in Section 2.5. In this section, we describe the implementation details and our extensions. Figure 4.22 presents a high level overview of the compiler with our extensions. We list our extensions in bold.

4.5.1 Interprocedural Analysis

The first main phase of the compiler is interprocedural analysis. Vortex performs interprocedural analysis using the entire program. Vortex contains an interprocedural data-flow analysis framework for performing whole program optimizations. The framework presents a uniform interface that allows a compiler writer to define interprocedural optimizations conveniently. Using the framework, the compiler writer can specify the context-sensitivity and flow-sensitivity. An important part of the framework is that it allows the compiler writer to define an interprocedural analysis using the intraprocedural analysis as a component. We discuss the intraprocedural data-flow analysis framework in detail in Section 4.5.2. The interprocedural analysis algorithm operates on the program call graph, starting at the main node.

To perform a interprocedural analysis and optimization, Vortex must first create a call graph of the program. The call graph is a representation of calling relationships between the procedures, or methods, in the programs. The graph contains a node for each procedure, and a directed edge between two nodes indicates that one procedure may call the other. For example, if procedure A calls procedure B, then the call graph contains an edge from the

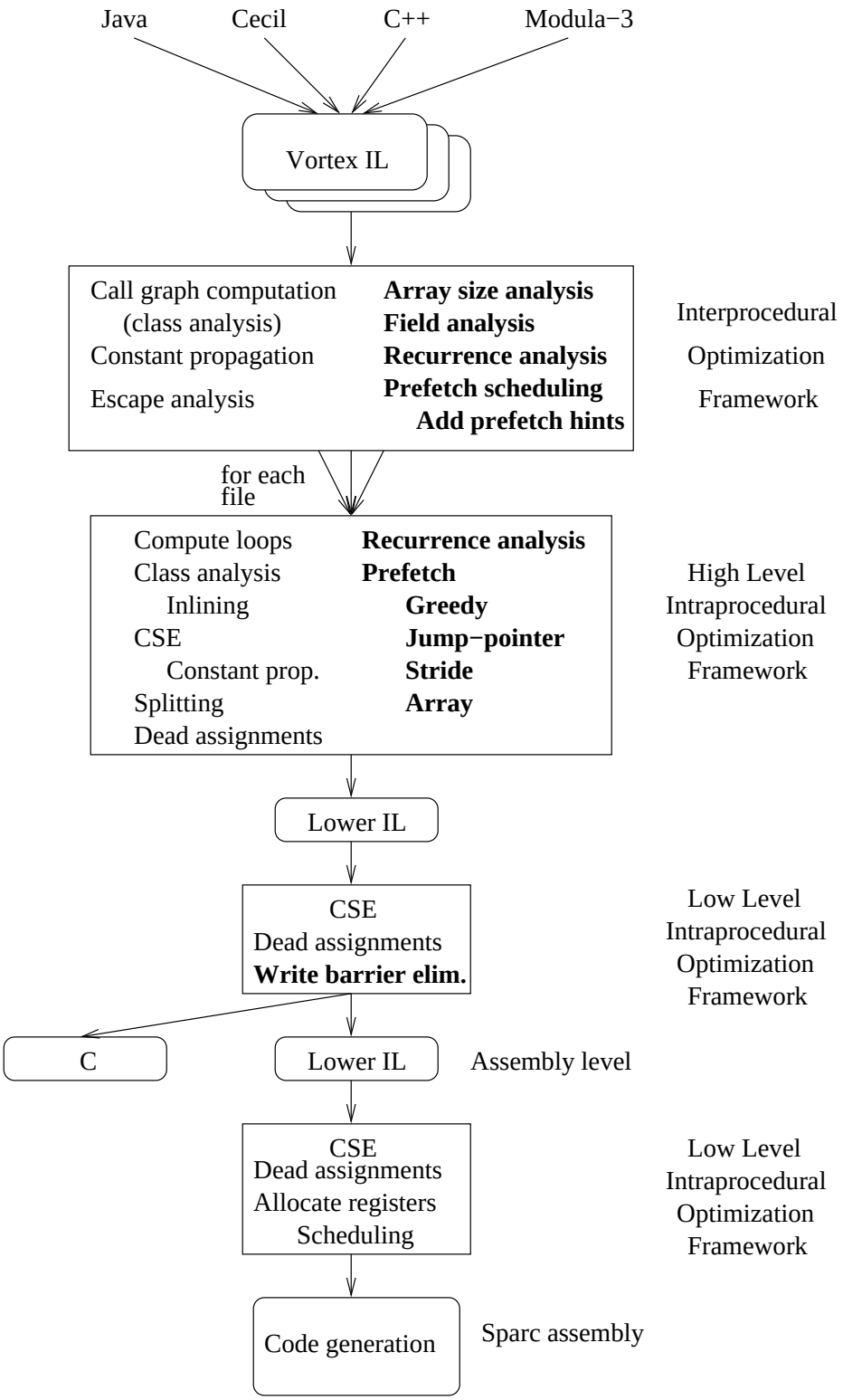


Figure 4.22. Overview of the Vortex Compiler: With Our Extensions

node representing A to the node representing B. In some languages, such as C or Fortran, creating the call graph is fairly simple and straightforward, except when C programs use function pointers frequently. The program source indicates the specific target for each procedure call. In object-oriented languages, the call graph may be more difficult to create accurately due to virtual method calls. For a virtual call, the target may not be known until run time. When the exact target is not known, the call graph must conservatively include edges to all potential targets.

Vortex analyzes the program to reduce the number of potential targets of a method call, which reduces the size of the call graph. The benefit of a more accurate call graph is that interprocedural optimizations are more efficient. Vortex contains a set of algorithms for constructing the call graph that vary in complexity and accuracy. The default class analysis algorithm in Vortex is 0-CFA (zero-order control-flow analysis) [95]. Shivers originally defined the *k*-CFA algorithm for Scheme programs. The algorithm is flow sensitive, and the algorithm is context insensitive when *k* is 0. Larger values for *k* indicate the degree of context sensitivity. The 0-CFA algorithm performs an iterative data- and control-flow analysis of the program when constructing the call graph. The algorithm propagates type information available in the program to compute potential callees at each call site. In practice the algorithm works well and is reasonably fast for large programs.

After performing call graph analysis, Vortex performs interprocedural analysis by iterating over the call graph. Vortex implements several interprocedural algorithms including constant propagation, escape analysis, and mod/ref analysis. We implement several new interprocedural analysis phases, including array size analysis, shared object analysis, recurrence analysis, and prefetch scheduling. We describe the array size analysis in Section 3.4.2, the shared object analysis in Section 3.4.1, the recurrence analysis in Section 3.3.3, and the prefetch scheduling algorithms in the previous sections of this chapter.

Each interprocedural analysis may be run independently of the others using command line options. We group some of the optimizations together when running the compiler.

When we run array size analysis, we also run interprocedural constant propagation. Otherwise we are unable to identify array sizes accurately. When we perform interprocedural linked-structure prefetching, we also run the array size analysis to help identify prefetch opportunities. Of course, we first perform interprocedural recurrence analysis when running any of the interprocedural prefetch algorithms.

The interprocedural optimizations do not actually make changes to the intermediate representation. An interprocedural analysis records analysis information in a separate data structure that Vortex uses during the intraprocedural analysis phase of the compiler. Since an interprocedural analysis uses the intraprocedural analysis as a component, incorporating the interprocedural results is straightforward. In general, the interprocedural analysis records information about the method formal parameters and return values.

4.5.2 Intraprocedural Data-Flow Analysis and Optimization

Vortex operates on each file, *i.e.*, Java class, separately after the interprocedural phase. The user may specify different compilation levels and enable/disable specific optimizations for each file. The intraprocedural compilation step consists of several phases starting with a high-level representation and successively lowering the representation until code generation. During each phase the compiler performs a set of optimizations using a data-flow analysis framework.

We describe the foundations of data-flow analysis in Section 2.3. The data-flow framework in Vortex is general and parameterizable. The compiler writer describes the following information about a specific data-flow analysis problem:

- A data structure to represent the data-flow information, *e.g.*, a bit vector
- The join operator to combine data-flow information, *e.g.*, bit vector OR
- A function that returns true once the analysis reaches a fixed point, *e.g.*, checking if the bit vector changes

- Transfer functions for each appropriate statement type
- The direction of the analysis, either forward or backward
- The flow sensitivity of the analysis, either flow-sensitive or flow-insensitive

The analysis framework iterates over the control-flow graph and calls the user-supplied transfer function for each appropriate statement. The framework applies the identify transfer function to all other statements. The framework allows the transfer functions to modify the flow graph during an analysis.

4.5.2.1 High-Level Optimization

Vortex initially performs several optimizations on the high-level intermediate representation. The first step is to analyze each method to identify loops using a dominator algorithm. When the user specifies the highest level of optimization, Vortex performs the following optimizations during a single pass:

- Class analysis and method inlining
- Common subexpression elimination, with constant and copy propagation
- Splitting

The goal of class analysis is to analyze the program in order to convert virtual method calls to direct calls. Since class analysis determines which method calls can be direct, it is also responsible for inlining appropriate methods. Vortex uses several heuristics, including a cost model that depends upon the method's expressions, to determine the direct method calls to inline. Class analysis works by propagating type information throughout a method. Many statements in a method provide explicit type information, such as `new` expressions that create objects of a specific type.

The common subexpression elimination (CSE) phase performs constant and copy propagation as well as removing redundant expressions. The CSE algorithm uses value numbering to compute equivalent expressions. CSE also attempts to eliminate redundant load and store expressions whenever it is safe. As part of constant propagation, the CSE phase eliminates branches when the outcome of the branch is a known constant value.

Splitting is a technique that eliminates redundant type tests. Prior to a virtual method call, Vortex inserts a type test that checks the object type of the method call, *i.e.*, message send. The type test enables Vortex to generate a direct method call instead of an indirect call because the type of the callee, *i.e.*, receiver, is known. If a program contains several method calls to the same object, then Vortex generates the *same* type test prior to each call. Vortex uses a forward type propagation data-flow analysis to determine when it is possible to apply splitting. At each type test, the analysis checks if a prior control flow merge includes the type as a possible data-flow value. If so, then the compiler attempts to move the statements below the current type test to the prior type test. Vortex's implementation of splitting does not support splitting past a loop node.

Vortex runs the recurrence analysis and prefetch scheduling algorithms during the high-level optimization phase. We implement the recurrence analysis and prefetch optimization as a single data-flow analysis pass. As the recurrence analysis discovers linked structures or induction variables, the prefetch optimization uses the information to insert prefetch instructions appropriately. The prefetch optimizations also require type information, so Vortex runs the prefetch algorithm with class analysis, if class analysis has not yet been performed.

Vortex performs dead-assignment elimination once all the major optimizations and analyses are done. Dead-assignment elimination performs a reverse pass over the control-flow graph. The optimization eliminates a statement if the left hand side value is never used again, and the right hand side does not cause an exception.

4.5.2.2 Low-Level Optimization I

After performing the high-level optimizations, Vortex converts some of the high-level operations in the intermediate representation to a low-level form. The main reason for the conversion is to prepare the compiler to generate C code. The compiler stills performs the lowering even when generating assembly language. The lowering phase uses the data-flow analysis framework to traverse the intermediate representation and replace nodes. The lowering is a single pass and does not require a data-flow meet operator.

Lowering is mainly responsible for cleaning up the intermediate representation after applying the high-level optimizations. The lowering phase eliminates some type tests and lowers operations that are specific to Cecil and Modula-3.

After lowering the representation, Vortex runs another set of optimizations. It performs common subexpression elimination, dead-store elimination, dead-assignment elimination, and write-barrier elimination. Dead-store elimination attempts to delete useless store and load instructions. This is different from dead-assignment elimination, which attempts to delete the results of useless computations. Dead-store elimination also performs a reverse pass over the control-flow graph and records the memory locations at each store and load instruction.

Vortex eliminates unnecessary write-barrier code sequences during this compilation phase. Vortex generates a write barrier for each pointer store instruction when the compiler generates assembly language and uses the generational garbage collector. The write barrier keeps track of references from older generations to younger generations. It is not necessary to generate a write barrier if the modified object is already in the youngest generation. The write-barrier elimination optimization indicates that the write barrier is not necessary when the source object is known to be in the nursery.

4.5.2.3 Low-Level Optimization II

When generating assembly language, Vortex lowers the intermediate representation in preparation for code generation. The goal of this lowering pass is to create a single node in the intermediate representation for each machine instruction. The lowering translates all high-level nodes, such as array references and object creation operations, into a sequence of low-level operations.

Vortex performs a series of optimizations on the lowered representation because the lowering may expose more optimization opportunities. The optimizations include common subexpression elimination, dead-store elimination, and dead-assignment elimination.

Vortex performs global register allocation and scheduling on this representation. The global register allocation implementation is based upon Briggs et al.'s algorithm [13]. The global register allocator creates an interference graph and assigns registers using a graph coloring algorithm. The algorithm spills registers to the stack as necessary. After performing register allocation, Vortex traverses the intermediate representation, adds moves, loads and stores, and replaces variables with physical registers. The scheduling phase is very simple and only tries to fill the delay slot of branch instructions on processors that use delay stots.

4.5.2.4 Code Generation

The final phase of the compiler generates SPARC assembly language. The code generator is straightforward because each node in the internal representation represents roughly one instruction. The code generator is responsible for choosing the correct instruction based upon the node type. In some cases, the code generator must generate a sequence of instructions. For example, the code generator generates a write-barrier sequence for each pointer store that needs one.

```
1  let ra := pair(new_recurrence_info(),
2                 &(n:RTL, data_flow_info:recurrence_info){
3                   n.find_recurrences(data_flow_info)
4                 });
5  -- add interprocedural recurrence information
6  let pf := pair(new_greedy_schedule_info(),
7                 &(n:RTL, data_flow_info:schedule_info) {
8                   n.schedule_prefetch(data_flow_info)
9                 });
10 traverse(cfg, forward, iterative,
11          new_composed_analysis([ra, pf]),
12          &(n:RTL, ca:composed_analysis_info) {
13            n.process(ca);
14          });
```

Figure 4.23. Example of Prefetch Optimization in Vortex

The code generator is responsible for generating the assembly directives for global variables, procedures, and other miscellaneous structures. The output of the code generator is a file containing valid assembly code for the SPARC assembler.

4.5.3 Implementation of Prefetching

In this section, we describe some details of our prefetch implementation in the Vortex infrastructure. Each of our prefetch algorithms has a similar form:

- Identify prefetch opportunities
- Schedule prefetch instructions

We use our recurrence analysis from Chapter 3 to discover the prefetching opportunities and the prefetch algorithms from this chapter to schedule prefetch instructions. Although these two steps are logically distinct, we run them together during a single pass of the control-flow graph. In Vortex, we create a composable analysis with the recurrence analysis and a specific prefetch optimization.

Figure 4.23 illustrates the use of the composable analysis in Vortex. We show the code example in Cecil. The & operator defines a closure. In our code example, we define three

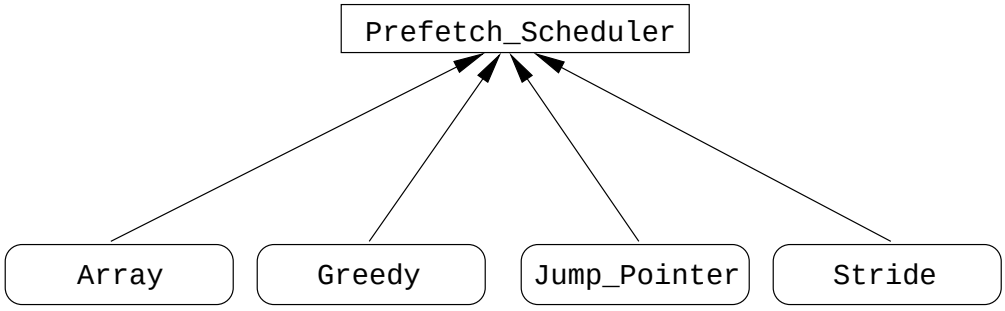


Figure 4.24. Class Hierarchy for Prefetching

closures that each take two parameters. The closures call the transfer functions for the specific data-flow analysis problem.

We create a recurrence analysis object at line 1. The recurrence analysis object is a pair of objects. The first element is a data-flow analysis object specific to the recurrence analysis algorithm. This object defines the information that we describe in Section 4.5.2. The second element is a closure that is executed at each statement. At line 5, we add interprocedural information to the recurrence object, but we exclude the actual code from the figure.

Line 6 defines the data-flow object to perform prefetching. We create a greedy prefetch object to perform the scheduling. We perform jump-pointer prefetching, stride prefetching, or array prefetching by changing line 6 to `new_jump_pointer_schedule_info()`, `new_stride_schedule_info()`, or `new_array_schedule_info()`, respectively.

Line 10 performs the composed intraprocedural analysis involving the recurrence analysis and greedy prefetching optimization. We specify a forward, iterative traversal over the control-flow graph. At each statement, `traverse` calls the closure that contains the call to `process`. The `process` method first calls the closure containing the recurrence analysis, *i.e.*, `find_recurrences`, and then calls the closure containing the prefetch optimization, ie `schedule_prefetches`, for each statement. We specify the analyses that the `traverse` method performs in an array at line 11.

In our implementation, we define an abstract prefetch scheduling class. Each specific prefetch implementation inherits from the abstract class. Figure 4.24 illustrates the class hierarchy we implement in Vortex. The `Prefetch_Scheduler` abstract class defines methods to perform prefetching. The `Greedy`, `Jump_Pointer`, `Stride`, and `Array` classes are subclasses that implement the prefetch algorithms. We implement the scheduling algorithms as data-flow analysis passes. The abstract class maintains a data structure to keep track of the variables that become recurrent. The different concrete scheduling algorithms implement the heuristics for a particular algorithm. For example, the greedy prefetch scheduling algorithm generates a prefetch for the recurrent field after determining that the object cannot be null. The array prefetching scheduler generates a prefetch when an induction variable is used in an array reference. We are able to run different prefetching algorithms at the same time by composing them in the data-flow analysis framework. For example, we prefetch arrays and perform greedy prefetching by creating a composable analysis with the array prefetching scheduler and greedy prefetching scheduler objects.

4.6 Chapter Summary

In this chapter, we describe the implementation of our new array prefetching algorithm, and three linked-structure prefetching algorithms. The linked-structure prefetching algorithms are greedy prefetching, jump-pointer prefetching, and stride prefetching. The overall structure of the prefetch algorithms are similar and require two steps each. In the first step, the compiler identifies the recurrences in the program using the recurrence analysis from Chapter 3. The second step schedules the prefetch instructions. The different algorithms perform different actions to schedule the prefetches. Jump-pointer prefetching requires a third step to initialize the jump-pointers.

We also describe the implementation of the prefetch schemes in Vortex, an optimizing compiler for object-oriented languages. We describe the overall structure of the compiler, and we discuss the changes to the compiler to add the prefetching algorithms. The im-

plementation requires changes in the interprocedural and intraprocedural compiler phases. Although the prefetch algorithms appear to be quite different, the compiler shares a large amount of code between the prefetch implementations.

CHAPTER 5

EXPERIMENTAL RESULTS

In this chapter, we evaluate the effectiveness of array and linked-structure prefetching. We first describe our experimental methodology in Section 5.1. We present results for array prefetching and linked-structure prefetching in Sections 5.2 and 5.3, respectively. For linked-structure prefetching, we present results for greedy prefetching, jump-pointer prefetching, and stride prefetching separately. At the end Section 5.3, we directly compare the results of the three different linked-structure prefetching techniques. In Section 5.4, we vary several architecture parameters to evaluate the effectiveness of prefetching on a range of architectures.

We show that array and linked-structure prefetching are effective techniques for improving the performance of Java programs. Our array prefetch algorithm produces large performance improvements. Array prefetching reduces the execution time in our benchmarks by a geometric mean[1] of 23%. The results show that complex analysis and loop transformations are not necessary to generate useful prefetch instructions for arrays. The linked-structure prefetch optimizations reduce execution time by a geometric mean of 5%, 10%, and 9% for greedy, jump-pointer, and stride prefetching, respectively. The linked-structure prefetch techniques produce improvements in programs that traverse large linked structures in a regular manner. However, generating effective prefetches for short linked structures is difficult.

[1]We use the geometric mean because we compute the mean of normalized execution times.

5.1 Methodology

As we describe in Section 4.5, we use Vortex to compile our Java programs, perform object-oriented and traditional optimizations, and generate SPARC assembly code. Since Vortex reads Java class files (*i.e.*, byte codes) as input, and not Java source files, we compile the Java programs using JDK 1.1.6.

We evaluate our prefetching algorithms using programs from the Jama library [46], the Java Grande benchmark suite [14], and a Java version of the Olden benchmark suite [17]. We evaluate array prefetching using the Jama library and Java Grande programs. The Jama library provides Java classes for performing basic linear algebra operations on dense matrices. The Java Grande benchmark suite is a set of programs for evaluating a variety of Java applications. We use the kernel programs from Section 2 of the sequential benchmarks. These programs operate mostly on large array data structures. We use the Olden benchmarks to evaluate the linked structure prefetching techniques. The Olden benchmark suite contains ten small programs that manipulate linked structures. The original Olden programs were written in C and used to evaluate parallel compiler techniques for linked structures. Other researchers use the C versions to evaluate optimizations, including prefetching, for pointer-based programs [25, 70, 90].

We use RSIM, the Rice Simulator for ILP Multiprocessors, to perform detailed cycle by cycle simulation of the programs. We summarize RSIM's processor model here, but we refer the reader to the RSIM Reference Manual [84] for more details. RSIM contains architecture features to exploit instruction level parallelism (ILP) aggressively. RSIM models a uniprocessor or shared-memory multiprocessor, but we use the uniprocessor configurations only. The key features of the processor model include superscalar execution, out-of-order scheduling, register renaming, dynamic branch prediction, non-blocking loads and stores, and speculative load execution. The key memory hierarchy features include two levels of cache, multiported and pipelined L1 cache, pipelined L2 cache, multiple outstanding cache requests, memory interleaving, and software prefetching.

The RSIM processor model is most similar to the MIPS R10000 [121]. RSIM models the R10000 active instruction list, register map table, and shadow mappers. The active instruction list, also known as a reorder buffer or instruction window, contains the current instructions that the processor can schedule dynamically. The register map table maintains a mapping between the logical and physical registers. The shadow mappers maintain the register state at branches to allow quick recovery on mispredictions. A main difference between RSIM and the R10000 is that RSIM executes the SPARC instruction set, which uses a register window mechanism that the R10000 does not implement.

The RSIM pipeline contains five stages: fetch, decode, issue, execute, and complete. The fetch and decode stages process instructions in program order, but the issue, execute, and complete stages may process the instructions out-of-order. Instructions graduate in-order after passing through all five stages, which enables RSIM to implement precise exceptions.

RSIM allows many of the processor and memory features to be configurable at simulation time. Table 5.1 lists many of the interesting simulation parameters that we use to obtain the base results. We refer the reader to the RSIM Reference Manual to obtain the complete list of parameters. We vary several of the memory parameters in Section 5.4.

We configure RSIM to fetch and graduate a maximum of four instructions per cycle. Our processor configuration contains two ALU, two FPU, and two address generation functional units. RSIM uses a two-bit history branch predictor that contains up to 512 counters. RSIM uses eight shadow mappers, which restricts the number of outstanding branches to eight.

Table 5.1 lists the latencies for the ALU and FPU instructions. The floating point conversion and division instructions have a repeat delay also. The repeat delay is the number of cycles that the processor must wait until using the functional unit after the instruction completes.

Table 5.1. Simulation Parameters

Processor parameters	
Issue width	4
Pipeline stages	5
Active list size	64 instructions
Memory queue size	16 entries
Functional units	2 ALU, 2 FPU, 2 Addr
Branch predication	2-bit history predictor
Outstanding branches	8
Integer multiplication	3 cycles
Integer division	9 cycles
Other integer operations	1 cycle
Floating point mult,add,sub	3 cycles
Floating point conversion	5 cycles, 2 cycle repeat delay
Floating point div,sqrt	10 cycles, 6 cycle repeat delay
Memory hierarchy parameters	
L1 cache	32KB, 32B line direct, write through, 2 ports
L2 cache	256KB, 32B line 4-way, write back, 1 port
Write buffer size	8 entries
Miss handlers (MSHR)	8 L1, 8 L2
L1 hit time	1 cycle
L2 hit time	12 cycles
Memory hit time	60 cycles
Memory banks	4-way interleaved
Bus width	32 bytes
Bus cycle time	3 cycles

RSIM supports non-blocking load and store instructions that may execute out-of-order, but they must appear to execute in-order. RSIM uses miss status holding registers (MSHRs) to maintain information about outstanding requests. RSIM uses a coalescing write buffer for stores. RSIM implements a software prefetch instruction, which brings one cache line into the L1 cache.

The L1 cache is write-through with a no allocate policy. The L1 cache has two ports, which means that two accesses can occur concurrently. The L2 cache is write-back with a write-allocate policy. The L2 cache maintains inclusion with the L1 cache. The L1 and L2 cache line size is 32 bytes. The memory is interleaved and we configure the memory with four banks.

The main metric we obtain from RSIM is execution time in cycles. RSIM divides the execution time in to busy time and memory stall time. The memory stall time includes both load and store instructions, but most of the time is due to load instructions. The busy time includes all other execution cycles, including branches and multi-cycle arithmetic operations. In a processor with ILP, dividing the execution time is not straightforward because instructions may overlap. RSIM counts a cycle as a memory stall if the first instruction that the processor cannot retire in a cycle is a load or store. Otherwise RSIM counts the cycle as busy time.

5.2 Array Prefetching

We evaluate array prefetching using scientific library routines in the Jama package [46], and programs from Section 2 of the Java Grande benchmark suite [14]. Table 5.2 lists the benchmarks we use in our experiments, along with some characteristics of each program. In several Java Grande benchmarks, we use input sizes other than the suggested size in order to complete our simulations within a reasonable time limit. We exclude `series` because it does not use an array as a main data structure.

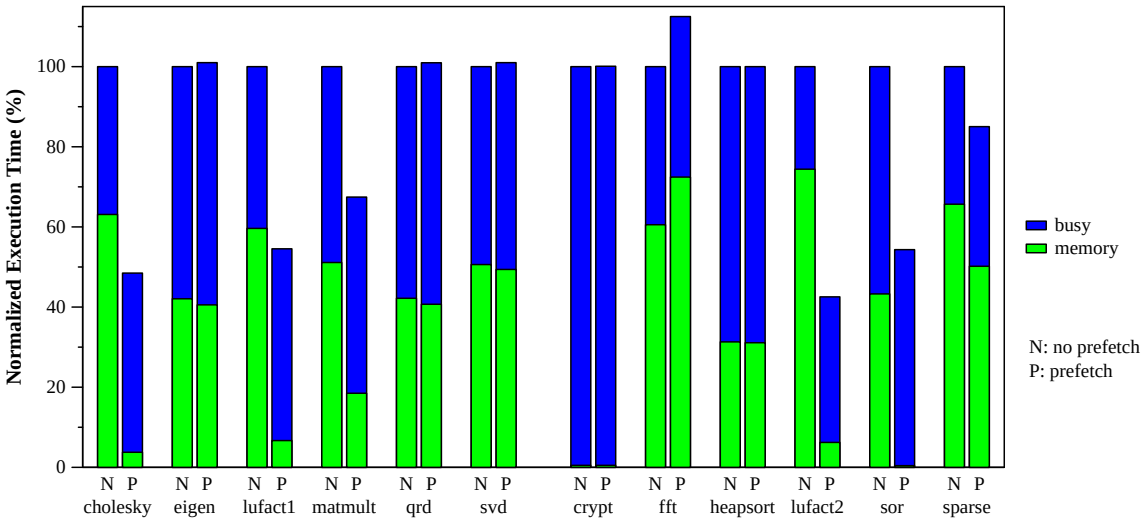


Figure 5.1. Array Prefetching Performance

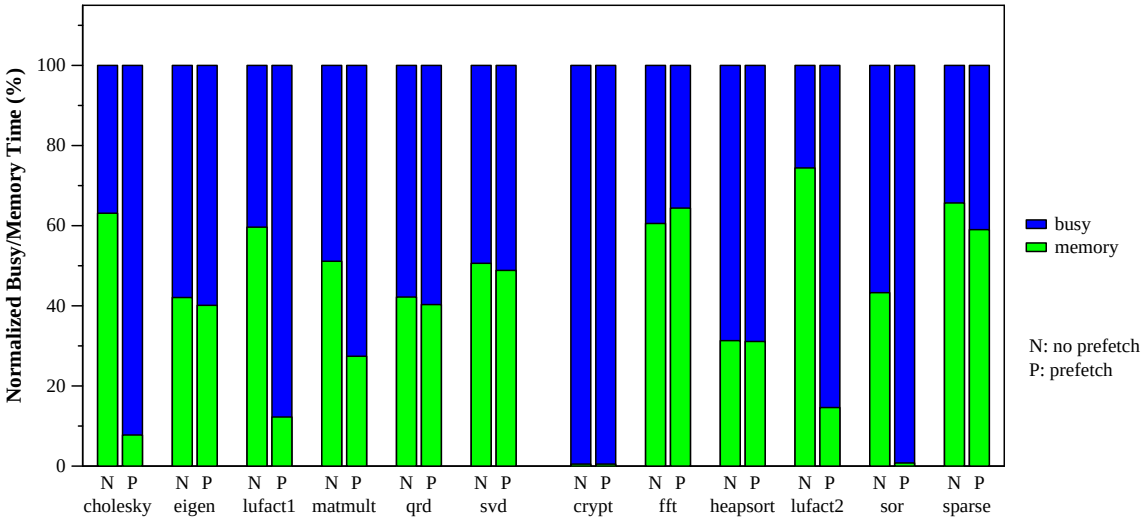


Figure 5.2. Effect of Array Prefetching on Busy/Memory Time

Table 5.2. Array-based Benchmark Programs

Name	Description	Inputs	Inst. Issued
Jama library			
cholesky	Cholesky decomposition	300x300 matrix	1381 M
eigen	Eigenvalue decomposition	250x250 matrix	1675 M
lufact1	LU factorization	300x300 matrix	1570 M
matmult	Matrix multiply	400x400 matrix	1744 M
qrd	QR factorization	400x400 matrix	1811 M
svd	Singular value decomposition	300x300 matrix	5733 M
Java Grande			
crypt	IDEA Encryption	250000 elements	2500 M
fft	FFT	262144 elements	1828 M
heapsort	Sorting	1000000 integers	2916 M
lufact2	LU factorization	500x500 matrix	1167 M
sor	SOR relaxation	1000x1000 matrix	6972 M
sparse	Sparse matrix multiply	12500x12500 matrix	815 M

Figure 5.1 presents the results of array prefetching (P) on our programs. We specify a prefetch distance of twenty elements as a compile-time option. We normalize the execution times to those without prefetching (N). Figure 5.2 normalizes the busy and memory stall times for each program. This graph shows how prefetching changes the amount of time that each program spends waiting for memory stalls. The six programs on the left side are part of the Jama library, and the other six programs are Java Grande benchmarks. We divide execution time into the amount of time spent waiting for memory requests, and the amount of time the processor is busy using the methodology described in Section 5.1.

Figure 5.1 shows that these programs spend a large fraction of time waiting for memory requests. Seven of the twelve programs spend at least 50% of execution time waiting for memory requests. Clearly, these programs have substantial room for improvement.

We see improvements in six programs, performance degrades in four programs, and there is no change in two programs. Across all programs, prefetching reduces the execution time by a geometric mean of 23%. The largest improvement occurs in `lufact2` where prefetching reduces the execution time by 58%. In five of the programs, prefetching reduces

the execution time by more than 30%. Prefetching increases execution time in `fft` by 13% due to a large number of conflict misses. In Section 5.2.3, we show that prefetching improves the performance of a different FFT implementation, which is faster than the Java Grande version.

Prefetching results in large improvements in `cholesky`, `lufact1`, `matmult`, `lufact2`, and `sor`. The performance improvement is due to memory stall reduction. In the programs that improve significantly, the amount of time spent stalling due to memory requests decreases substantially. Prefetching eliminates almost all the memory stalls in `sor`, `cholesky`, `lufact1`, and `lufact2`.

Prefetching does not have any effect on `crypt` or `heapsort`. The time spent on memory stalls in `crypt` is less than 1% of total execution time, so we do not expect to see any performance improvement. The access pattern in `heapsort` is not regular and is data dependent. It is difficult to improve `heapsort` using prefetching.

Software prefetching increases the number of executed instructions. The amount of busy time in the programs tends to increase slightly. The additional functional units in a superscalar processor are able to hide the cost of the additional instructions due to available functional units.

In summary, compile-time data prefetching is effective on array-based programs, even without loop transformations and array dependence information. Our results show that generating prefetches for array references that contain induction variables improves performance substantially. We now explore in more detail how array prefetching achieves its improvements.

5.2.1 Prefetch Effectiveness

Figure 5.3 categorizes the dynamic L1 prefetches as useful, late, early, or unnecessary. We describe the meaning of these categories in Section 2.2.1.1. Figure 5.3 shows that the percentage of useful prefetches does not need to be large to improve performance. The

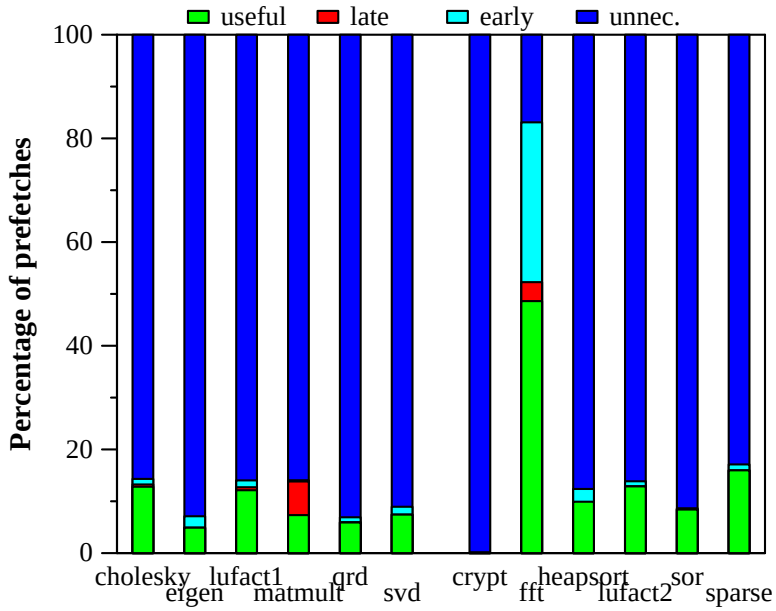


Figure 5.3. Array Prefetch Effectiveness

number of useful prefetches is less than 16% in each program, except for `fft`. In the program which the largest improvement, `lufact2`, the number of useful prefetches is just 12%.

Only `matmult` has a noticeable number of late prefetches. We can slightly improve performance in `matmult` by increasing the prefetch distance which reduces the number of late prefetches. The program with the largest number of useful prefetches, `fft`, is also the program with the worst overall performance. The large number of early prefetches results in poor performance. The early prefetches are due to conflict misses, which we discuss in more detail in Section 5.2.3. Another potential source of early prefetches is due to using a prefetch distance that is too large, but we do not see this effect in our programs.

Figure 5.3 shows that using a twenty element compile-time prefetch distance is effective in achieving most of the performance gains. The prefetch distance is large enough to bring data into the cache when needed, and small enough so that the data is not evicted prior to the demand request. We do not see a compelling reason to use more sophisticated analysis to determine the appropriate prefetch distance automatically. Section 5.2.4 shows that the results are stable when we vary the prefetch distance.

Table 5.3. Array Static and Dynamic Prefetch Statistics

Program	Static	Dynamic		Bus Utilization	
		total prefetches	prefetches/read	N	P
cholesky	32	97 374 473	10 %	13 %	28 %
eigen	153	47 214 494	10 %	8 %	8 %
lufact1	36	48 721 596	12 %	13 %	24 %
matmult	22	77 693 667	38 %	21 %	31 %
qrd	20	40 771 107	9 %	9 %	9 %
svd	74	134 700 673	9 %	10 %	10 %
crypt	31	500 794	3 %	0.1 %	0.1 %
fft	37	7 159 698	12 %	16 %	16 %
heapsort	27	1 442 004	0.3 %	9 %	9 %
lufact2	50	55 797 568	25 %	17 %	39 %
sor	30	224 896 104	16 %	11 %	20 %
sparse	29	24 234 710	19 %	20 %	24 %

Table 5.3 lists statistics about the static and dynamic prefetches. The static prefetch value is the number of prefetch instructions that the compiler generates. We present the number of dynamic prefetch instructions executed, and the percentage of dynamic prefetches relative to the number of dynamic load instructions. The largest percentage of dynamic prefetches occurs in `matmult` because most of the execution time is spent in a short inner loop. The small number of prefetches in `heapsort` is one reason that prefetching is ineffective. It is difficult to generate effective prefetches in `heapsort` because the array access pattern is data dependent and irregular.

Table 5.3 also shows the bus utilization values with (P) and without prefetching (N). Prefetching does increase the bus utilization. In some cases, the utilization percentages double, but the bus utilization in these programs remains under 40% even with prefetching. The main reason for the utilization increase is that the execution time decreases substantially. Except for `fft`, prefetching uses the data brought into the cache effectively.

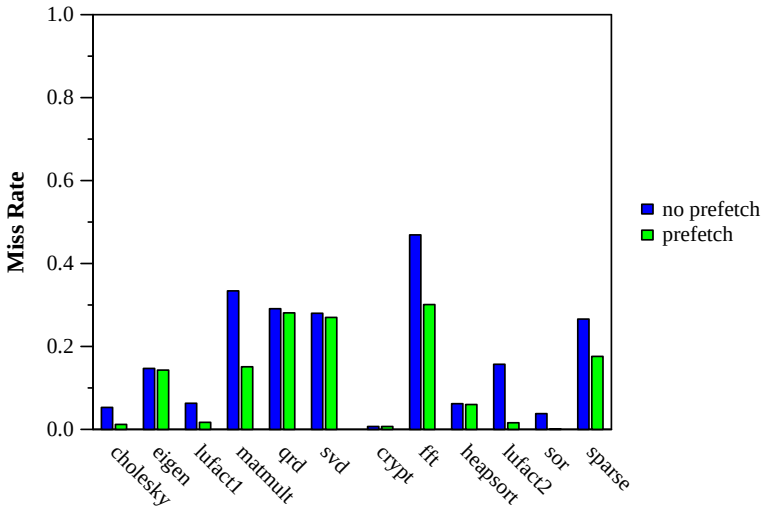


Figure 5.4. L1 Cache Miss Rate (Array Prefetching)

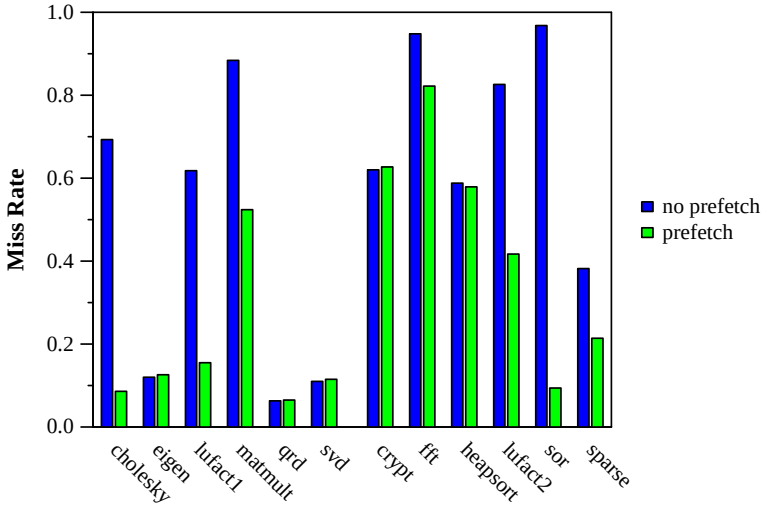


Figure 5.5. L2 Cache Miss Rate (Array Prefetching)

5.2.2 Cache Statistics

The cache miss rate is a useful metric to illustrate the benefits of prefetching. Figures 5.4 and 5.5 show the L1 and L2 cache miss rates, respectively. Effective prefetching improves the miss rate by moving data into the cache prior to the demand request.

The miss rates vary considerably in our benchmark programs. The L1 miss rates are much better than the L2 miss rates for most programs. When computing the L2 miss rate we count only the references that miss in the L1 cache. The number of references to the L2 cache is far less than the number of references to the L1 cache in most programs.

The L1 miss rate varies from almost 0% to just under 50%. The L2 miss rate varies from 6% to 98%. Over 50% of the references that miss in the L1 cache also miss in the L2 cache for several programs. It is possible to increase the cache sizes to improve the miss rates. We find that increasing the cache size tends to improve prefetching slightly, until the data fits in cache.

Prefetching effectiveness does not correspond to high or low miss rates. Prefetching improves or does not affect the L1 miss rate in each program. In the programs with the largest execution time improvements, we see significant miss rate reductions. Prefetching almost completely eliminates L1 cache misses in `sor` by reducing the miss rate from 38% to 1%. Prefetching improves the L1 miss rate in each program, and the L2 miss rate in most programs. The L2 miss rate is slightly worse in several programs because there are fewer L2 references, but the *percentage* of references that miss is higher.

Improving the miss rate does not always correspond to execution time improvements. We see a significant improvement in the L1 and L2 miss rates for `fft` even though prefetching increases the execution time. The problem is due to conflict misses, which we discuss in the next section.

5.2.3 Conflict Misses

Performance degrades by 13% in `fft` due to a large number of cache conflict misses. The implementation uses the radix-2 algorithm, which computes results in-place using a single dimension array. The size of the array, and the strides through the array are powers of two. For large arrays, the power of two stride values cause the conflict misses. Without prefetching, 7% of the read references cause conflict misses in the L1 cache, and 37% of the read references in the L2 cache cause conflict misses. Prefetching exacerbates the problem by increasing the number of conflict misses. With prefetching, 8% and 34% of the read references cause conflict misses in the L1 and L2 cache, respectively. Due to a power of two prefetch distance, the prefetches evict data that are prefetched in prior iterations.

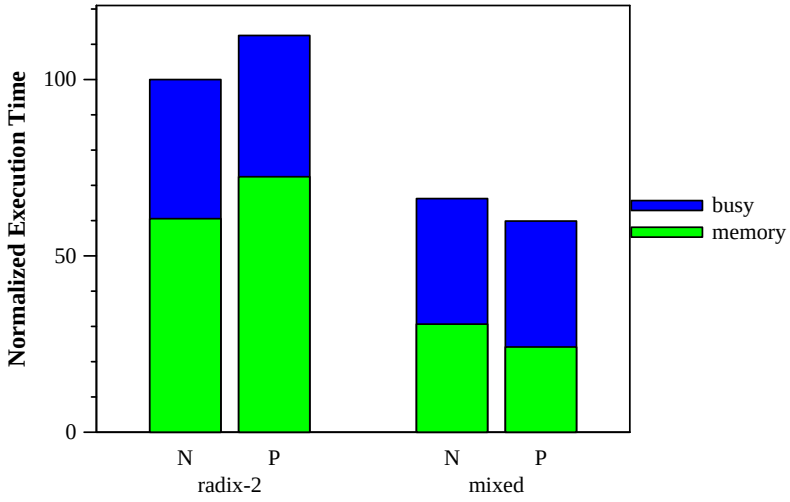


Figure 5.6. Comparing FFT Implementations

We also evaluate prefetching using a mixed radix implementation of FFT and compare the performance to the radix-2 implementation. The mixed radix version is a more complex algorithm which requires additional storage. We compare the two FFT versions in Figure 5.6. We present results with and without prefetching for the two implementations. We normalize each result in Figure 5.6 to `radix-2` with prefetching. The bars for `radix-2` on the left side are the same results shown in Figure 5.1. Our results show that `mixed` is 34% faster than `radix-2` and, furthermore, prefetching improves the performance of `mixed` by 10% over `mixed` without prefetching.

We made a change to the garbage collector to reduce the occurrence of conflict misses in two of the programs. Vortex uses the UMass Language-Independent Garbage Collector Toolkit for memory management [49]. The generational collector allocates memory in fixed-sized 64 KB blocks. Each generation may contain multiple blocks. Our collector contains a large object space for objects larger than 512 bytes. The initial large object space implementation allocated very large arrays in new blocks aligned on a 64 KB boundary. This allocation strategy results in many unnecessary conflict misses when programs access multiple large arrays at the same time. We fix the problem by adding a small number of pad

Table 5.4. Effect of Prefetch Distance on Prefetching (Execution Times Normalized to No Prefetching)

Program	Prefetch Distance			
	5	10	20	30
cholesky	52	48	48	49
eigen	101	102	102	102
lufact1	57	54	55	55
matmult	77	72	67	67
qrd	99	100	101	101
svd	101	101	101	101
crypt	100	100	100	100
fft	108	111	113	110
heapsort	100	100	100	100
lufact2	53	41	43	44
sor	58	54	54	54
sparse	96	85	85	85
Geom. Mean	80	77	77	77

bytes[2] to the beginning of each large object. The pad bytes eliminate conflict misses and result in improvements in `sparse` and `qrd`. Without the pad bytes, prefetching actually degrades performance by a few percent. The pad bytes do not help in `fft` because `fft` allocates a single array.

5.2.4 Varying the Prefetch Distance

In this section, we vary the prefetch distance to examine the impact on the results. By default, the compiler uses a prefetch distance of twenty elements. We run experiments using a prefetch distance of five, ten, and thirty elements. We show that computing a specific prefetch distance is not necessary because the results are similar across a range of short distances.

Table 5.4 summarizes the results by presenting the normalized execution times for each program. We normalize the execution times to those without prefetching. These results

[2]The number of pad bytes needs to be larger than the prefetch distance (in bytes). The collector increases the number of pad bytes for each large object and resets the pad byte value after allocating ten large objects.

show that the overall performance is stable across different prefetch distances. The only noticeable difference occurs when the prefetch distance is five elements.

Although the performance is stable when we aggregate the execution times, we see some differences among the individual programs. The largest difference occur when the prefetch distance is five elements. Prefetching is not as effective in `sparse`, `lufact1`, and `lufact2` when the prefetch distance is five elements. For example, the difference is 11% for `sparse` relative to the longer prefetch distances. The shorter prefetch distance results in more late prefetches, which has a large effect in some programs.

When prefetching degrades performance, the short prefetch distance reduces the prefetching penalty. For example, although prefetching hurts performance in `fft`, using a five element prefetch distance increases execution time by 8%, but using a twenty element prefetch distance increases execution time by 13%. Using the shorter prefetch distance reduces the number of conflict misses compared to the longer prefetch distance.

5.2.5 Case Study: Matrix Multiplication

In this section, we examine the effects of loop transformations and additional analyses on performance by applying loop unrolling, software pipelining, and read miss clustering [82] on matrix multiplication.

Figure 5.7 presents results for four versions of matrix multiplication with different code and data transformations on an out-of-order and an in-order processor. We provide results for each version with and without prefetching. We normalize all times to `original`, the Jama library version from Figure 5.1, without prefetching on either an out-of-order or in-order processor. We do not directly compare the execution time on the out-of-order processor to the in-order processor because the out-of-order processor is much faster. We perform the transformations by hand starting with the code in `original`.

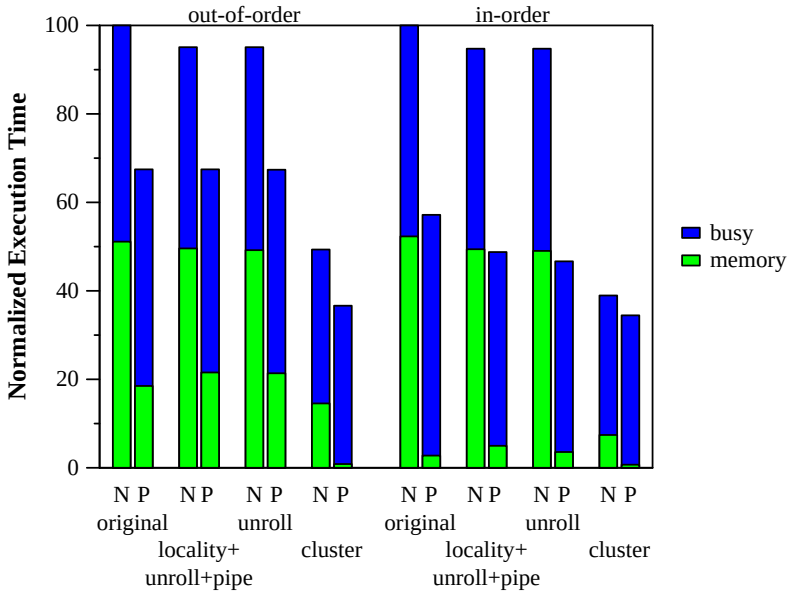


Figure 5.7. Applying Different Loop Transformations to Matrix Multiplication

We obtain the out-of-order results using the processor configuration parameters from Table 5.1. To obtain the in-order results, we simulate a single issue processor with blocking reads. The prefetch instructions do not stall the in-order processor.

We apply Mowry et al.'s [79] prefetch algorithm to matrix multiplication in `locality+unroll+pipe`. We present results for loop unrolling only in `unroll`. In `cluster`, we apply read miss clustering, which is a loop transformation that improves performance by increasing parallelism in the memory system [82].

Transforming `locality+unroll+pipe` requires several steps. We unroll the innermost loop four times to generate a single prefetch instruction for an entire cache line. We perform software pipelining on the innermost loop to begin prefetching the array data prior to the loop. We generate a prefetch for one of the arrays only. Matrix multiplication operates on the same portion of the second array during the two innermost loops. Thus, prefetching the second array results in many unnecessary prefetches.

Figure 5.7 shows that a state of the art prefetching algorithm does not provide much benefit beyond our simpler technique for matrix multiplication on the out-of-order processor. Without prefetching, both `locality+unroll+pipe` and `unroll` improve per-

formance by 5%. The execution time of `original` with prefetching is the same as the execution time after applying loop transformations, locality analysis, and prefetching. But, the transformations and locality analysis do improve prefetch effectiveness. Only 3% of prefetches in `locality+unroll+pipe` are unnecessary compared to 86% in `original`.

The loop transformations do have an impact on the in-order processor. In `original`, prefetching improves performance by 43%, which is larger than the performance improvement on the out-of-order processor. The better scheduling methods improve performance by an additional 18% over `original` with prefetching on the in-order processor. The improvement occurs because the locality analysis and loop transformations reduce the number of dynamic instructions. These results show that careful scheduling is more important on the in-order processor than the out-of-order processor for matrix multiplication.

A major bottleneck in `original` is a lack of memory parallelism, not an inability to prefetch the correct data. The lack of memory parallelism is evident by a large number of read instructions that stall the processor and stop other completed instructions from graduating in-order. Pai and Avde propose read miss clustering, which uses unroll-and-jam, to improve performance by increasing memory parallelism [82]. Read miss clustering groups together multiple misses in order to overlap their latencies. Clustering reduces execution time by 50% without any prefetching. Our prefetching method with clustering improves performance further, and almost eliminates all memory stalls. Pai and Adve also show that combining prefetching and clustering results in larger improvements than either technique alone [83]. The main disadvantage of clustering is that unroll-and-jam is difficult to implement, and requires dependence and other related analysis.

Our results suggest that advanced prefetch algorithms are not necessary to achieve benefits from prefetching on modern processors. In a superscalar, out-of-order processor, the cost of checking if data is already in the cache is cheap. The additional functional units and out-of-order execution hide the effects of issuing unnecessary prefetches. Loop transforma-

tions can reduce the number of unnecessary prefetches, but the resulting performance gain may be negligible. Furthermore, transformations may not be possible due to exceptions or inexact array analysis information. When loop transformations are possible, we strongly suggest implementing read miss clustering to improve memory parallelism.

5.2.6 True Multidimensional Arrays

Java treats arrays as objects. The elements of an array can be a primitive type, *e.g.*, `float`, or a reference type, *e.g.*, `Object`. Java implements multidimensional arrays as arrays-of-arrays, unlike languages such as Fortran that implement true multidimensional arrays. Java allocates each array dimension separately, so there is no guarantee that memory allocator allocates a contiguous region of memory for the entire array. True multidimensional arrays allocate a single contiguous region of memory for the entire array. Using a single contiguous region of memory simplifies compile-time analysis and optimization because the compiler can compute the address of any element relative the start of the array. The array specification in Java makes it challenging to apply existing array analysis and loop transformations.

In this section, we examine the performance of prefetching on true multidimensional arrays. We simulate true multidimensional arrays using a single array with explicit index expressions. We also use the *multiarray* package from IBM, which is a Java class that contains an implementation of true multidimensional arrays [76]. The underlying structure is a one dimensional array, and the class provides methods that mimic Fortran style array operations. IBM is attempting get the package included in the standard Java library. We compare the performance of standard Java arrays, simulated true multidimensional arrays, and the IBM multiarrays using matrix multiplication.

We can simulate a true multidimensional array by allocating a single array and using explicit index expressions to treat the array as a multidimensional array. Figure 5.8 shows the implementation of matrix multiplication when we implement a two-dimensional array

```
for (int i=0; i<rows; i++) {
  for (int j=0; j<cols; j++) {
    double val = 0;
    for (int k=0; k<cols; k++) {
      val += m1[i*cols+k] * m2[k*cols+j];
    }
    m3[i*cols+j] = val;
  }
}
```

Figure 5.8. Matrix Multiplication With a Single Array

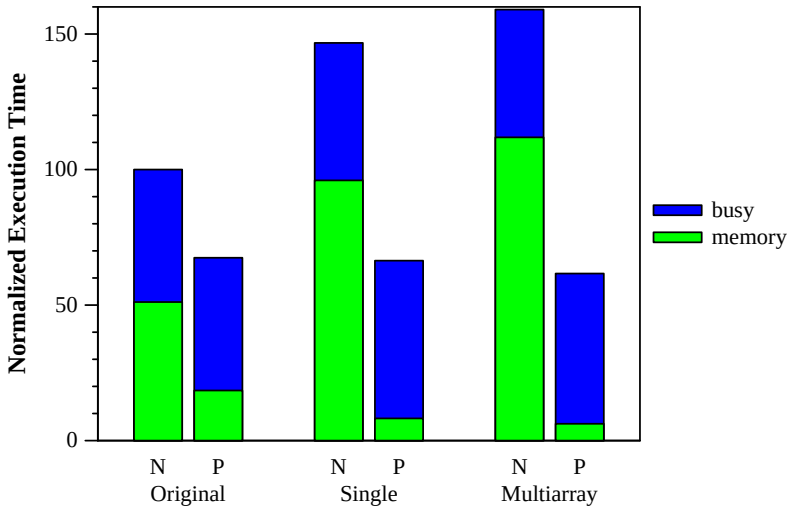


Figure 5.9. Performance of Prefetching on True Multidimensional Arrays

using a single dimension. The code multiplies array `m1` by array `m2` and places the result in array `m3`. The array index expression `m1[i*cols+k]` is equivalent to the expression `m1[i][k]`.

Figure 5.9 shows the results of prefetching on the standard array representation, the simulated true multidimensional array representation, and the multiarray representation for matrix multiplication. We normalize all times to `Original`, the Jama library version from Figure 5.1. The `Single` version uses a single array with explicit addressing to simulate a two dimensional array, and the `Multiarray` version uses IBM's multiarray package. The performance of `Single` and `Multiarray` without prefetching is 46% and 59% worse than the performance of `Original` without prefetching. One reason is that the

programmer is able to hoist the loop invariant address expressions out of the inner loop in `Original`. `Multiarray` has an additional cost because of more method calls and object allocations. Figure 5.9 shows that our array prefetching algorithm is able to discover the complex loop induction expression and insert effective prefetch instructions. The performance of `Single` with prefetching is slightly better than the performance of `Original` with prefetching by 1%. Prefetching improves the performance of `Multiarray` further. The performance of `Single` and `Multiarray` with prefetching is better than `Original` with prefetching because there are less late prefetches. Since `Single` and `Multiarray` perform more work in the inner loop, there is more time for the prefetches to bring data into the L1 cache.

5.2.7 Additional Prefetch Opportunities

In this section, we illustrate that our analysis and prefetching techniques work on other program idioms. Although these idioms do not appear in any of our benchmarks, we believe they are useful in Java programs.

5.2.7.1 Arrays of Objects

Java allows arrays of references as well as arrays of primitive types such as `double`. In an array of references, the array element contains a pointer to another object instead of the actual data. As we describe in Section 4.1, our compiler generates prefetches for array element value and referent object. The prefetch distance for the array element is twice as much as the prefetch distance for the array element referent object.

Since none of the array benchmarks use arrays of objects, we changed the Jama version of matrix multiplication to use `Complex` objects instead of `double` values. The `Complex` object contains two fields, which represent the real and imaginary parts of a complex number.

Figure 5.10 shows the results for matrix multiplication with complex numbers. We generate results for no prefetching, prefetching just the array elements, and prefetching both

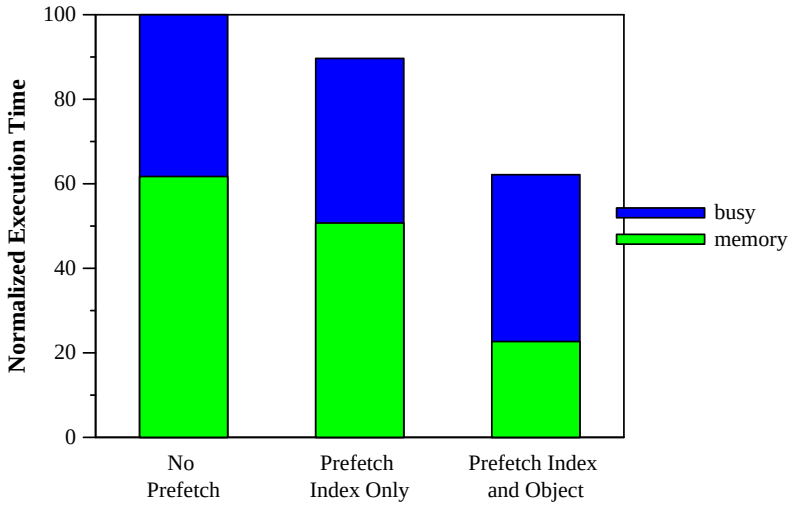


Figure 5.10. Prefetching Arrays of Objects

the elements and the referent objects. Prefetching just the array elements reduces execution time by just 10%. When we prefetch both the array elements and the referent objects, prefetching reduces execution time by almost 38%. This large improvement occurs even though we increase the number of instructions by issuing two prefetches and an additional load for each array reference.

5.2.7.2 Enumeration Class

The `Enumeration` class is a convenient mechanism for encapsulating iteration over a data structure. For example, programmers use the `Enumeration` class to iterate over elements in a `Vector`. Figure 5.11 illustrates the use of the `Enumeration` class. We also show the version after the compiler performs inlining and inserts a prefetch.

Our induction variable algorithm detects that `e.count` is an induction variable even though `e.count` is an object field and not just a simple variable. We discuss the analysis extensions for object fields in Section 3.3.5. Once the analysis detects the induction variable in an object field, the array prefetch algorithm generates a prefetch for a reference that contains the object field in the index expression.

```
Enumeration e=o.elements();
while (e.hasMoreElements()) {
  Element m = (Element)e.nextElement();
  sum += m.value();
}
// Inlined Enumeration for a Vector object
VectorEnumerator e;
e.vector=o;
e.count=0;
while (e.count < e.vector.elementCount)
  prefetch &e.vector.elementData[e.count + d];
  Element m = e.vector.elementData[e.count];
  e.count = e.count + 1;
  sum += m.value();
}
```

Figure 5.11. Using the `Enumeration` Class

5.3 Linked-Structure Prefetching

In this section, we evaluate the performance of prefetching linked data structures using greedy, jump-pointer, and stride prefetching. We first present the results for the three schemes separately. We summarize the results and we discuss the advantages of each scheme.

We evaluate prefetching using a Java version of the Olden benchmarks written in an object-oriented style [17]. Other researchers use the C version of the Olden suite to evaluate optimizations for pointer-based programs [25, 70, 90]. Tables 5.5 and 5.6 list the benchmarks we use in our experiments along with some characteristics of each program. The lines of code (LOC) number excludes comments and blank lines.

We present the results in this section with garbage collection disabled. The programs allocate memory, but the garbage collector never frees objects that the program no longer references. We run experiments in this manner because garbage collection affects the results, but our prefetching schemes do not target the allocation portion of programs. Furthermore, different garbage collection algorithms and heap sizes have a significant impact on performance. By disabling garbage collection initially, we are able to see the impact of pre-

Table 5.5. Linked-Structure Benchmark Suite

Name	Main Data Structure(s)	Inputs
bh	oct-tree, linked list	4096 bodies, 2 iters.
bisort	binary tree	100,000 numbers
em3d	linked list	2000 nodes, 100 degree, 4 iters.
health	quad-tree, linked list	5 levels, 500 iters.
mst	hashtable	1024 nodes
perimeter	quad-tree	4K x 4K image
power	tree	10K customers
treeadd	binary tree	20 levels
tsp	binary tree, linked list	60,000 cities
voronoi	binary tree	20,000 points

Table 5.6. Benchmark Program Statistics

Name	LOC	Methods	Inst. Issued	Total Memory	Max. Live	Bytes /Obj.
bh	487	74	731 M	76 MB	1.3 MB	28
bisort	164	14	1292 M	1.5 MB	1.5 MB	24.1
em3d	182	22	2120 M	6.5 MB	0.58MB	413
health	279	36	366 M	22 MB	2.6 MB	19.4
mst	183	26	955 M	41 MB	40 MB	25.9
perimeter	242	47	188 M	3.5 MB	3.5 MB	32
power	347	30	2086 M	24 MB	0.7 MB	32
treeadd	81	11	168 M	24 MB	24 MB	24
tsp	289	15	787 M	7 MB	3 MB	37
voronoi	526	70	848 M	68 MB	27 MB	25

fetching more easily. We discuss issues of prefetching with garbage collection, including results, in Chapter 6.

5.3.1 Greedy Prefetching

Figure 5.12 shows the results of greedy (G) prefetching. We normalize the results to those without prefetching (N). We divide execution time into the amount of time spent waiting for memory requests, and the amount of time the processor is busy using the methodology described in Section 5.1.

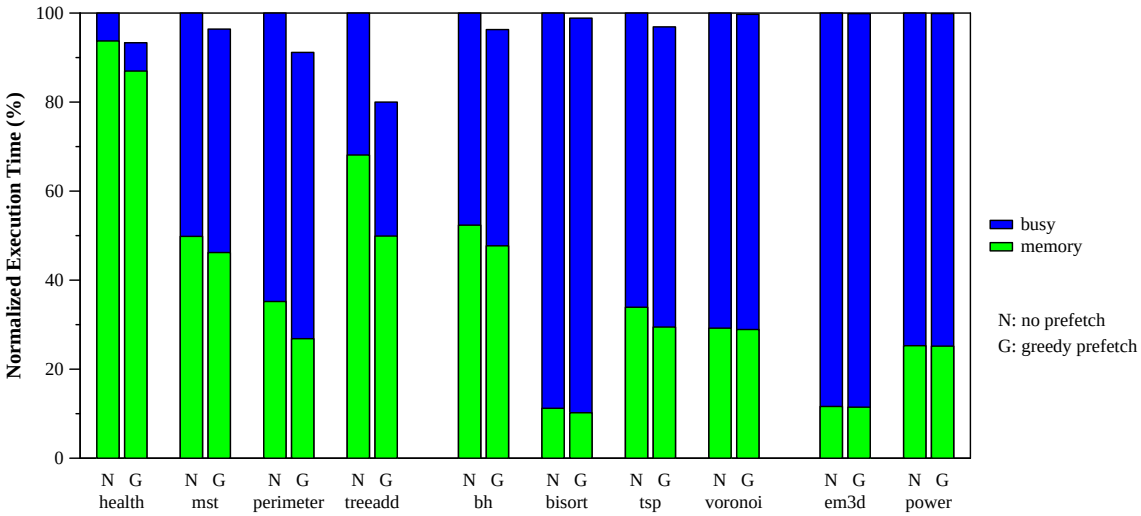


Figure 5.12. Greedy Prefetch Performance

In Figure 5.12, we group the Olden programs into three sets based upon our results. We see the largest improvements in the first set of four programs, smaller improvements in the second four programs, and the final two programs do not contain significant linked structure accesses. In the programs, greedy prefetching improves performance by as much as 20% in `treeadd`. Across all benchmarks, we see improvements of 5% using the geometric mean.

Figure 5.12 shows execution times for the entire program from start to finish. Several of the programs divide the execution time into creation phases and traversal phases. Greedy prefetching inserts prefetch instructions only in traversals. Greedy prefetching does not add prefetch instructions during the creation phase, unless the program traverses the linked structure while it is being created. In Figure 5.13, we show the performance of greedy prefetching in the traversal phase of five of the programs that contain separate creation and traversal phases. Above each result in Figure 5.13, we list the percentage of time the program spends in the traversal phase. For example, `mst` spends 50% of its time creating a binary tree, and 50% of its time traversing the tree.

Greedy prefetching reduces the execution time in the traversal phase of the programs in Figure 5.13 by a geometric mean of 12%. The largest decrease in execution time is 26%

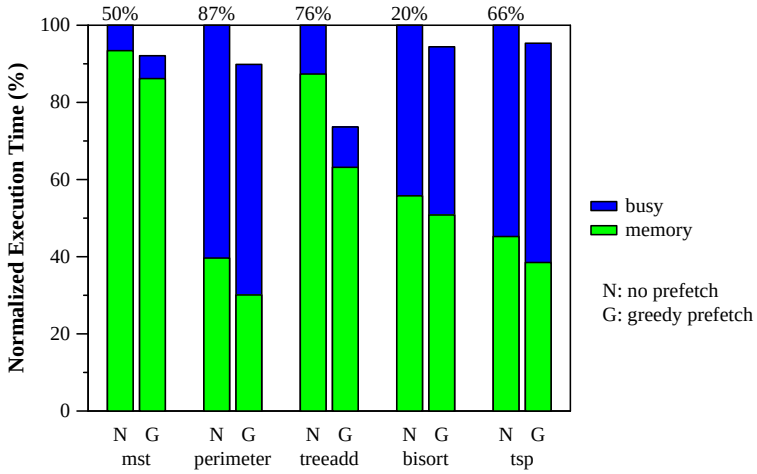


Figure 5.13. Traversal Phase Performance (Greedy Prefetching)

in `treeadd`. Figure 5.13 also shows that the memory performance is much worse in the traversal phase than in the rest of the program.

We present prefetch effectiveness and cache statistics in the following two sections. Then, we discuss the performance of the individual programs in more detail.

5.3.1.1 Prefetch Effectiveness

Figure 5.14 categorizes the dynamic prefetches as useful, late, early, and unnecessary for the L1 cache. We discuss the meaning of the different types of prefetches in Section 2.2.1.1.

Figure 5.14 shows that the percentage of *useful* prefetches does not have to be large to improve performance. Except for one program, the percentage of useful prefetches is less than 35%. The percentage of *late* prefetches illustrates why the potential of greedy prefetching is limited. Several of the programs contain a large number of late prefetches. Since greedy prefetching is able to prefetch only directly connected objects, reducing the number of late prefetches is difficult. Most of the programs do not have a large number of *early* prefetches. In general, greedy prefetching generates *early* prefetches only in programs that search large *n*-ary trees, *e.g.*, `bh`. Several programs have a large number of *unnecessary* prefetches, but they tend not to hurt performance. The cost of checking if a

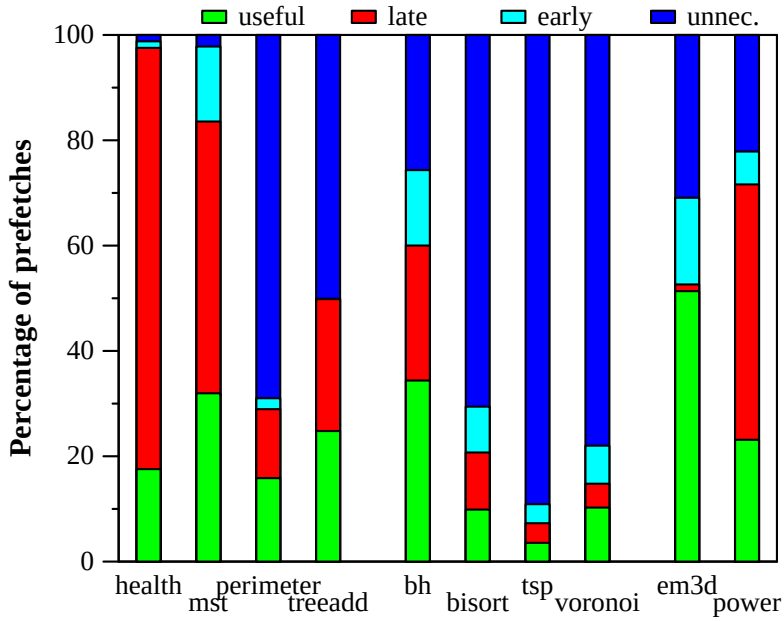


Figure 5.14. Greedy Prefetch Effectiveness

cache line is already in the L1 cache is cheap, and the additional functional units in the processor are able to hide the cost of an unnecessary prefetch.

Table 5.7 lists the number of static and dynamic prefetches that our benchmarks generate. The static value represents the number of compiler generated prefetches. The dynamic value is the number of prefetches issued at run time. We also list the number of prefetches as a percentage of the read instructions. A low percentage suggests that the prefetch algorithm is not able to identify opportunities very well. Finally, the third value is the percentage increase in bus bandwidth due to prefetching.

Table 5.7 shows that the number of static and dynamic prefetches does not need to be large to achieve improvements. To be effective, the prefetch algorithms must identify a few key places to insert prefetch instructions. The dynamic prefetching counts show why prefetching is ineffective on `em3d` and `power`; these programs do not access linked structures frequently. Although bus traffic increases due to prefetching, the maximum bus utilization with and without prefetching is 31% and 26%, respectively. The largest increase in bandwidth is 6% for `treeadd`. In general, an increase in the bandwidth is due to a decrease in run time, and not from prefetching useless data.

Table 5.7. Greedy Static and Dynamic Prefetch Statistics

Program	Static	Dynamic		Bus Utilization	
		total prefetches	prefetches/read	N	P
health	18	9 870 906	20 %	26 %	28 %
mst	6	2 879 206	10 %	15 %	16 %
perimeter	17	718 768	2.7 %	7 %	8 %
treeadd	2	856 600	10 %	25 %	31 %
bh	45	1 701 207	1.3 %	16 %	17 %
bisort	10	3 297 458	14 %	2 %	3 %
tsp	26	12 115 480	19 %	3 %	4 %
voronoi	15	522 169	6.3 %	9 %	9 %
em3d	24	55 861	0.06 %	4 %	4 %
power	4	44 957	0.01 %	2 %	2 %

5.3.1.2 Cache Statistics

The cache miss rate is a measure of the effectiveness of the caches at execution time. Effective prefetching reduces the miss rate by moving data into the cache prior to the demand request. Figures 5.15 and 5.16 show the L1 and L2 cache miss statistics, respectively.

Greedy prefetching improves the L1 miss rate for 5 of the programs, and the L2 miss rate for 9 of the programs. In a processor that allows outstanding loads, not all misses are equal. Some of the references that miss in the cache, hit in the MSHR, and do not need to pay the entire cost of going to the L2 cache. A reference hits in the MSHR when a prior miss causes the memory subsystem to transfer the reference to the cache. Although the miss rate increases for `health`, the percentage of MSHR hits increases from 16% to 33%. Thus, with greedy prefetching many of the misses are in the process of moving into the L1 cache. The cache statistics mirror the overall results; when prefetching improves overall performance, the number of cache hits and coalesced hits increase as well.

5.3.1.3 Analysis Features

Table 5.8 shows the features of our recurrence analysis that are responsible for generating static prefetch instructions in our greedy prefetching scheme. We show the contribution from interprocedural analysis (*IP*), analyzing stores into fields (*F*), and intraproce-

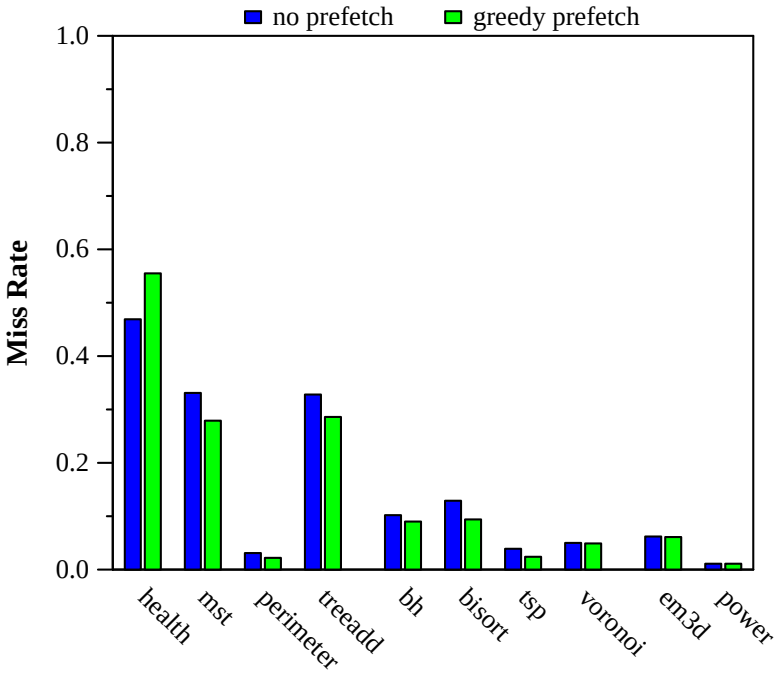


Figure 5.15. L1 Miss Rate (Greedy Prefetching)

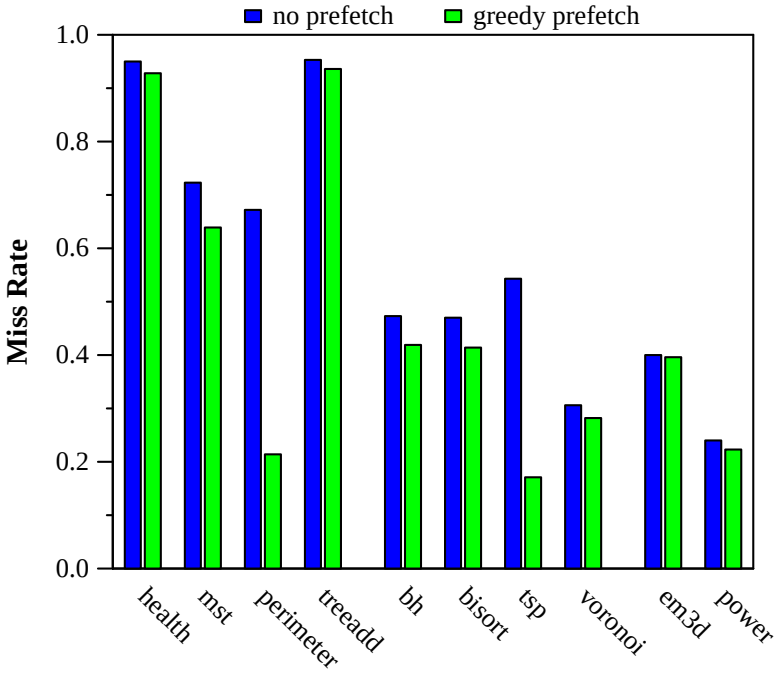


Figure 5.16. L2 Miss Rate (Greedy Prefetching)

Table 5.8. Static Greedy Prefetch Statistics

Program	IP M	IP P	Fields (F)	Intra (I)	Inline	Array size	Total
health	12		5	1	5 (F)	4 (IP)	18
mst	6				4 (I)		6
perimeter	9	8					17
treeadd	2						2
bh	22	9	5		5 (F)	8 (IP)	36[a]
bisort	6			4			10
tsp	4		7	6			17[a]
voronoi	9		1		9 (I)		10[a]
em3d	2		10		10 (F)		12[a]
power	4						4

[a] These values differ from those in Table 5.7. In this table, we do not consider the size of objects and the cache line size.

dural analysis only (*I*). The total number of greedy prefetches is *IP+F+I*. Interprocedural class analysis divides the IP results into monomorphic (*M*) and polymorphic (*P*) recursive method calls. The *Inline* column shows how inlining affects our recurrent analysis. For example, in `health` we generate the five prefetches in the *Fields* column only when inlining is enabled and interprocedural analysis is disabled. In `mst`, the compiler still generates the prefetches if we perform intraprocedural analysis only, and inlining is enabled. The *Array size* column shows the results of our analysis for computing the array sizes. Only two of the programs, `health` and `bh`, use constant size arrays to represent *n*-ary trees.

Table 5.8 shows that both interprocedural analysis and the extensions for propagating the data flow into fields fields are important in our Java programs. Intraprocedural analysis rarely discovers recurrent accesses on its own. In `health`, most of the improvement comes from analyzing field stores. When we disable field stores, performance improves by less than 1% instead of 7%. In `perimeter`, the prefetches in the monomorphic and polymorphic recursive calls contribute 2% and 4%, respectively.

5.3.1.4 Individual Program Performance

We describe the effect of greedy prefetching on the performance of each Olden benchmark below.

`health` Greedy prefetching improves `health` by almost 7%. The improvement is not surprising considering its poor locality. `Health` spends 94% of its time waiting for memory while traversing very long singly linked lists. There is significant room for improvement, but greedy prefetching is limited because it can only prefetch the next element in the list. The limiting factor is that there is not enough computation on each node in the list to tolerate the memory latency.

`mst` Improving the performance of `mst` using prefetching is difficult because the linked structure is a hashtable. Because the hashtable accesses a different element during each probe, it is difficult to predict what to prefetch. In `mst`, the hashtable is small and each hash entry contains a list of objects. Because the hashtable is small, each probe of the hashtable results in a linked list traversal. The traversal often accesses only a few elements.

Greedy prefetching improves performance by 4%. Figure 5.13 shows that greedy prefetching improves the traversal phase by 8%. Figure 5.14 shows that over 80% of the prefetches are either useful or late, which is why performance improves. However, `mst` has a noticeable number of early prefetches as well. Early prefetches can be very harmful to performance by evicting useful data. In `mst`, the early prefetches occur because the list traversals are conditional, and may end before reaching the end of the list. When this occurs, the last prefetch is early because the object is not referenced.

`perimeter` The main data structure in `perimeter` is a quad tree. After creating the tree, `perimeter` traverses the tree twice. The first traversal is a simple depth-first pass over all the nodes in the quad tree, which counts the number of leaves in the tree.

The second traversal is a directed pass over the quad tree to compute the perimeter of the image that the tree represents. The traversal actions to compute the perimeter depend upon the node type in the quad tree.

Prefetching improves the performance of `perimeter` by 9%. When we consider the traversal phase only, the improvement is 11%. Prefetching improves the performance of the initial pass over the quad tree by 33%, but counting the leaf nodes is very fast and does not contribute to the overall execution time significantly.

We obtain performance improvements even though most of the prefetches hit in the L1 cache. Only 3% of the memory references result in cache misses. Prefetching reduces the miss rate to 2%. Figure 5.16 shows a large reduction in the L2 miss rate from 67% to 21%.

`treeadd` `Treeadd` shows the largest performance improvement from greedy prefetching. `Treeadd` is a very simple program that creates a binary tree and traverses the tree in a depth first manner. A binary tree is the model data structure for illustrating the potential of greedy prefetching. Greedy prefetching generates two prefetch instructions for each node, one for the left child and one for the right child. We expect to hide the latency of one of the prefetches only partially, and to hide the latency of the other prefetch completely. If the tree is too large, then we may evict prefetched data. Early prefetches are rare because they will occur only towards the root of the tree.

Greedy prefetching reduces the whole program execution time by 20%, and the traversal phase execution time by 26%. Figure 5.14 show that 50% of the prefetches are either useful or late, and 50% hit in the L1 cache. The percentage of useful and late prefetches is 25% each, which we expect since, at each node, prefetching only partially hides the latency of one prefetch and completely hides the latency of the other prefetch. We see a large number of unnecessary prefetches because each node

in the tree is smaller than a cache line, so each prefetch instruction brings two tree nodes into the L1 cache. At the leaf nodes, the processor drops the prefetch instructions because address to prefetch is zero.

bh Greedy prefetching slightly improves performance by 4%. The main linked structure in `bh` is an Oct-tree. The internal nodes in the tree are larger than a cache line. The program contains a couple of methods that traverse the Oct-tree. One of the methods, `walkSubTree`, conditionally traverses the children. If the compiler generates the prefetch instructions prior to checking the condition, then the performance of `bh` degrades. The improvement occurs only when the compiler inserts the prefetches after the condition.

bisort `Bisort` performs a bitonic sort. The main data structure is a binary tree, which the program updates while traversing. `Bisort` first sorts, and traverses, the binary tree in one direction, and then sorts the tree in the opposite direction.

Greedy prefetching only slightly improves the performance of `bisort` by 1.2%. A limiting factor is that the access pattern is data dependent. The prefetching effectiveness values illustrate the problems that data dependent traversals pose to greedy prefetching. In `bisort`, 10% of prefetches are useful, 10% are late, 9% are early, and the rest are unnecessary. Greedy prefetching is able to find prefetching opportunities, but the data dependent traversal results in many early prefetches. Even without the data dependent traversal, obtaining further improvements in `bisort` is difficult because the memory stall percentage is low.

tsp Greedy prefetching reduces the execution time in `tsp` by 3%. `Tsp` creates a binary tree with linked lists between the nodes to represent the cities to visit. The performance improvement is due to the prefetch instructions at the linked list traversals.

The number of unnecessary prefetches is high because the miss rate in this program is low. There is not much room for improving the L1 miss rate. Prefetching reduces

the L2 miss rate from 54% to 17%. The rest of the prefetches are equally divided among useful, late, and early at approximately 3.5% each. The reason is that the linked lists tend to be short.

`voronoi` The reduction in execution time is negligible. The main data structure is a small binary tree that represents a diagram. `Voronoi` also contains a quad-tree, which represents the edges in the diagram. `Voronoi` links the edges using a linked list. The number of dynamic prefetches in `voronoi` is small. The ratio between prefetch and read instructions is less than 7%. In `voronoi`, 79% of the dynamic prefetches are unnecessary. One reason for the large number of unnecessary prefetches is the low miss rate. Only 5% of the cache references miss in the L1 cache.

`em3d` `Em3d` does not suffer from poor memory performance. Only 12% of the execution time is spent waiting for memory. The L1 miss rate is only 6%. Because memory performance is good, greedy prefetching has little effect on `em3d`. Another factor is that the linked structure traversals do not contribute to the misses significantly. Table 5.7 shows that the number of dynamic prefetch instructions is very small relative to the number of read instructions.

`power` Greedy prefetching does not affect the execution time of `power`. Although the main data structure is a tree, the time spent on memory operations is not due to linked structure traversals. `Power` performs a significant amount of floating point computation. Figure 5.15 shows that the L1 miss rate is 1%.

5.3.2 Jump-Pointer Prefetching

Figure 5.17 shows the results of jump-pointer (J) prefetching. We divide execution time into the amount of time spent waiting for memory requests and the amount of time the processor is busy using the methodology described in Section 5.1. We normalize the results to those without prefetching (N). We use a prefetch distance of eight objects by

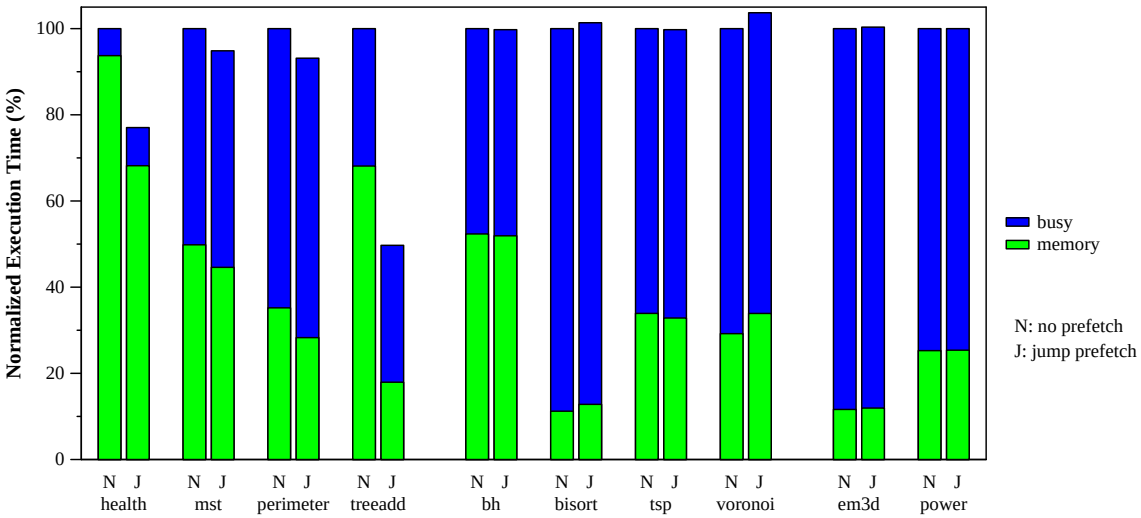


Figure 5.17. Jump-Pointer Prefetching Performance

default, except for `mst` which uses a distance of two. In most of the programs, using a distance value greater than eight does not improve performance significantly. The compiler could compute this distance based upon the number of instructions between accesses, but we have not implemented this cost model.

Jump-pointer prefetching improves the execution time in four of the ten programs. Across all benchmarks, we reduce the execution time by a geometric mean of 10%. Prefetching reduces the execution time in `health`, `mst`, `perimeter`, and `treeadd`. In these four programs, the execution time reduction ranges from 5% in `mst` to 50% in `treeadd` for an improvement of a geometric mean of 24%. Prefetching has little effect in four other programs. Prefetching increases the execution time in `bisort` by 1% and `voronoi` by 3%.

Figure 5.18 shows the performance of jump-pointer prefetching in the traversal phase of the five programs that contain separate creation and traversal phases. Our prefetch algorithms insert prefetch instructions only in linked-structure traversals, so we demonstrate the full effect of prefetching by isolating the traversal phase from the creation phase. Figure 5.13 presents a similar graph for greedy prefetching. An important difference between the two results is that jump-pointer prefetching increases the execution time in one of the

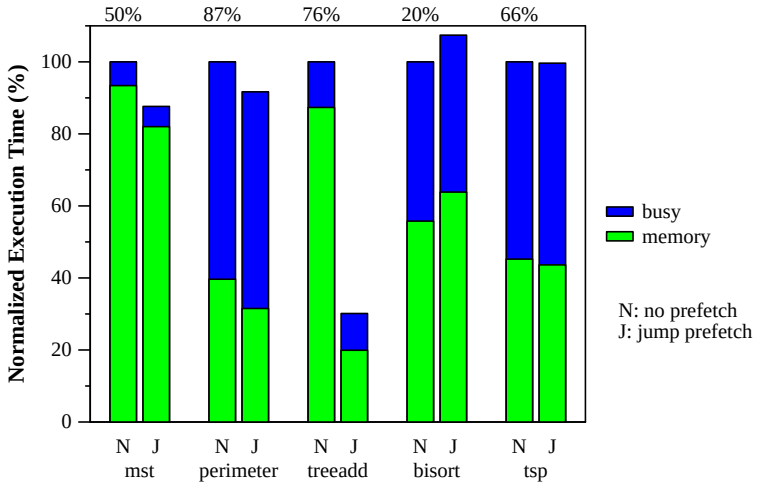


Figure 5.18. Traversal Phase Performance (Jump-Pointer Prefetching)

programs. Above each result in Figure 5.18 we list the percentage of time the program spends in the traversal phase.

Jump-pointer prefetching reduces the execution time of the traversal phase by a geometric mean of 24%. When we consider the whole program time, the reduction in execution time is just 15%. The two programs with the most significant execution time improvements are `mst` and `treeadd`. Prefetching reduces the execution time of the traversal phase of `treeadd` by 70%. When we consider the entire program, the improvement is 50%. Jump-pointer prefetching negatively affects the performance of `bisort`, so the performance degradation is even larger when we consider the traversal phase only.

We present prefetch effectiveness and cache statistics in the following two sections. Then, we discuss the performance of the individual programs in more detail.

5.3.2.1 Prefetch Effectiveness

Figure 5.14 categorizes the dynamic prefetches as useful, late, early, and unnecessary for the L1 cache. Section 2.2.1.1 describes the meaning of each type of prefetch.

In the four programs for which prefetching improves performance, the percentage of useful and late prefetches is high. In the programs that prefetching does not affect, the percentage of early prefetches is relatively high, especially when we compare the results

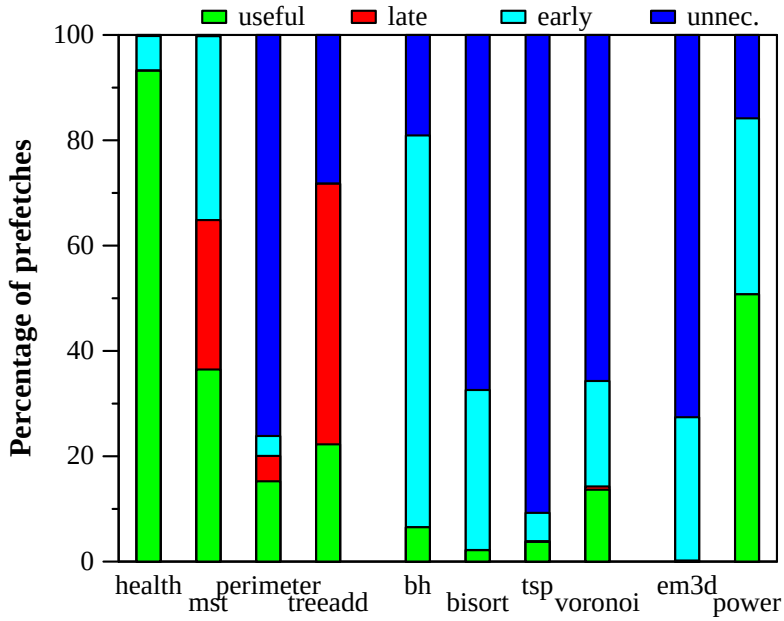


Figure 5.19. Jump-Pointer Prefetch Effectiveness

to greedy prefetching. The increase in the number of early prefetches is an important issue with jump-pointer prefetching. If the compiler is not careful when creating the jump-pointers, then the referent of the jump-pointer may not be a useful object to prefetch.

Table 5.9 provides information about the static and dynamic prefetches. The static values are the number of prefetch instructions that the compiler generates. The dynamic values are the number of prefetch instructions executed at run time. The number of compiler inserted prefetches for jump-pointer prefetching is typically less than the number inserted for greedy prefetching. Jump-pointer prefetching inserts one prefetch instruction for each linked structure, while greedy prefetching inserts a prefetch for each recurrent field in a linked structure. Jump-pointer prefetching results in fewer dynamic prefetches for the same reason.

5.3.2.2 Cache Statistics

Figures 5.20 and 5.21 present the cache miss rate statistics for the L1 and L2 caches, respectively. Jump-pointer prefetching reduces the L1 miss rate in six of the ten programs, and reduces the L2 miss rate in all of the programs. The largest L1 miss rate reduction is

Table 5.9. Jump-Pointer Prefetch Statistics

Program	Static	Dynamic		Bus Util.	
		total prefetches	prefetches/read	N	P
health	10	10 162 465	13 %	26 %	36 %
mst	6	2 370 368	7.4 %	15 %	18 %
perimeter	16	1 018 676	3.7 %	7 %	9 %
treeadd	1	786 663	7.8 %	25 %	50 %
bh	19	761 242	0.6 %	16 %	17 %
bisort	6	3 468 116	13 %	2 %	4 %
tsp	16	11 879 660	20 %	3 %	5 %
voronoi	14	500 779	6 %	9 %	11 %
em3d	22	21 056	0.02 %	4 %	4 %
power	4	52 400	0.01 %	2 %	2 %

18% in `health`, and we see large improvements in `treeadd` and `em3d`. The average miss rate reduction is 2% and 6% for the L1 and L2 caches, respectively.

In all four of the programs for which prefetching reduces the execution time, the L1 and L2 miss rates decrease in each of the programs. Figure 5.21 shows large reductions in the L2 miss rate. Prefetching reduces the miss rate in the four programs by an average of 4% and 26% for the L1 and L2 caches, respectively. These results predict the execution time improvements.

5.3.2.3 Individual program performance

`health` Jump-pointer prefetching reduces the execution time of `health` by 33%. The execution time improvement is a result of the flexibility of jump-pointers to tolerate more latency than greedy prefetching. Figure 5.19 shows that 93% of the prefetch instructions are useful, and there are no more late prefetches. `Health` contains a small percentage of early prefetches because the program often deletes and inserts objects in the linked lists. Figure 5.20 shows a larger reduction in the L1 miss rate with prefetching.

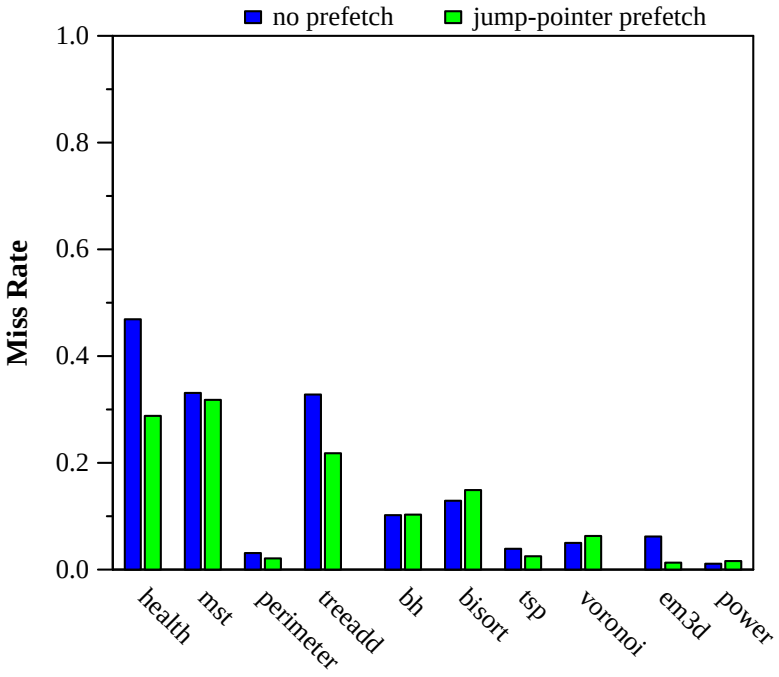


Figure 5.20. L1 Miss Rate (Jump-Pointer Prefetching)

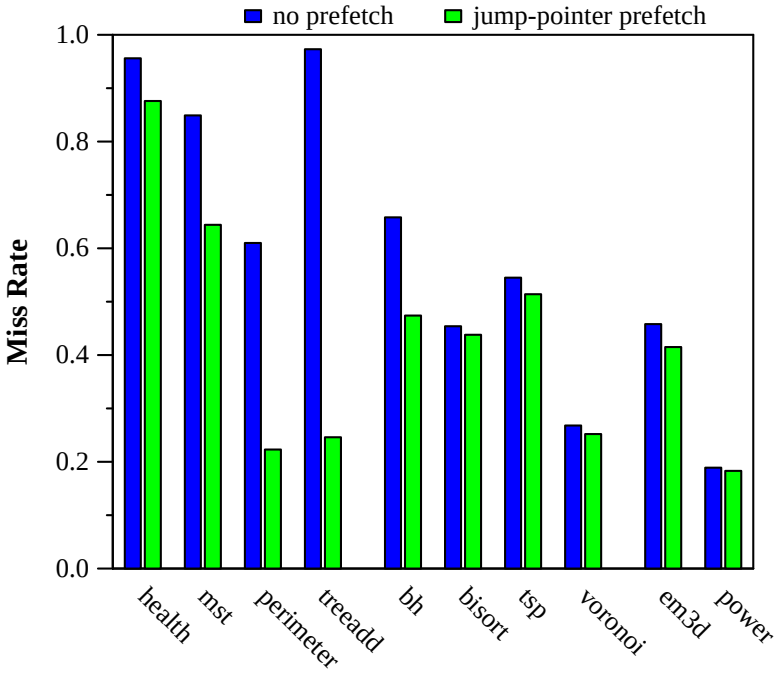


Figure 5.21. L2 Miss Rate (Jump-Pointer Prefetching)

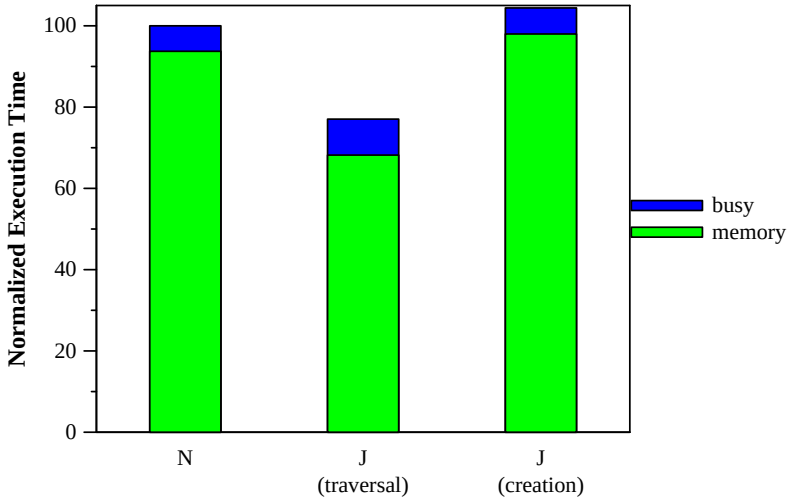


Figure 5.22. Different Versions of Health

`Health` is the only program that both builds jump-pointers while traversing its linked structures and contains indirect recurrent variables. The jump-pointers eliminate all of the late prefetches, but we do not see larger improvements due to the overhead of updating the jump-pointers at traversal time.

Figure 5.22 shows the results when the compiler creates jump-pointers at traversal time versus creation time. We normalize the times to those without prefetching. When `health` creates jump-pointers at the creation site, the execution time increases by 4%. When `health` creates the jump-pointers while traversing the linked structures, the execution time decreases by 23%. The reason for the difference in performance is that `health` frequently updates the linked structures at run time. This result shows that it is important that the compiler generate the jump-pointers appropriately.

`mst` Jump-pointer prefetching increases the number of useful prefetches, but the number of early prefetches also increases because the objects at the end of each list do not have useful jump-pointers.

We create the jump-pointers in the reverse direction because new elements are added to the beginning of each list. We must limit the jump-pointer prefetch distance to

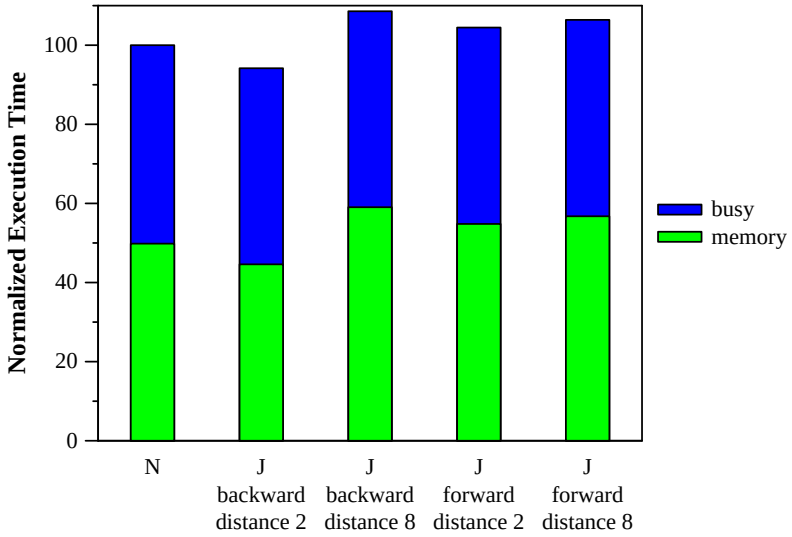


Figure 5.23. Different Versions of MST

see an improvement because each linked list is small. Figure 5.23 compares the performance of `mst` when the compiler creates the jump-pointers in the backward and forward directions.

`perimeter` Jump-pointer prefetching reduces the execution time by 7%. When we consider the traversal phase only, the execution time decreases by 10% with jump-pointers. The decrease in execution time is slightly less than with greedy prefetching. With jump-pointer prefetching, the percentage of useful and late prefetches is 15% and 5%, respectively. Greedy prefetching results in the same number of useful prefetches, but has more late prefetches. Jump-pointer prefetching is less effective because `perimeter` has data dependent traversals.

Adding the jump-pointer field increases the object size from 32 bytes to 40 bytes, so the compiler inserts two prefetch instructions instead of just one. Prefetching the extra cache line reduces the L2 hit rate relative to the greedy prefetching results. Prefetching the extra cache line does help; if we prefetch only one cache line, then performance improves only by 1% instead of 7%.

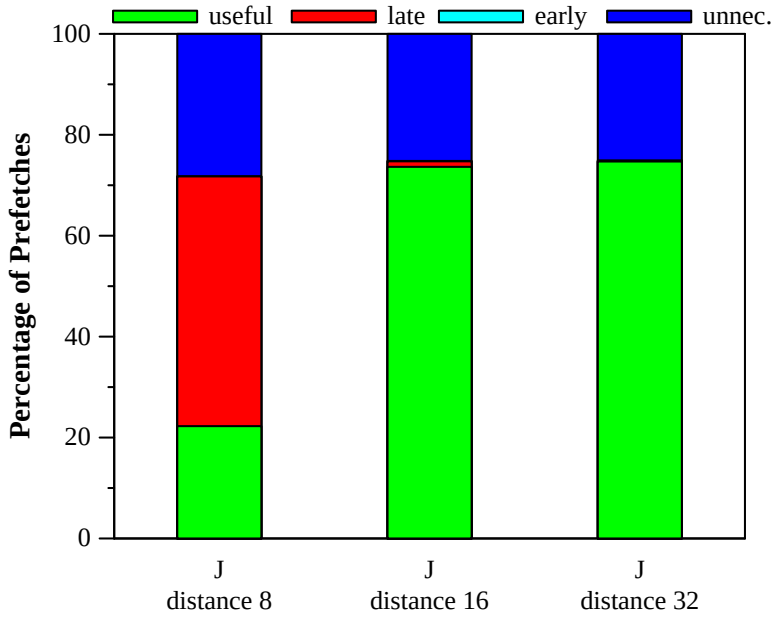


Figure 5.24. Prefetch Effectiveness in Treeadd

`treeadd` Jump-pointer prefetching reduces the execution time in `treeadd` by 50%. Jump-pointer prefetching is effective because `treeadd` creates and traverses the binary tree in the same order. The improvement is much larger than with greedy prefetching because jump-pointer prefetching is able to tolerate more latency. Figure 5.19 shows that jump-pointer prefetching contains a large percentage of useful and late prefetches. Figures 5.20 and 5.21 show large reductions in the miss rate, especially in the L2 cache where the miss rate decreases from 97% to 25%.

Figure 5.24 shows that increasing the prefetch distance almost completely eliminates the late prefetches in `treeadd`. We use a compile-time option to create jump pointers with a distance of sixteen and thirty two objects. When the prefetch distance is eight objects, 50% of the prefetches are late, and only 22% are useful. A distance of sixteen objects almost completely eliminates the late prefetches; only 1% of the prefetches are late. Doubling the prefetch distance again completely eliminates the late prefetches, but the number of early prefetches increases slightly to 2%.

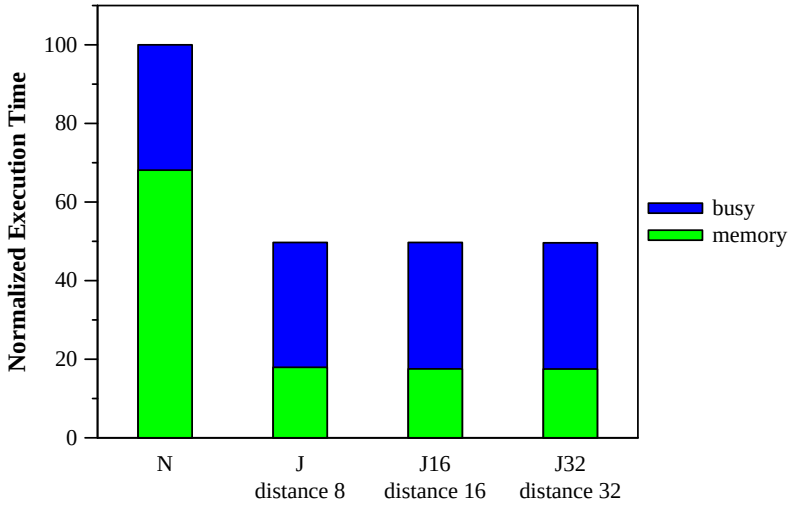


Figure 5.25. Varying Prefetch Distance in Treeadd

Increasing the prefetch distance from eight objects to sixteen reduces the L1 miss rate from 41% to just 2%. Unfortunately, Figure 5.25 shows that increasing the prefetching distance in `treeadd` from eight objects to sixteen and thirty two objects does not have an impact on execution time. The reason is that accessing the L1 cache becomes a bottleneck when we increase the prefetch distance, and many of the tree node objects are in cache when accessed.

`bh` Jump-pointer prefetching has very little effect on the performance of `bh`. In contrast, greedy prefetching causes the execution time to increase. One reason that performance does not degrade with jump-pointer is that the compiler generates less prefetches. Since `bh` uses an Oct-tree, greedy prefetch generates eight prefetches per node, but jump-pointer prefetching generates only one prefetch. Unfortunately, the traversal path is data dependent so the single prefetch instruction is often ineffective. As a result, Figure 5.19 shows that jump-pointer prefetching results in a large percentage of early prefetches.

`bisort` Jump-pointer prefetching results in a slight increase in execution time of 1%. `Bisort` traverses a binary tree two times, once in each direction, and updates the tree during each traversal. Because the traversal direction changes, and the tree struc-

ture changes at run time, the jump-pointers are often ineffective. Figure 5.19 shows that 30% of the prefetches are early, and only 2% are useful. Figure 5.20 shows that the ineffective prefetches cause the L1 miss rate to increase.

`tsp` Jump-pointer prefetching improves performance by less than 1%. Since the miss rate is so small in `tsp`, there is very little room for improvement. Due to the low miss rate, most of the prefetches hit in the L1 cache and are unnecessary. Because `tsp` has many short linked lists, the prefetches that do not hit in the L1 cache often prefetch objects that the program never references. These early prefetches are just 5% of all prefetches, but only 4% of the prefetches are useful.

`voronoi` Jump-pointer prefetching increases the execution time by almost 4%. The main linked structures in `Voronoi` are a binary tree and a quad-tree. Jump-pointer prefetching is able to insert prefetches for the binary tree effectively, but unable to insert effective prefetches for the quad-tree. The quad tree is a complex structure that uses an array of four objects to represent the children. The four children are connected in a ring using a linked list. Due to the data dependent traversal pattern on the quad-tree, the jump-pointers are unable to prefetch useful objects. With jump-pointers, the percentage of early prefetches is 11%, and the percentage of useful prefetches is only 5%. The rest of the prefetches are unnecessary. The early prefetches pollute the cache and cause the L1 miss rate to increase slightly by 1%.

`em3d`, `power` Similar to greedy prefetching, jump-pointer prefetching has little effect on performance for both `em3d` and `power`. As we mention in Section 5.3.1.4, there is very little room for improvement in either program because the linked structure accesses contribute very little to overall performance.

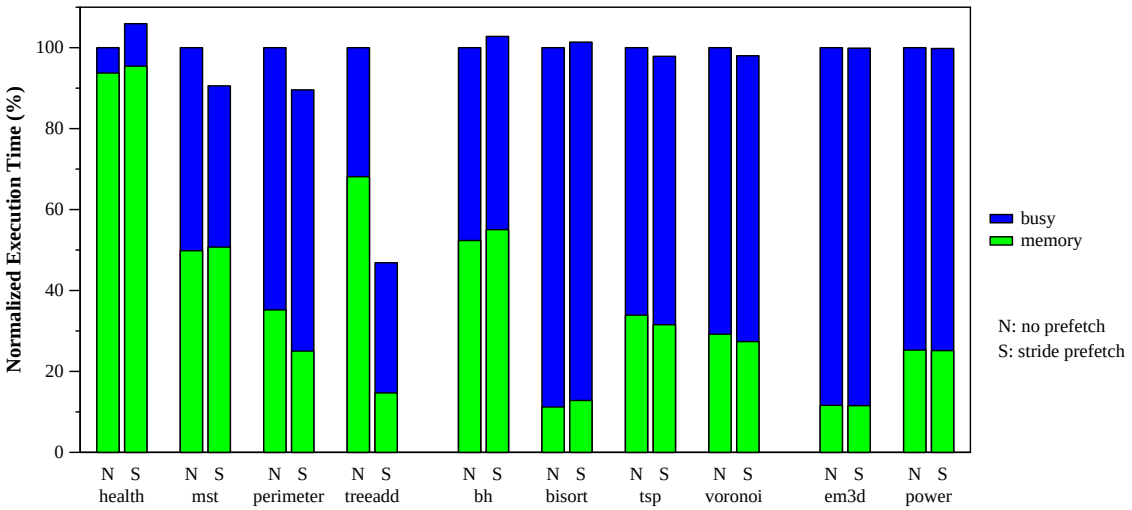


Figure 5.26. Stride Prefetching Performance

5.3.3 Stride Prefetching

Figure 5.26 shows the results of stride (S) prefetching. We normalize the results to those without prefetching (N). We divide execution time into the amount of time spent waiting for memory requests and the amount of time the processor is busy using the methodology described in Section 5.1. We use a stride distance of ten objects[3] by default, except for `mst` and `health` which use a stride distance equal to three objects.

The results in Figure 5.26 show that stride prefetching improves performance overall, but the results are inconsistent. Stride prefetching improves the performance of four of the ten programs, but degrades performance in four programs as well. Stride prefetching decreases the execution time of all the programs by a geometric mean of 9%. The largest improvement is 53% in `treeadd`.

Stride prefetching is most effective when the layout of the data matches the traversal direction. Chapter 6 shows that garbage collection can improve the performance of stride prefetching. Stride prefetching is not effective in programs that contain data dependent traversals or short linked structures. Since stride prefetching does not explicitly consider the

[3]The distance is 10 * the size of each object.

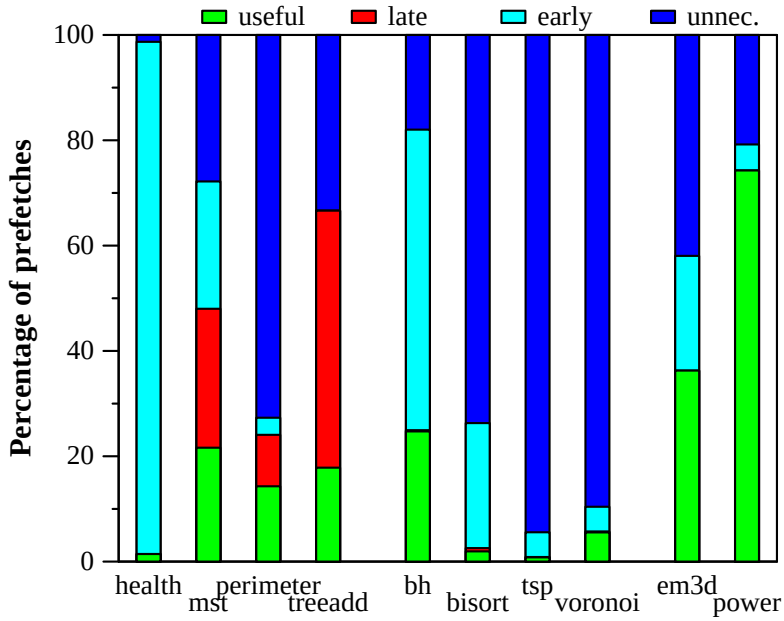


Figure 5.27. Stride Prefetch Effectiveness

relationship among objects in a linked structure, the prefetch instructions are ineffective if the linked structure is not allocated in the proper order. The data that the prefetch instruction brings in may not be useful, and performance suffers when this happens.

Figure 5.27 shows the prefetch effectiveness statistics for stride prefetching. In the programs for which stride prefetching degrades performance, we see a large percentage of early prefetches. The prefetch instructions in these programs do not bring in cache lines that are used again.

5.3.3.1 Individual Program Performance

Rather than discussing each program separately, we group the programs into sets that share similar performance characteristics.

`health`, `mst` Stride prefetching increases the execution time of `health` by 6%, and decreases the execution time of `mst` 9%. For both of these programs, the stride value is negative because the programs add new objects to the beginning of the linked structures. If the stride value is positive then prefetching increases the execution time in `mst`.

The execution time of `health` increases because most of the prefetches are ineffective. Figure 5.27 shows that 97% of the prefetches are early. One reason for the excessive number of early prefetches is that `health` frequently inserts and deletes objects from its linked structures. As the linked structures change over time, the effectiveness of stride prefetching decreases.

`perimeter`,`treeadd` Stride prefetching reduces the execution time of `perimeter` and `treeadd` by 10% and 53%, respectively. Both programs create and traverse a binary tree in the same order. The memory allocator lays out the tree node objects in consecutive locations, which enables the prefetch instructions to be effective. Stride prefetching is especially effective because the prefetch instructions are able to tolerate large latencies without the space and time cost of a jump-pointer. The space cost is an issue especially with `perimeter`. Recall that the jump-pointer increases the size of each object from 32 bytes to 36 bytes, which is larger than the cache line in our experiments.

`bh`, `bisort`, `tsp`, `voronoi` Stride prefetching either increases the execution time of these programs slightly, or has no effect on performance. The results are similar to jump-pointer prefetching. The difficulty is that these programs use data dependent traversals, have short linked structures, or low miss rates. Any of these characteristics make stride prefetching ineffective. Figure 5.27 shows that many prefetches in these programs are early or unnecessary.

`em3d`, `power` There is little room for improvement in these programs. Similar to the other linked-structure prefetching techniques, stride prefetching does not have any effect. Figure 5.27 shows that many of the prefetches are useful, but neither program issues many dynamic prefetch instructions.

5.3.4 Summary of Prefetching Linked Structures

Each linked structure prefetching technique produces noticeable improvements on four of the ten programs, *i.e.*, `health`, `mst`, `perimeter`, and `treeadd`, although stride prefetching is ineffective on `health`. Either greedy, jump-pointer, or stride prefetching slightly improves performance in `bh`, `bisort`, `tsp`, and `voronoi`. None of the linked-structure prefetching techniques is able to improve `em3d` or `power`. The number of cache misses is very low in these programs so most of the prefetches hit in the L1 cache. Our prefetching results are similar to those reported in related work for linked structures in C programs [68, 70, 90].

Figure 5.28 presents a direct comparison of jump-pointer, greedy, and stride prefetching by combining the data in Figures 5.12, 5.17, and 5.26. Greedy prefetching, jump-pointer prefetching, and stride prefetching reduce the execution time by a geometric mean of 5%, 10%, and 9%, respectively. The largest reduction in execution time occurs in `treeadd` from stride prefetching.

In Figure 5.29 we normalize the bars to compare the busy and memory stall percentages directly. In contrast, Figure 5.28 normalizes the execution times and scales the busy and memory times appropriately. Figure 5.29 enables us to see the how much the prefetch methods affect the percentage of busy and memory time.

The effectiveness of greedy prefetching is limited. As the processor-memory gap increases, the usefulness of greedy prefetching will decrease. The problem is that the amount of latency that can be hidden is limited by the amount of computation that each object performs. Greedy prefetching performs best on binary trees where the prefetch instructions will hide the latency of accessing one child only partially, but may hide the latency of accessing the other child completely.

Jump-pointer prefetching has the best potential for improving the performance of programs with linked structures. The advantage of jump pointers is the potential to tolerate larger amounts of latency than greedy prefetching. The disadvantage of jump pointers is

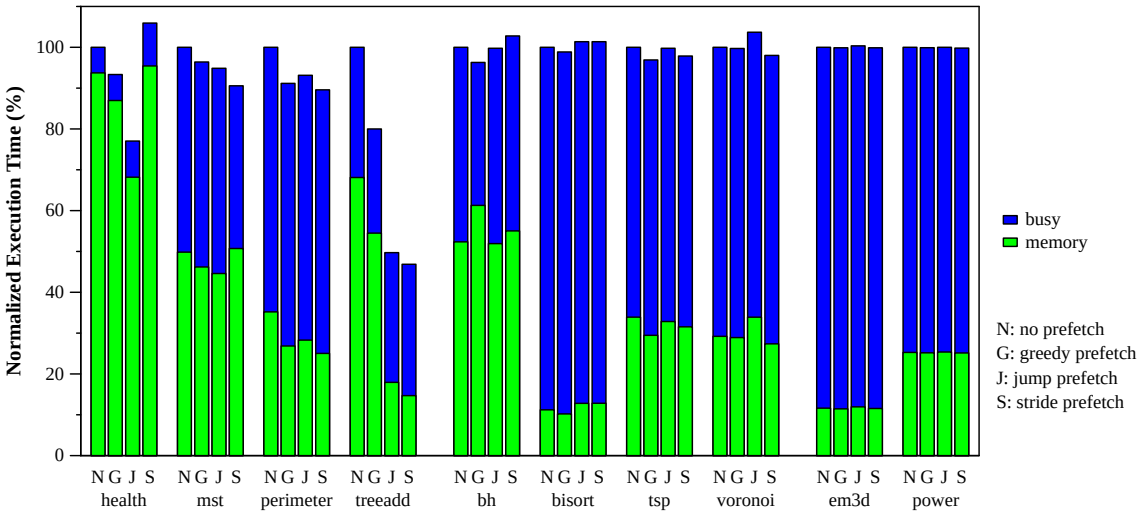


Figure 5.28. Comparing Execution Time in the Linked Structure Prefetching Methods

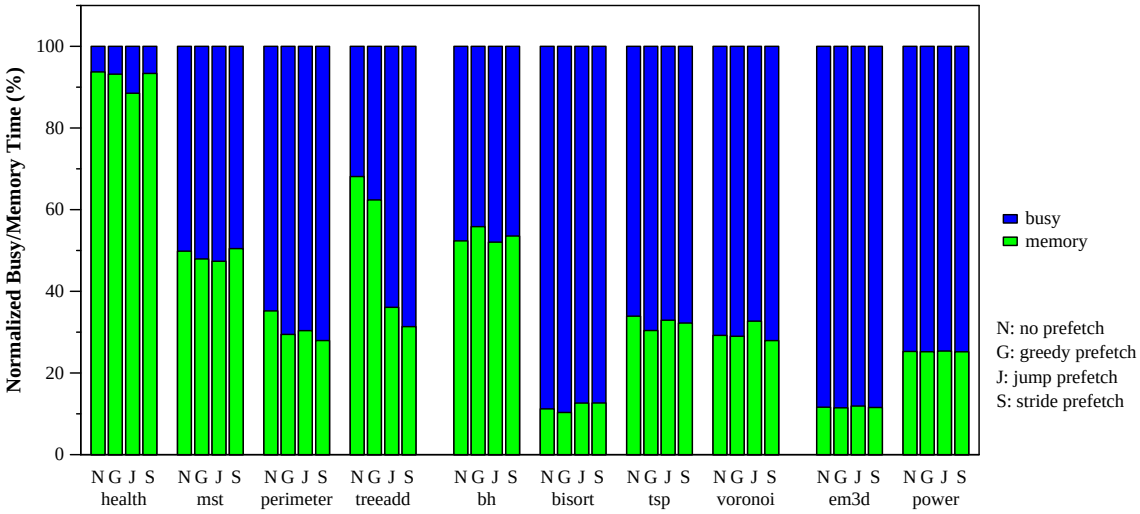


Figure 5.29. Comparing Busy/Memory Time in the Linked Structure Prefetching Methods

the run time cost of creating, updating, and using them, and the space cost for the additional field.

Stride prefetching eliminates the cost of the jump pointers, and is able to hide large amounts of latency. The effectiveness of the prefetch instructions relies upon the data layout. Chapter 6 discusses the interactions between stride prefetching and garbage collection. We show that the collector has the potential to improve the effectiveness of stride prefetching by laying out the data appropriately.

5.4 Architectural Sensitivity

The results in the previous sections of this chapter use a fixed architecture. We describe the architecture in Section 5.1, and list the simulation parameters in Table 5.1. In this section, we examine the performance of prefetching using different simulation parameters. We run experiments using different size caches, and we vary the memory hierarchy access times.

We run experiments using three other simulation configurations: *fast*, *large*, and *future*. Table 5.10 lists the parameters for each configuration. In the *fast* configuration, we decrease the L2 cache size and memory access times by dividing them in half. We change the memory parameters in the *future* configuration to model a realistic architecture that may appear several years in the future. The *future* parameters roughly correspond to projections made by Agarwal et al. [3]. The future projections suggests that cache sizes will remain small, and the cache access times will increase. RSIM allows only a single cache line size for both the L1 and L2 caches, but the future projections indicate that the L2 cache line size will be much larger. In the *large* configuration, we increase the L1 and L2 cache sizes.

For each cache configuration, we run experiments using array prefetching, greedy prefetching, and jump-pointer prefetching. We present overall execution times and normalize the execution times to those without prefetching. In each graph, the percentage at the top of each program's bar is the execution time of the new configuration normalized to that

Table 5.10. Different Simulation Configurations

Memory Parameter	Configuration			
	Base	Fast	Large	Future
L1 size	32K	32K	**64K**	32K
L2 size	256K	256K	**1024K**	**512K**
L1 time (cycles)	1	1	1	**2**
L2 time (cycles)	12	**6**	12	**16**
Mem. time (cycles)	60	**30**	60	**100**
L1 associativity	1	1	1	**2**
L2 associativity	4	4	4	4
Line size	32B	32B	32B	32B

of the *base* configuration. Both the *fast* and *large* configurations result in faster execution times than the *base* configuration. The execution time improvement is not surprising since we increase the cache sizes and decrease the access times in these configurations. The *future* configuration requires more cycles, which is not surprising because we increase the memory access time. However, we expect the clock speed in future architectures to be faster.

5.4.1 Array Prefetching

Figures 5.30, 5.31, and 5.32 show normalized execution times for array prefetching with the *fast*, *large*, and *future* configurations, respectively. Table 5.11 summarizes the average reduction in execution time for each configuration. Prefetching on the *future* architecture results in the largest reduction in execution time across all the benchmarks. The *fast* architecture results in the smallest improvement. The variation between the different configurations is high, and ranges from 26% to 15%.

In general, the trends in each configuration are similar. Prefetching improves the same programs, but the amount depends upon the memory parameters. In the *fast* and *large* configurations, the amount of time spent waiting for memory operations decreases. Thus there is less room for improvement. The *future* configuration has more room for improvement, which is the reason why prefetching performs the best on this configuration.

Table 5.11. Overall Results for Array Prefetching

	Base	Fast	Large	Future
Avg. Change	23 %	15 %	17 %	26 %

Table 5.12. Overall Results for Greedy Prefetching

Avg. Change	Base	Fast	Large	Future
All	5 %	3 %	4 %	5 %
Top Four	10 %	8 %	10 %	10 %

Table 5.13. Overall Results for Jump-Pointer Prefetching

Avg. Change	Base	Fast	Large	Future
All	10 %	9 %	8 %	9 %
Top Four	24 %	21 %	21 %	23 %

In each of the experiments, we use the same prefetch distance. Changing the prefetch distance for some individual benchmarks may result in slightly different results, but using a fixed value is robust for these configurations. Only a couple of programs may benefit from a larger prefetch distance in the *future* configuration, but using a smaller prefetch distance in the other configurations will most likely not change the results. In the *fast* and *large* configurations, we do not see a noticeable increase in the percentage of early prefetches.

5.4.2 Greedy Prefetching

Figures 5.33, 5.34, and 5.35 show normalized execution times for greedy prefetching with the *fast*, *large*, and *future* configurations, respectively. Table 5.12 summarizes the average reduction in execution time for each configuration. We show the average reduction for each program and for the four programs that prefetching improves, *i.e.*, `health`, `mst`, `perimeter`, and `treeadd`.

The smallest improvement occurs with the *fast* configuration, and the largest occurs with the *future* configuration. The variation in the performance improvements is not large. Across all programs, the smallest improvement is 3% and the largest is 5%. In the top four programs, the smallest improvement is 8% and the largest is 10%.

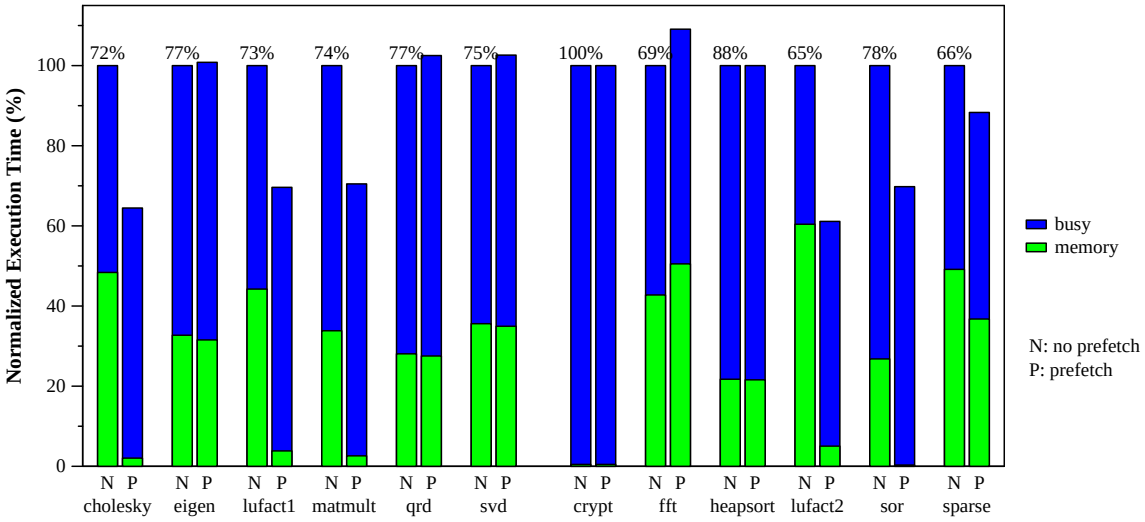


Figure 5.30. Array Prefetching Performance Using the *fast* Configuration

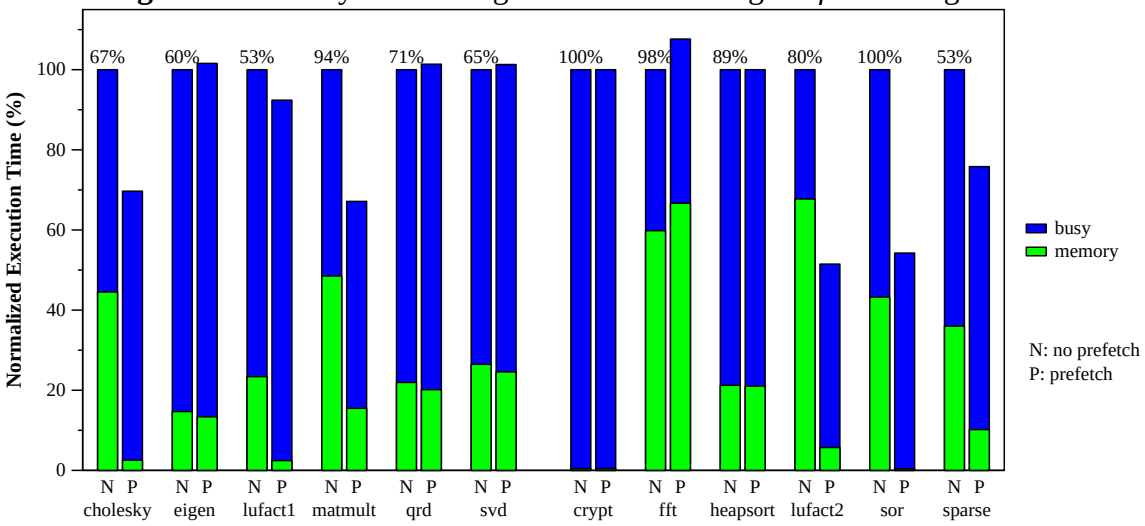


Figure 5.31. Array Prefetching Performance Using the *large* Configuration

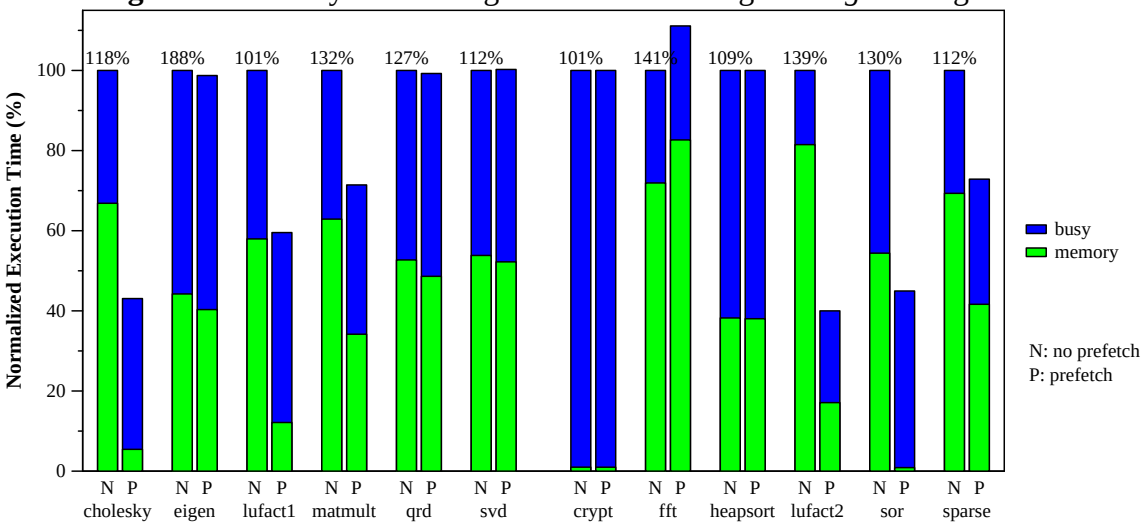


Figure 5.32. Array Prefetching Performance Using the *future* Configuration

Greedy prefetching results in larger improvements as the cache access time increases because there is more room for improvement. As the memory access time increases, greedy prefetching generates slightly more late prefetches.

5.4.3 Jump-Pointer Prefetching

Figures 5.36, 5.37, and 5.38 show normalized execution times for jump-pointer prefetching with the *fast*, *large*, and *future* configurations, respectively. Table 5.13 summaries the average reduction in execution time for each configuration. We show the average reduction for all the programs, and the four programs that prefetching improves, *i.e.*, `health`, `mst`, `perimeter`, and `treeadd`.

With jump-pointer prefetching, the smallest improvements occur when the caches are large. The largest improvements occurs on the *base* configuration. The average improvement on all the programs range from 8% to 10%. In the top four programs, the average improvement ranges from 21% to 24%. These results indicate that pointer prefetching is robust to changes in memory parameters for these programs.

5.4.4 Architectural Sensitivity Summary

In this section, we vary several of the memory hierarchy parameters and discuss the impact on prefetching performance. As we expect, prefetching has less effect on the architectures with faster memory access times or larger caches. The memory system is less of a bottleneck when the caches are more efficient. Prefetching results in better improvements with slower caches. As the gap between processor speed and memory speed continues to grow, our results suggest that prefetching will have a greater impact.

5.5 Chapter Summary

In this chapter, we evaluate the effectiveness of array and linked structure prefetching. We use two sets of array-based Java programs to evaluate array prefetching. These programs are from the Jama library package and the Java Grande benchmark suite. We eval-

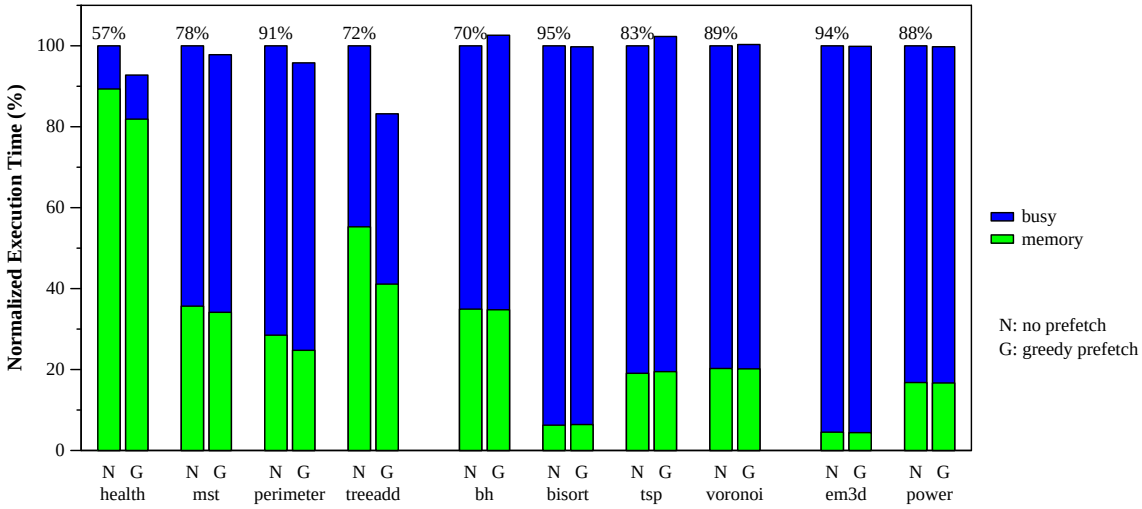


Figure 5.33. Greedy Prefetching Performance Using the *fast* Configuration

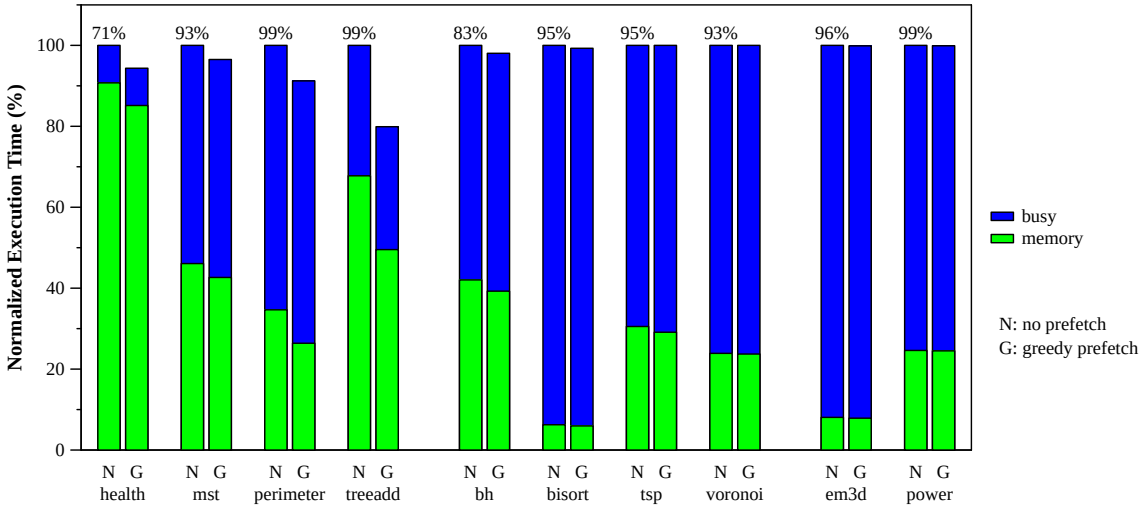


Figure 5.34. Greedy Prefetching Performance Using the *large* Configuration

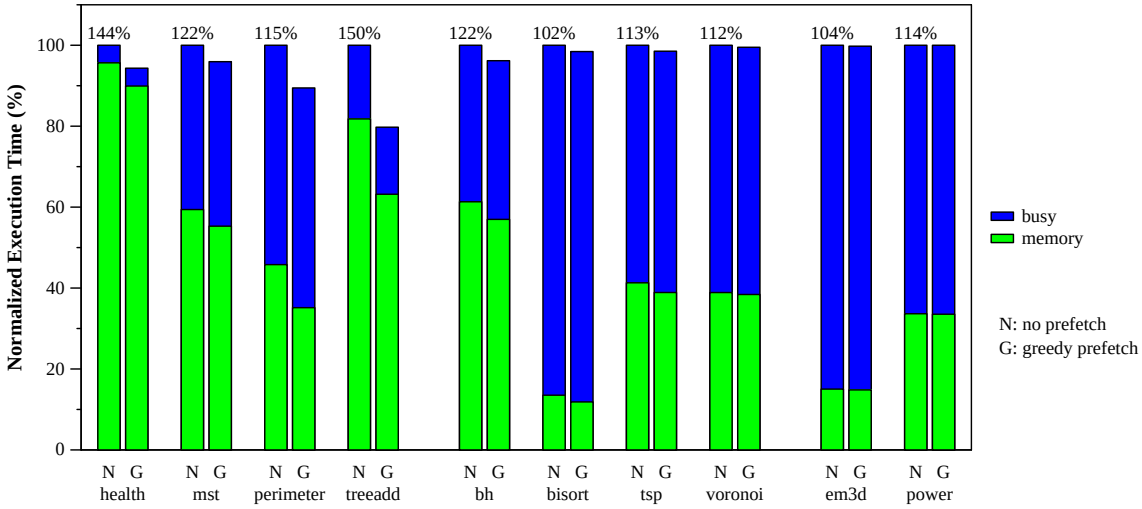


Figure 5.35. Greedy Prefetching Performance Using the *future* Configuration

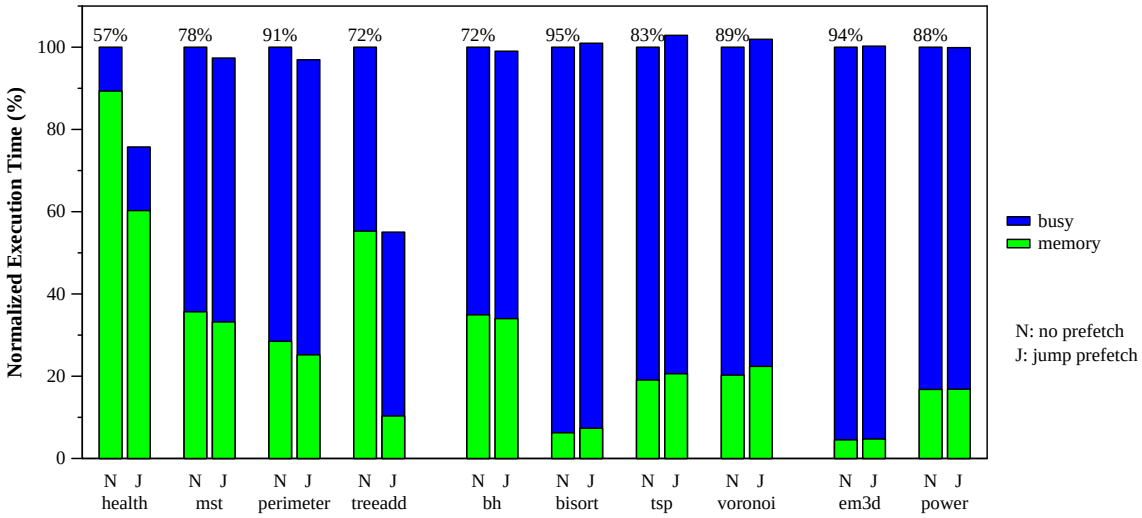


Figure 5.36. Jump-Pointer Prefetching Performance Using the *fast* Configuration

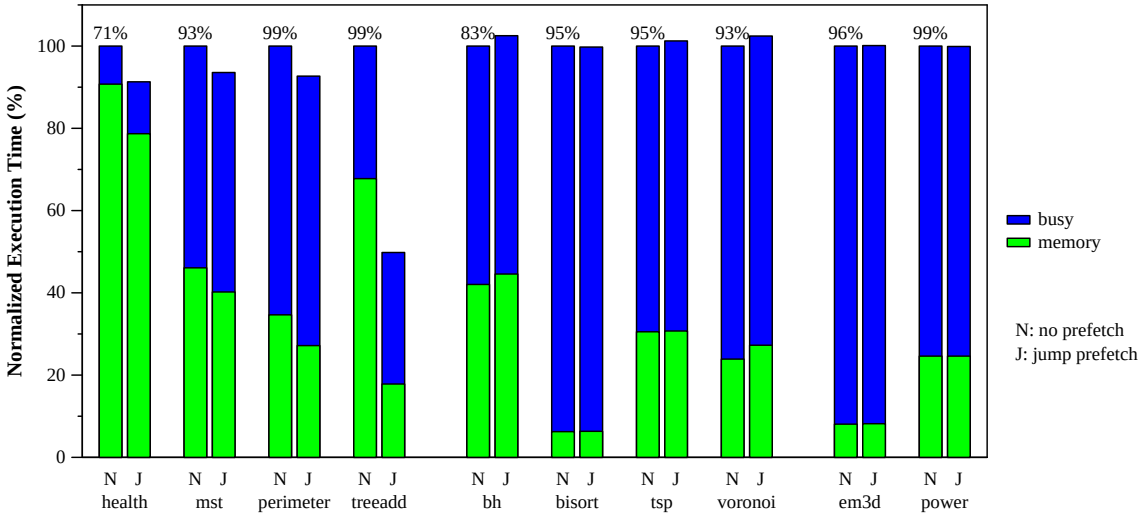


Figure 5.37. Jump-Pointer Prefetching Performance Using the *large* Configuration

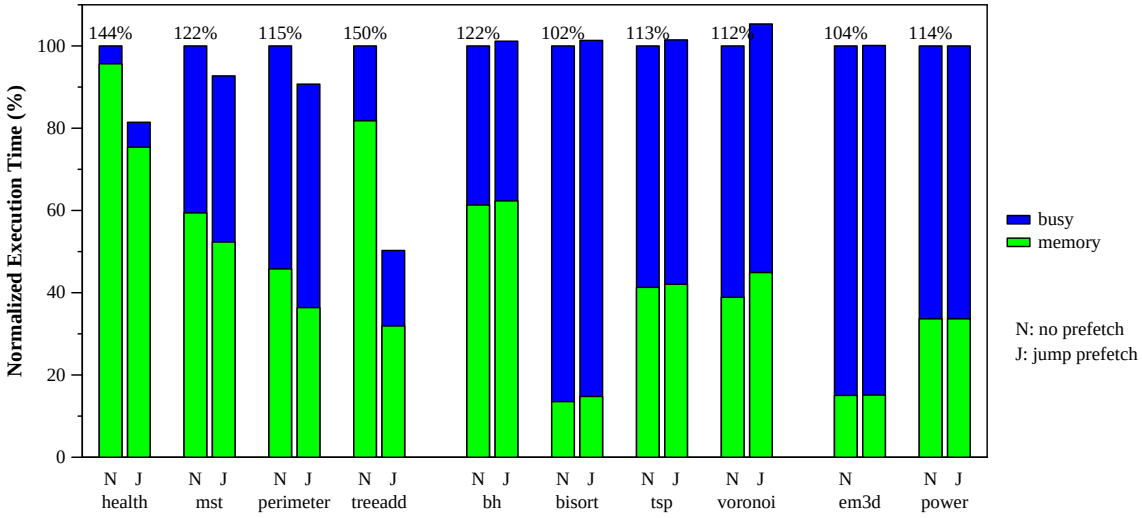


Figure 5.38. Jump-Pointer Prefetching Performance Using the *future* Configuration

uate three prefetch algorithms for linked-structure prefetching: greedy prefetching, jump-pointer prefetching, and stride prefetching. We use a Java version of the Olden benchmark suite to evaluate the linked-structure prefetch methods. We use RSIM, a simulator for an out-of-order superscalar processor, to obtain results with and without prefetching.

Array prefetching improves performance in six of the twelve programs by a geometric mean of 23%. The largest improvement is 58%, which occurs in LU factorization. Our results show that loop transformations and array analysis are not necessary to achieve large performance gains with prefetching in Java programs. Greedy prefetching often improves the performance of our programs even in the presence of object-oriented features, such as encapsulation, that hide accesses to underlying data structures. As memory latency increases, we show that greedy prefetching will become less effective in improving memory performance. Jump-pointer prefetching results in bigger improvements than greedy prefetching for some programs, but it is less consistent overall. Better compiler analysis is necessary to improve jump-pointer prefetching. Stride prefetching produces large improvements for some programs, but the results depend upon the data layout of the linked structures. The largest improvement occurs from using stride prefetching for `treeadd`. In the future, combining stride prefetching with dynamic optimization may produce more reliable results. Even with prefetching, our results show that there is still considerable room for improving the locality of object-oriented programs.

Compile-time data prefetching is effective in improving the memory performance of Java programs that traverses arrays and linked structures. We show that complex analyses and transformations are not necessary improve the performance of array-based codes. With modern processors, additional effort is not likely to produce much larger improvements over our results. Prefetching linked structures is more difficult. Although we see improvements, there is still room for further gains.

CHAPTER 6

GARBAGE COLLECTION AND PREFETCHING

Garbage collection automatically reclaims heap allocated memory that a program no longer references. Many modern object-oriented languages, including Java, require garbage collection for dynamic memory management. Garbage collection reduces the programmer's burden of managing memory and provides software engineering benefits.

In the previous chapter, we present our results without garbage collection because garbage collection can affect program performance significantly. Since our linked-structure prefetch algorithms do not alter the collector, disabling the collector enables us to understand the effects of prefetching better. In this chapter, we discuss the effect of a generational copying garbage collection on our prefetching schemes. Our results show that this garbage collector has little effect on the performance of prefetching for most programs. In some cases, it improves prefetching by improving locality.

We also show that memory performance *during garbage collection* is consistently poor. Across a range of programs, 50% of the execution time during garbage collection is spent waiting for memory. We propose using prefetching to improve the memory performance of a generational copying garbage collector. We add prefetch instructions to three places in the collector's algorithm by hand. Our approach is easy to implement in any copying collector. Our initial results show that the prefetch instructions are effective in reducing the execution time of garbage collection by as much as 32%, and by a geometric mean of 26% across all programs in one collector configuration. Although our prefetch instructions improve performance, the collector still spends a large percentage of time waiting for memory. The successful results from using just three prefetch instructions suggests that more aggressive

prefetching can further reduce the memory penalty. However, identifying more effective prefetch strategies is challenging.

We organize the chapter as follows. Section 6.1 describes the garbage collector in Vortex. In Section 6.2, we evaluate the impact of garbage collection on prefetching. We discuss specifically how the collector handles jump-pointers. We use several collector sizes in our evaluation. In Section 6.3, we experiment with adding prefetch instructions to the garbage collection algorithm itself. We show that our generational copying collector has poor memory performance, and that a few prefetch instructions can improve performance.

6.1 Garbage Collection in Vortex

The garbage collector in Vortex uses the UMass Language-Independent Garbage Collection Toolkit [49]. The collector uses a generational copying algorithm. A semi-space copying collector divides the heap into two equal size areas called From-space and To-space. The allocator obtains memory from To-space, and From-space contains old data. Garbage collection typically occurs when To-space is full. A copying garbage collector begins by flipping the meaning of To-space and From-space. The collector traverses the live objects in From-space, and copies them to To-space. After processing all the live objects, To-space contains a copy of all the live objects. The objects in From-space are dead. At this point, the program continues to allocate memory from To-space.

We illustrate the start of a collection in Figure 6.1. The heap contains six live objects. The collector begins by copying the root objects, `A` and `F`, to To-space. The collector processes the root objects, and copies all objects reachable from the roots. Figure 6.2 shows the heap after the collection. Notice that the live objects have been linearized in To-space.

The garbage collection toolkit uses Cheney's copying algorithm [23]. Cheney's algorithm copies the objects without using recursion by using two additional pointers, called `scan` and `free`. The pointers delineate the unprocessed objects, and `free` provides the

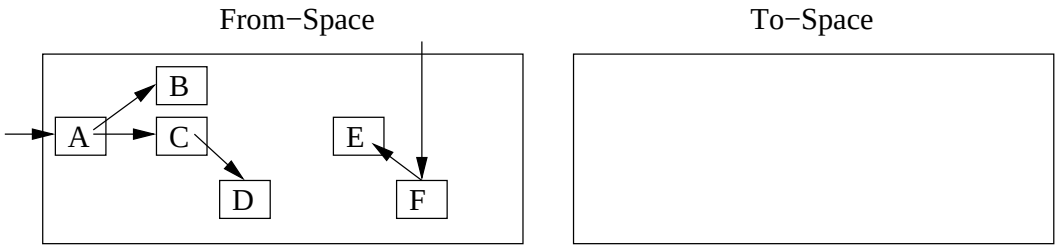


Figure 6.1. The Heap at the Start of a Collection

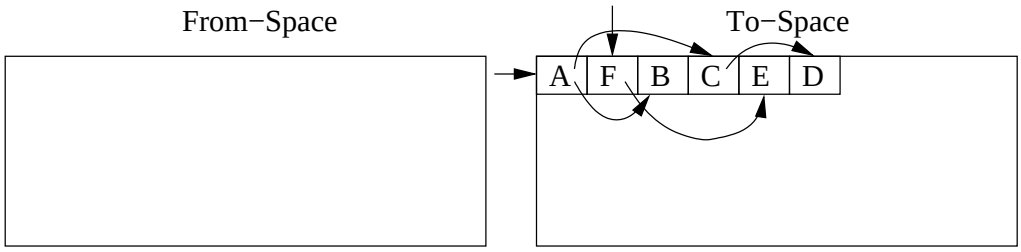


Figure 6.2. The Heap at the End of a Collection

```
cheney_scan() {
  scan = free = To-space;

  for each root R
    *R = copy(R);

  while (scan < free) {
    for each field P of scan
      *P = copy(P);
    scan = scan + size(scan);
  }
```

Figure 6.3. Cheney's Algorithm (from Jones and Lin [52])

current allocation point. Cheney's algorithm processes each object between `scan` and `free`, copying each object's reference fields, until the two pointers are equal. The algorithm performs a breadth first traversal of the live objects. Figure 6.3 shows the pseudo code for Cheney's algorithm.

Figure 6.4 shows a snapshot of To-space during a collection. The collector has processed the dark objects to the left of `scan`. The collector still needs to process the light objects between `scan` and `free`. The collector copies the newly encountered objects to the right of `free`.

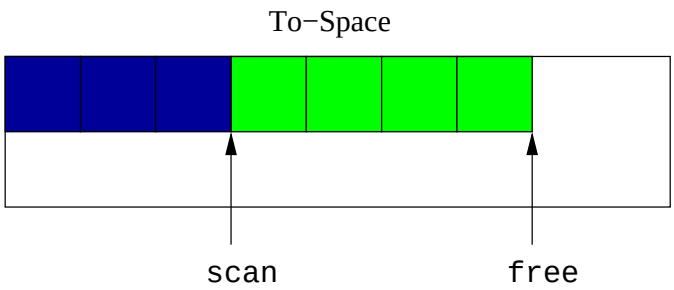


Figure 6.4. Snapshot of Cheney's Copying Algorithm

A generational collector divides the heap into two or more regions, called *generations*, that contain increasingly older objects. The youngest generation is the *nursery* and contains the most recently allocated objects. As objects survive collections, the collector copies them to older generations. The collector processes the older generations less frequently than the younger generations. The basic principle behind generational collectors is the weak generational hypothesis, which states that most objects die young [108].

The garbage collection toolkit enables the user to configure the heap at program start-up. A configuration file specifies the number of generations, the size of each generation, and when to promote objects between generations. The configuration file specifies a fixed size for each generation. A collector with fixed sized generations does not allow the nursery to grow during execution time and performs a collection when the amount of new allocation reaches the fixed limit.

6.2 Effect of GC on Prefetching Linked Structures

In this section, we evaluate the impact of garbage collection on the performance of our prefetch algorithms. Garbage collection can increase a program's execution time due to the additional cost of determining the live objects and copying them. In some cases, a copying collector may reduce execution time by improving the locality of the program. The cost of generational garbage collection depends upon several factors, including the size of each generation, the frequency of collections, and the number of generations.

6.2.1 Handling Jump-Pointers in the Collector

In this section, we discuss the details of how the garbage collector handles the jump-pointer prefetching field. As we describe in Section 4.3, jump-pointer prefetching adds a field to recurrent objects. The additional field contains a reference to another object, which we use to perform prefetching.

The garbage collector needs to be aware of the jump-pointer field and must handle the field specially. The garbage collector must not identify the referent of the jump-pointer field as a live object. If the jump-pointer is the only reference to an object, then the collector should not copy the object, which allows the memory allocator to reclaim the object. If the collector treats the jump-pointer as a regular field reference, then the collector retains excess objects that may be dead.

We extend Cheney's algorithm to handle the jump-pointers. The collector must *update* the jump-pointers to refer to the new objects in To-space, and the collector has an opportunity to *improve* the effectiveness of the jump-pointers. Updating the jump-pointers is a correctness issue. Since the collector must do work to update the jump-pointers, we re-initialize all the jump-pointers in order to improve the effectiveness as well.

While copying objects, the collector updates each field reference to refer to the new location of the copied object. Since the collector does not trace the jump-pointers, we need to extend the copying algorithm to update the jump-pointers. The difficulty is that the referent for the jump-pointer may not have been copied yet. We need a method to keep track of the pending jump-pointers that the collector can process once it copies the referent object.

The collector has an opportunity to improve the effectiveness of the jump-pointers. If a program updates a linked structure frequently during execution, then the jump-pointers may become ineffective over time. Programs that initialize the jump-pointers during object creation are prone to this problem. Rather than paying the cost of updating the jump-

pointers during each linked structure traversal, we use the collector to re-initialize the jump-pointers as it copies the objects to To-space.

Section 4.3.1 describes our technique for creating jump-pointers during the traversal of a linked structure. Since Cheney's algorithm is a breadth first traversal of all the live objects, we incorporate the ideas from Section 4.3.1 into Cheney's algorithm. To initialize jump-pointers, the extended algorithm uses a circular queue that contains the last *n* objects copied. The extended algorithm uses a separate circular queue for each class that contains a jump-pointer and only inserts objects of the appropriate type into the queue.

Figure 6.5 shows the code for the extended algorithm. The code corresponds to the sequence in Figure 4.17. The code uses three functions to access the jump-pointer information for an object. The method `jpp_queue()` returns the circular queue for an object. In our implementation, there is one queue for each type that contains a jump-pointer prefetch field. The method `jpp_index()` returns the current index into the queue, and `jpp_next_index()` advances the index to the next entry in the circular queue.

The extended algorithm updates the jump-pointers correctly and potentially improves the effectiveness of the jump-pointers. We no longer have the problem of updating jump-pointers that contain references to objects in From-space. Since the extended algorithm inserts the objects into a queue and creates jump-pointer links between objects in the queue, we ensure that the objects are in To-space.

6.2.2 Experimental Results

We evaluate the effect of garbage collection on the performance of greedy, jump-pointer, and stride prefetching. In our experiments, we vary the size of the generations, but we fix the number of generations at two. Objects that survive a collection are promoted from the nursery to the older generation. We experiment with different heap sizes.

In our experiments, the size of the generations depends upon the amount of memory that each program allocates. Table 5.6 lists the total amount of memory allocated, and

```
cheney_scan_jump_pointer() {
   scan = free = To-Space;
   for each root R
     *R = copy(R);

   while (scan < free) {
     for each jump pointer field J of scan {
       jqueue = jpp_queue(J);
       jindex = jpp_index(J);
       jObj = jqueue[jindex];
       jObj.prefetch = scan;
       jqueue[jindex] = scan;
       jpp_next_index(J);
     }
     for each field P of scan
       *P = copy(P);
     scan = scan + size(scan);
   }
}
```

Figure 6.5. Extended to Cheney's Algorithm to Handle Jump-Pointers

the maximum amount of memory live at any given point for each program. We use three different collector configurations in our experiments. In *GC1*, the size of the heap is the maximum live size for each program. Thus, since we specify two generations, each generation is half the maximum live size. The *GC1* heap size is very small. In *GC2*, the size of the heap is twice the maximum live size. In *GC3*, the size of the heap is three times the maximum live size.

Figure 6.6 shows the results of the Olden programs with garbage collection, but without prefetching. These results illustrate the cost of garbage collection in each program. For each program, Figure 6.6 shows the results with no garbage collection (N), the *GC1* collector (1), the *GC2* collector (2), and the *GC3* collector (3), from left to right. Note that the order of the programs on the x-axis in Figure 6.6 is different than in the figures from Chapter 5. We divide the programs into two groups. The programs in the second group, on the right side of Figure 6.6, have a maximum live size that is equal or almost equal to

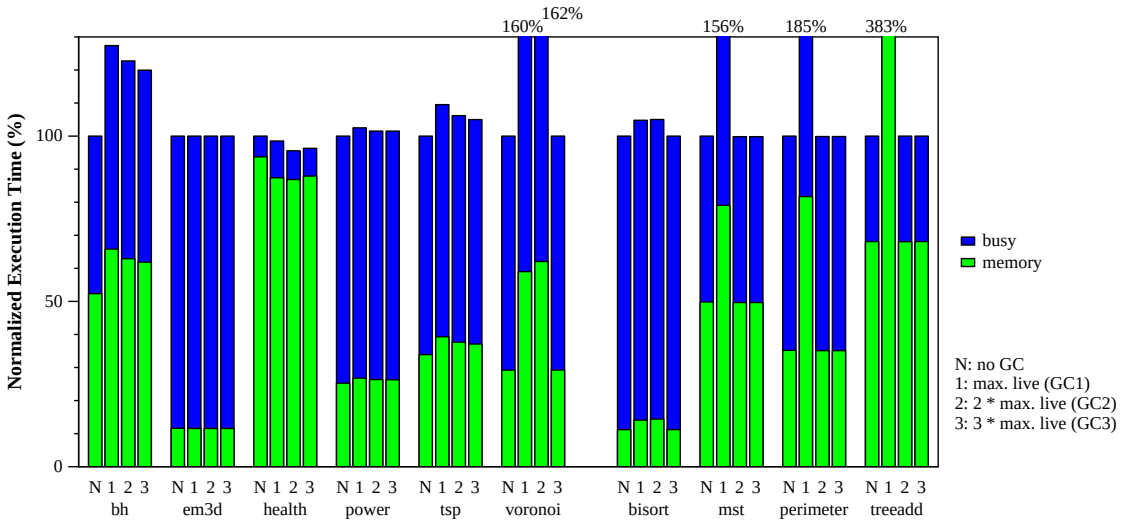


Figure 6.6. Performance with Garbage Collection

the total memory allocated. These four programs do not have interesting garbage collection characteristics, and should not be used to draw meaningful conclusions. We normalize all times to those without garbage collection. Several of the programs have large running times with garbage collection. The figure displays values up to 130% only, but we list the normalized execution time above the bars that have very long running times.

Garbage collection has an interesting effect on `health`. The execution time of `health` actually decreases when using garbage collection. The reason for the decrease in execution time is that the program achieves better locality when the collector reorganizes the data. Figure 6.2 shows that a collector tends to linearize data and co-locate objects that have connections. Co-locating objects increases spatial locality, and improves the performance of `health`.

Garbage collection has a severe negative impact on the performance of `mst`, `perimeter`, `treeadd`, and `voronoi` for the small collectors. These programs contain large heap allocated data structures that are persistent during the program's execution. The performance of these programs suffers because the collection cost in a copying collector is proportional to the amount of live data. For example, `treeadd` creates a 24MB binary tree that never changes. The program frequently performs garbage collection, and all the

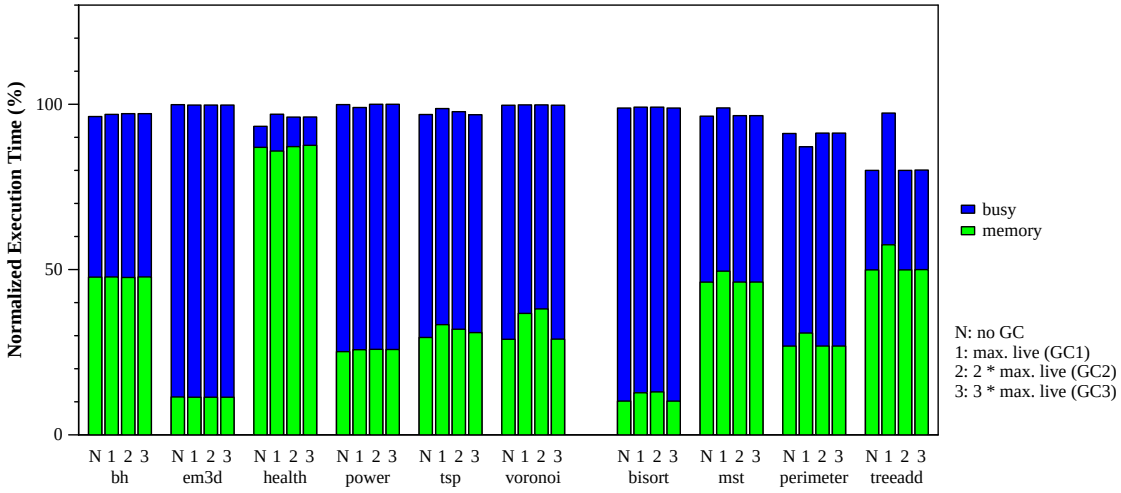


Figure 6.7. Greedy Prefetch Performance with Garbage Collection

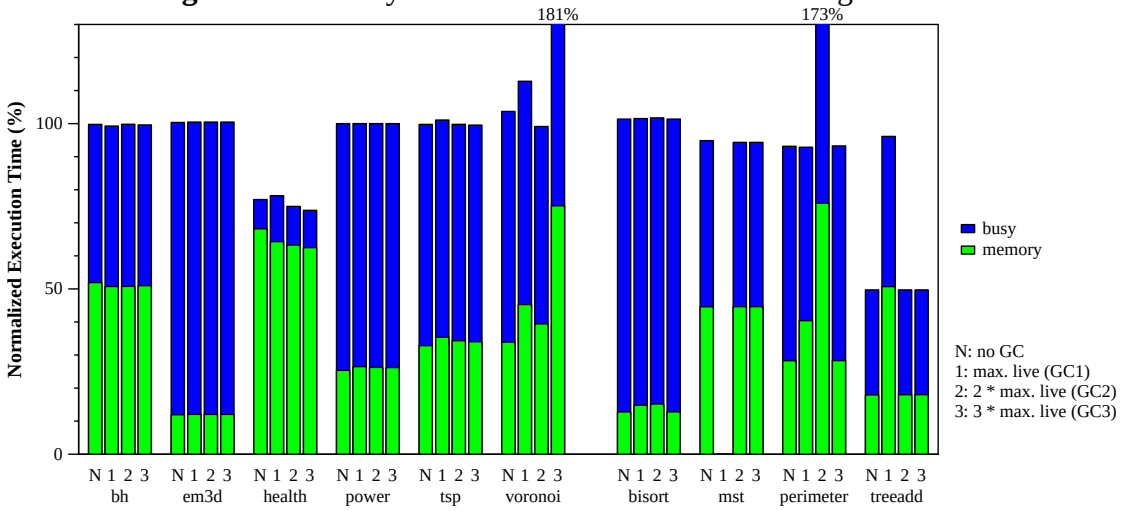


Figure 6.8. Jump-Pointer Prefetch Performance with Garbage Collection

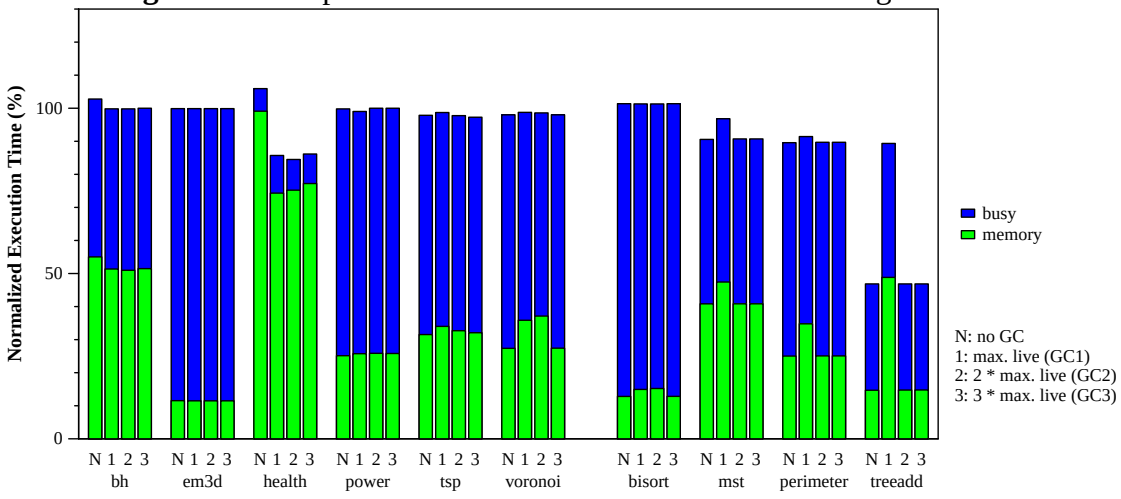


Figure 6.9. Stride Prefetch Performance with Garbage Collection

objects are live during each collection. The cost of collecting the large structure is very high, which is why the execution time of `treeadd` with garbage collection increases by 383%.

The performance of *GC3*, the large configuration, is similar to the performance without garbage collection, but *GC3* triggers collections in only four of the ten programs. These programs are `health`, `bh`, `tsp`, and `power`. The execution time with *GC3* ranges from 4% faster to 20% slower in `health` and `bh`, respectively.

Figure 6.7 shows the performance of greedy prefetching with garbage collection. Since we are interested in the effect of garbage collection, we normalize the execution times to the corresponding values in Figure 6.6. Garbage collection has a small effect on the performance of greedy prefetching in most programs. Using a garbage collector has a small positive effect on greedy prefetching in `perimeter` and `tsp`. `Treeadd` is the only program that produces significant variation among the different collectors. Greedy prefetching improves `treeadd` by only 3% with *GC1*, but by 20% with *GC2*, *GC3*, and no garbage collection. The difference is due to the large amount of time spent in garbage collection with *GC1* and *GC2*. With these collectors, `treeadd` spends just 17% of the execution time traversing the binary tree, and 77% of the execution time in the garbage collector.

Figure 6.8 shows the performance of jump-pointer prefetching with garbage collection. We normalize the execution times to the corresponding values in Figure 6.6.[1] There is a tight relationship between prefetching and garbage collection with jump-pointer prefetching. Section 4.3.3 summarizes how the garbage collector deals with jump-pointers. Due to the interaction, there is a potential for cooperation between the collector and the prefetching algorithm.

[1]We do not show a result for `mst` with *GC1* due to an undetermined bug.

The performance of `health` improves at different levels of garbage collection. The execution time in `health` decreases by 8% with *GC3* when compared to the execution time without garbage collection. The performance of `voronoi` shows significant variation with different collectors. With *GC2*, jump-pointer prefetching slightly reduces the execution time, but the execution time increases with *GC1* and *GC2* in `voronoi`.

Figure 6.8 illustrates an interesting effect due to the interaction between jump-pointer prefetching and garbage collection. Jump-pointer prefetching adds an extra field to objects that contain jump pointers. The extra field increases the amount of memory the program allocates. In most cases, the extra memory is not an important issue. In `perimeter` and `voronoi`, the extra memory results in additional garbage collection. With *GC2*, the additional collection causes `perimeter` to run 173% more slowly with prefetching. Without the jump-pointer field, `perimeter` does not invoke the collector with *GC2*. The same effect occurs in `voronoi` with *GC3*.

Figure 6.9 shows the results for stride prefetching when we enable garbage collection. The most interesting result occurs in `health`. Without garbage collection, stride prefetching increases the execution time by 6% because the linked structures in `health` become disjoint as the program inserts and deletes objects. As Figure 6.2 shows, the garbage collector linearizes the linked structures making them amenable to stride prefetching. With *GC2*, stride prefetching reduces the execution time by 15%.

6.3 Prefetching in the Garbage Collector

In this section, we experiment with adding prefetch instructions to the garbage collector itself. We show that prefetching is able to reduce the cost of garbage collection by improving memory performance.

The memory performance during garbage collection is poor. Figure 6.10 shows that memory performance is a problem during the garbage collection portion of a program's execution time. In Figure 6.10, the nursery size is half the maximum live size for each pro-

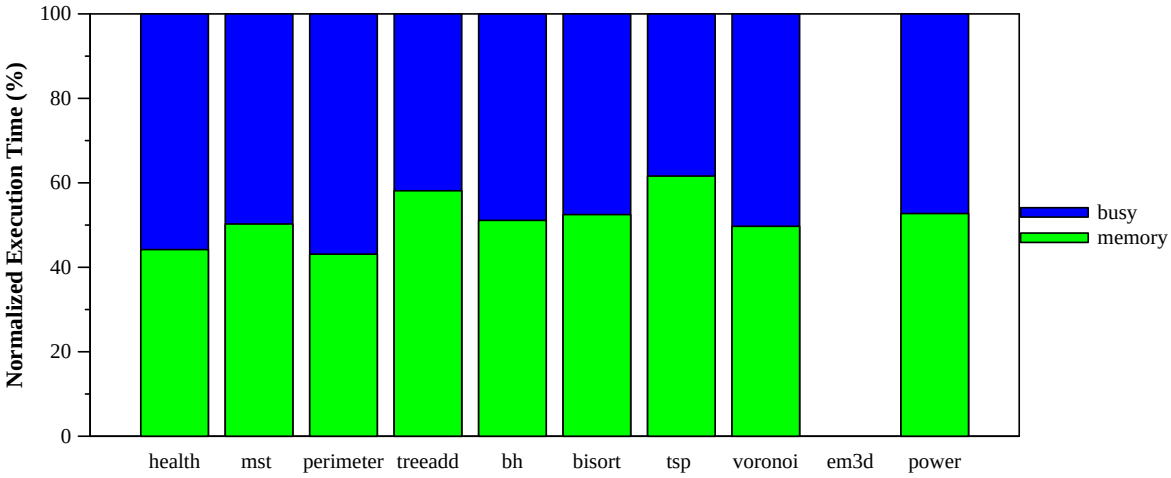


Figure 6.10. Memory Penalty During Garbage Collection Using a Small Heap

```
cheney_scan_prefetch() {
  scan = free = To-Space;
  for each root R
    *R = copy_prefetch(R);

  while (scan < free) {
    prefetch (scan+n);
    for each field P of scan
      *P = copy_prefetch(P);
    scan = scan + size(scan);
  }
}
```

Figure 6.11. Prefetching in To-Space

```
copy_prefetch(P) {
  if forwarded(P)
    return forward_addr(P);

  prefetch (P+n);
  addr = free;
  memcpy(free, P, size(P));
  free = free + size(P);
  forward_addr(P) = addr;
  return addr;
}
```

Figure 6.12. Prefetching in From-Space

gram, but we see similar values for larger heaps. The size of the heap affects the number of collections that occur and does not affect the memory penalty. Table 5.6 lists the maximum live size for each of the Olden programs. Figure 6.10 shows that garbage collector often spends at least 50% of the time waiting for memory. The amount is fairly steady across different programs, which use different amounts of memory, and invoke the garbage collector at different rates. We do not have results for `em3d` because it does not perform garbage collection.

We experiment with adding prefetch instructions during three parts of the garbage collection algorithm.

To-Space Figure 6.2 shows that the live objects become linearized as the collector copies them to To-space. Figure 6.4 shows a snapshot of To-space during the collection process. We add a prefetch instruction to the Cheney scan algorithm, as shown in Figure 6.11. Prior to processing the object to the right of `scan`, we prefetch *n* bytes ahead. Thus, we ensure that future objects will be in the cache when they are scanned.

From-Space Figure 6.1 depicts the heap just prior to the collection. Although the objects in this picture are spread throughout memory, live objects often cluster into consecutive locations in practice. We thus prefetch objects in From-space during the copying algorithm also. Whenever the algorithm copies an object, the collector prefetches *n* bytes ahead. Figure 6.12 shows the copying code with the prefetch instruction.

Object Scan The prior two methods do not explicitly take advantage of a linked object's pointer structure. While scanning the objects in To-space, we add prefetch instructions that prefetch the fields of each object. The prefetch is effective when the fields reference objects that the collector has not copied yet. Figure 6.13 shows the change from adding a prefetch instruction to Cheney's algorithm. Before the algorithm copies the fields of an object, we add a loop to prefetch the fields of the object. In the implementation, we unroll the prefetch loop to handle the common cases when an object contains one, two, or three fields.

In our experiments, the prefetch distance is 32 bytes for the *to-space* and *from-space* prefetch instructions. Figures 6.14, 6.15, and 6.16 show the effect of prefetching during garbage collection when the size of the initial heap is one, two, and three times the maximum live size, respectively. The y-axis is the execution time *during garbage collection* only. We normalize all the garbage collection times to those without prefetching (N). The value above each bar without prefetching (N) is the percentage of total execution time that the programs spend in the collector. For example, in Figure 6.14 `health` spends 8% of its execution time in the collector. The figures present separate results for prefetching in

```
cheney_scan_prefetch() {
  scan = free = To-Space;
  for each root R
    *R = copy(R);

  while (scan < free) {
    for each field P of scan
      prefetch (P);
    for each field P of scan
      *P = copy(P);
    scan = scan + size(scan);
  }
}
```

Figure 6.13. Prefetching Fields During the Object Scan

To-space (T), From-space (F), in the object scan (R), and with all the prefetch instructions (A).

In Figure 6.14, the amount of time spent in the collector ranges from 2% to 77% of total execution time for the programs that perform garbage collection. `Treeadd` is an outlier since the next closest value is 38% in `perimeter`. In `treeadd`, a negligible number of objects become garbage, so the collector copies the entire binary tree repeatedly. Prefetching in To-space, From-space, and in the object scan decreases the collector's execution time by a geometric mean of 10%, 11%, and 12%, respectively. The largest improvement occurs in `power` by prefetching the objects in To-space. With all three prefetches, the execution time of the collector decreases by a geometric mean of 26%.

In Figure 6.15, the amount of time spent in the collector ranges from 1% to 38% of total execution time. When we increase the size of the nursery, neither `mst` nor `perimeter` require garbage collection. Prefetching in To-space, From-space, and in the object scan decreases the execution time by a geometric mean 10%, 10%, and 11%, respectively. With all the prefetches, the collector execution time decreases by a geometric mean of 25%.

In Figure 6.16, the amount of time spent in the collector ranges from 1% to 18% of total execution time. Only four of the ten programs allocate enough objects to invoke

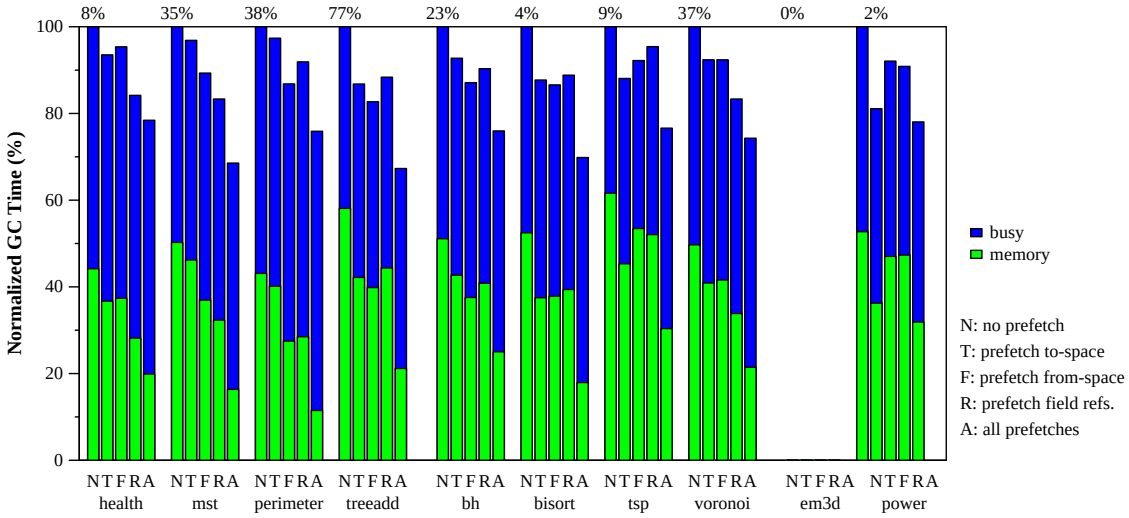


Figure 6.14. Prefetch During Garbage Collection (1*max. live)

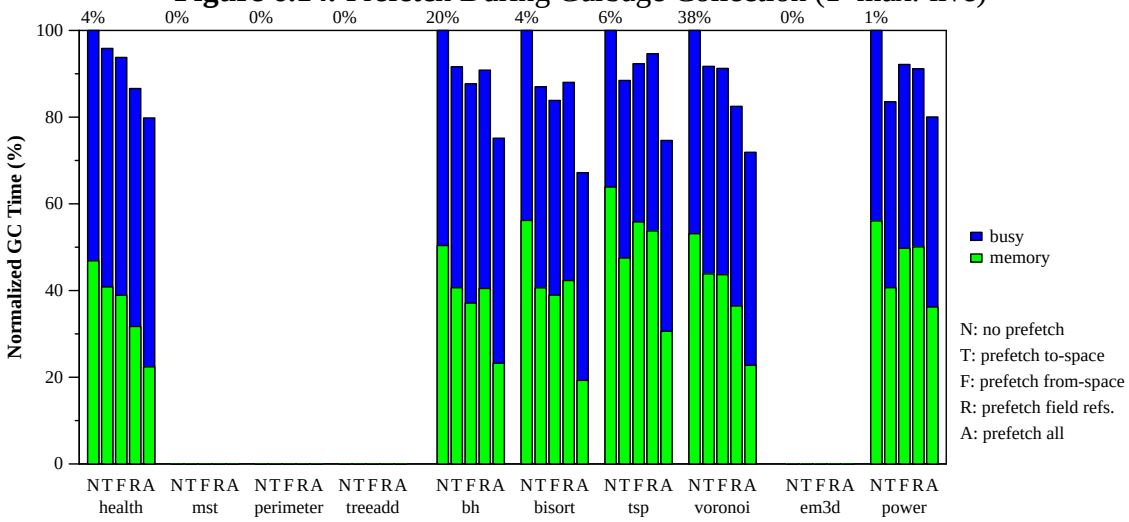


Figure 6.15. Prefetch During Garbage Collection (2*max. live)

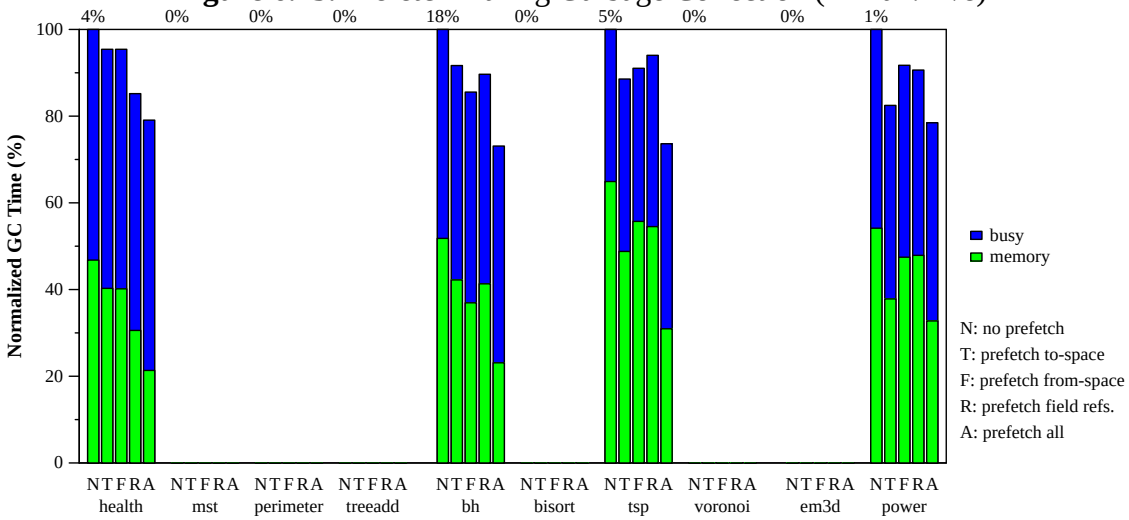


Figure 6.16. Prefetch During Garbage Collection (3*max. live)

the collector. Prefetching in To-space, From-space, and in the object scan improves the collector performance by a geometric mean of 11%, 9%, and 10%, respectively. With all three prefetches, the execution time decreases by a geometric mean of 24%.

The amount that the prefetch instructions improve the garbage collector appears to be independent of the nursery size. Each prefetch instruction contributes to the execution time improvements. There is only small variation between the programs, so the improvements are fairly stable. The largest reduction in execution time is 32% in `bisort` when using all three prefetch instructions. These programs still suffer from poor memory performance even with prefetching, but we show that simple prefetching techniques in the collector are able to improve performance.

6.4 Chapter Summary

In this chapter, we discuss the effect of garbage collection on prefetching. We enable garbage collection at run time and discuss the results for the linked-structure prefetching schemes. The cost of garbage collection is high for several of our benchmarks. We vary the heap size to obtain more general results. Garbage collection does not have a significant impact on prefetching in many of the programs, but there are a few exceptions. For example, jump-pointers consume more memory, which results in additional garbage collections in a couple of the programs. In contrast, we notice that the garbage collector is able to reorganize data such that the performance of prefetching improves. More research is necessary to quantify and exploit this effect further.

We also show that the memory performance of the garbage collector is poor. We improve the memory performance by adding three prefetch instructions. One instruction prefetches memory in To-space, one prefetches memory in From-space, and the third prefetches field references during the scan phase. The prefetch instructions reduce the garbage collection time by as much as 32%. Even with prefetch instructions, there is still room

for improvement. We believe it is worthwhile to pursue more aggressive techniques for reducing the memory penalty during garbage collection.

CHAPTER 7

CONCLUSIONS

The memory hierarchy in modern architectures continues to be a major performance bottleneck. Many existing techniques for improving memory performance focus on Fortran and C programs. Memory latency is also a barrier to achieving high performance in object-oriented languages. Existing software techniques are inadequate for exposing optimization opportunities in object-oriented programs that traverse linked structures and arrays. In this dissertation, we develop and evaluate new compiler algorithms for software prefetching. We show that software prefetching can improve the memory performance of Java programs that use arrays and linked structures.

In this chapter, we discuss directions for future work, and we summarize our contributions. The techniques that we develop in this dissertation lead directly to a number of potential directions for further investigation. We conclude by summarizing our contributions.

7.1 Future Work

We improve the memory performance of linked structures, but there is still much room for improvement. Better techniques are needed to reduce the memory penalty further, especially in programs that contain irregular traversals of linked structures. We suggest combining prefetching with data layout optimizations or code transformations to improve locality. We also believe that more advanced jump-pointer prefetching techniques can improve the potential of prefetching. Some possible extensions include adding multiple jump-pointers

to each object, inserting jump-pointers between objects of different types, and more selective methods for updating jump-pointers to minimize the cost.

We are interested in evaluating performance of our prefetching techniques on a real processor that contains a useful prefetch instruction. Our compiler currently generates SPARC assembly language, but we are unlikely to see improvements from prefetching on an UltraSPARC II because it is an in-order processor, allows a very limited number of outstanding loads, and prefetches into the L2 cache only. We would like to evaluate our prefetching technique on an UltraSPARC III, which implements hardware prefetching and contains a separate cache for prefetched data. We believe that prefetching will produce more significant results on architectures that contain better support for prefetching, such as the POWER4 or Pentium 4 architectures. To support effective software prefetching, an architecture needs a non-binding instruction that prefetches a cache line into the L1 cache, multiple ports to the L1 cache, the ability to support a large number of outstanding loads, the ability to prefetch integer and floating point data, and allow out-of-order execution.

We also are interested in evaluating our analysis and prefetching algorithms in a just-in-time (JIT) Java compiler. We believe our intraprocedural recurrence analysis is efficient enough to run in a JIT environment. Several JIT compilers, such as the Jikes RVM and Sun's HotSpot, contain data-flow analysis frameworks already. We would like to investigate techniques to make our interprocedural analysis more efficient so that it may used in a JIT compiler. Since the current interprocedural analysis is context-sensitive, it is too expensive to run in a JIT compiler. Discovering efficient interprocedural analysis techniques in a JIT compiler is a stimulating research focus.

This dissertation focuses on static analysis to discover prefetching opportunities. A potential research direction is to use run-time information to identify objects to prefetch. The Java environment encourages dynamic and adaptive compilation strategies. The challenge is to find techniques that compute useful information about memory references cheaply at run time.

Our results are encouraging, but we would like to evaluate prefetching on a larger set of benchmarks. Evaluating optimizations for Java is difficult due to a lack of interesting benchmark programs. Other than the programs we use, the SPECjvm98 benchmarks suite is the only set of standard Java programs that researchers use to evaluate performance. We have performed initial experiments on several of the SPECjvm98 programs, but we have encountered limited success. As a whole, the SPECjvm98 programs do not spend a significant amount of time traversing linked structures or arrays in a regular manner. The community needs a larger set of interesting programs for performance evaluation.

We use our recurrence analysis for prefetching only. We are interested in exploring the potential of using the recurrence analysis to solve other problems. We believe that further improvements are possible by using the recurrence analysis for data layout optimizations and code transformations. One potential idea is to extend the garbage collector to use the recurrence information for improving locality.

Our initial results from investigating the synergy between the garbage collector and prefetching suggest that further research will be useful. The ability of the garbage collector to assist prefetching is very interesting. Our preliminary results show that a copying collector can help organize the linked structures to improve the effectiveness of prefetching. Our results also show that prefetching can improve the memory performance of the collector itself. There is still room for improvement though.

7.2 Contributions

We develop a new data-flow analysis for identifying recurrences in object-oriented programs. The types of recurrences include induction variables and linked structure traversals. Our unified treatment of arrays and linked structures is unique. The analysis contains an intraprocedural component for finding recurrences in loops, and an interprocedural component for finding recurrences across procedure calls. We extend the analysis to track recurrences that are stored in fields and arrays between loop iterations. We use our analysis

to implement and evaluate array prefetching and three linked-structure prefetching techniques: greedy prefetching, jump-pointer prefetching, and stride prefetching.

We implement a new array prefetch technique that does not require locality analysis or loop transformations. Our algorithm generates prefetches for all array references containing loop induction variables. We generate an additional prefetch for array elements that contain object references. We present results showing the effectiveness of prefetching on a set of array-based Java programs. Prefetching improves performance in six of the twelve programs by a geometric mean of 23%. The largest improvement is 58%, which occurs in LU factorization. Our simple technique is able to eliminate almost all the memory stalls in several programs. Our results show that loop transformations and array dependence analysis are not necessary to achieve large performance gains with prefetching on modern processors.

Linked structure prefetching often improves the performance of our programs even in the presence of object-oriented features, such as encapsulation, that hide accesses to underlying data structures. The linked structure prefetching techniques improve performance in several of the Olden benchmark programs. Greedy prefetching improves performance by a small, yet consistent, amount. Jump-pointer and stride prefetching produce larger improvements than greedy prefetching, but the techniques are less consistent. In one program, stride prefetching improves performance by 53%. One reason that prefetching is not more effective is that several of our programs do not spend very much time accessing linked structures. Prefetching is most effective on programs that exhibit poor locality during linked structure traversals.

As memory latency increases, greedy prefetching will become less effective in improving memory performance. Jump-pointer and stride prefetching have the potential for larger improvements, but also can increase execution time by prefetching useless data. Better compiler analysis with additional run-time support is necessary to improve jump-pointer prefetching. Stride prefetching requires dynamic information or greater assistance from

the garbage collector to be most effective. Although prefetching improves the performance of programs with linked structures, there is still room for improvement.

We also evaluate the effect of garbage collection on prefetching, and we improve our collector's memory performance by inserting prefetch instructions into the collector itself. The collector can potentially improve the performance of jump-pointer and stride prefetching, and we see this effect in one program. For most of the other programs, garbage collection does not change the overall trends significantly. We show that the collector itself has poor memory performance. We add prefetch instructions to the collector in three parts of the algorithm. The prefetch instructions improve the collector's memory performance and reduce the execution time of the collector by as much as 32%.

We develop a unified whole-program data-flow analysis for identifying recurrences in Java programs. We use our recurrence analysis to identify and exploit prefetch opportunities in array and linked structure traversals. We show that compile-time software data prefetching is effective in improving the memory performance of Java programs.

BIBLIOGRAPHY

[1] Walid Abu-Sufah. *Improving the Performance of Virtual Memory Computers*. PhD thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, 1979.

[2] Anant Agarwal, Beng Hong Lim, David Kranz, and John Kubiatowicz. APRIL: A processor architecture for multprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 104–114, Seattle, WA, June 1990.

[3] Vikas Agarwal, M.S. Hrishikesh, Stephan W. Keckler, and Doug Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 248–259, 2000.

[4] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[5] Jonathan Aldrich, Craig Chambers, Emin Gun Sirer, and Susan Eggers. Static analyses for eliminating unnecessary synchronization from Java programs. In *Proceedings of the Sixth International Static Analysis Symposium*, pages 19–38, Venice, Italy, September 1999.

[6] Frances Allen, Michael Burke, Phillippe Charles, Ron Cytron, and Jeanne Ferrante. An overview of the PTRAN analysis system for multiprocessing. *Journal of Parallel and Distributed Computing*, 5(5):617–640, October 1988.

[7] B. Alpern, C.R. Attanasio, J.J. Barton, M.G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S.J. Fink, D. Grove, M. Hind, S.F. Hummel, D. Lieber, V. Litvinov, M.F Mergen, T. Hgo, J.R. Russell, V. Sarkar, M.J. Serrano, J.C. Sheperd, S.E. Smith, V.C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM System Journal*, 39(1):211–238, February 2000.

[8] Zahira Ammarguellat and W.L. Harrison III. Automatic recognition of induction variables and recurrence relations by abstract interpretation. In *Proceedings of the 1990 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 283–295, White Plains, NY, June 1990.

[9] Pedro V. Artigas, Manish Gupta, Samuel P. Midkiff, and Jose E. Moreira. Automatic loop transformations and parallelization for Java. In *Proceedings of the 2000 International Conference on Supercomputing*, pages 1–10, Santa Fe, NM, May 2000.

[10] Jean-Loup Baer and Tien-Fu Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of 1991 Conference on Supercomputing*, pages 176–186, Alburquerque, NM, November 1991.

[11] David Bernstein, Doron Cohen, Ari Freund, and Dror E. Maydan. Compiler techniques for data prefetching on the PowerPC. In *Proceedings of the 1995 International Conference on Parallel Architectures and Compilation Techniques*, pages 19–26, Limassos, Cyprus, June 1995.

[12] Hans-J. Boehm. Reducing garbage collector cache misses. In *Proceedings of the 2000 International Symposium on Memory Management*, pages 59–64, Minneapolis, Minnesota, October 2000.

[13] Preston Briggs, Keith Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.

[14] J.M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A benchmarks suite for high performance Java. *Concurrency : Practice and Experience*, 12(6):375–388, May 2000.

[15] Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. Cache-conscious data placement. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 139–149, San Jose, CA, October 1998.

[16] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, Santa Clara, CA, April 1991.

[17] Martin C. Carlisle and Anne Rogers. Software caching and computation migration in Olden. In *Proceedings of the 1995 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 29–38, Santa Barbara, CA, July 1995.

[18] Craig Chambers, Jeffrey Dean, and David Grove. Frameworks for intra- and interprocedural dataflow analysis. Technical Report TR 96-11-02, Department of Computer Science, University of Washington, November 1996.

[19] Tien-Fu Chen. *Data Prefetching for High Performance Processors*. PhD thesis, University of Washington, Department of Computer Science and Engineering, July 1993.

[20] Tien-Fu Chen. An effective programmable prefetch engine for on-chip caches. In *Proceedings of the 28th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-28*, pages 237–242, Ann Arbor, MI, December 1995.

[21] Tien-Fu Chen and Jean-Loup Baer. Reducing memory latency via non-blocking and prefetching caches. In *Fifth International Conference on Architectural Support*

for Programming Languages and Operating Systems, pages 51–61, Boston, MA, October 1992.

[22] William Y. Chen, Scott A. Mahlke, Pohua P. Chang, and Wen-mei W. Hwu. Data access microarchitectures for superscalar processors with compiler-assisted data prefetching. In *Proceedings of the 24th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-24*, pages 69–73, November 1991.

[23] C.J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.

[24] Trishul M. Chilimbi, Bob Davidson, and James R. Larus. Cache-conscious structure definition. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–24, Atlanta, GA, May 1999.

[25] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, Atlanta, GA, May 1999.

[26] Trishul M. Chilimbi and Martin Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–209, Berlin, Germany, June 2002.

[27] Trishul M. Chilimbi and James R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *Proceedings of the 1998 International Symposium on Memory Management*, pages 37–48, Vancouver, BC, October 1998.

[28] Jamison D. Collins, Hong Wang, Dean M. Tullsen, Christopher Hughes, Yong-Fong Lee, Dan Lavery, and John P. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 14–25, Goteborg, Sweden, June 2001.

[29] Robert Cooksey, Dennis Colarelli, and Dirk Grunwald. Content-based prefetching: Initial results. In *Proceedings of the 2nd Workshop on Intelligent Memory Systems*, Boston, MA, November 2000.

[30] Robert Courts. Improving locality of reference in a garbage-collecting memory management system. *Communications of the ACM*, 31(9):1128–1138, September 1988.

[31] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[32] Fredrik Dahlgren, Michel Dubois, and Per Stenstrom. Fixed and adaptive sequential prefetching in shared-memory multiprocessors. In *Proceedings of the 1993 International Conference on Parallel Processing*, volume 1, pages 56–63, August 1993.

[33] Fredrik Dahlgren and Per Stenstrom. Effectiveness of hardware-based stride and sequential prefetching in shared-memory multiprocessors. In *Proceedings of the 1st International Symposium on High Performance Computer Architecture*, pages 68–77, January 1995.

[34] Jeffrey Dean, Greg DeFouw, David Grove, Vassily Litinov, and Craig Chambers. Vortex: An optimizing compiler for object-oriented languages. In *Proceedings of the 1996 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, pages 83–100, San Jose, CA, October 1996.

[35] Julian Dolby and Andrew A. Chien. An automatic object inlining optimization and its evaluation. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 345–357, Vancouver, Canada, June 2000.

[36] James Dundas and Trevor Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the 1997 International Conference on Supercomputing*, pages 68–75, Vienna, Austria, July 1997.

[37] Wei fen Lin, Steven K. Reinhardt, and Doug Burger. Designing a modern memory hierarchy with hardware prefetching. *IEEE Transactions on Computers*, 50(11):1–17, November 2001.

[38] Michael Franz and Thomas Kistler. Splitting data objects to increase cache utilization. Technical Report TR 98-34, Department of Information and Computer Science, University of California, Irvine, CA, October 1998.

[39] John W.C. Fu and Janak H. Patel. Data prefetching in multiprocessor vector cache memories. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 54–63, Toronto, Canada, May 1991.

[40] Michael P. Gerlek, Eric Stoltz, and Michael Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA form. *ACM Transactions on Programming Languages and Systems*, 17(1):85–122, 1995.

[41] Edward H. Gornish. *Adaptive and Integrated Data Cache Prefetching for Shared-Memory Multiprocessors*. PhD thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, October 1995.

[42] Edward H. Gornish, Elana D. Granston, and Alexander V. Veidenbaum. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 342–353, Amsterdam, The Netherlands, June 1990.

[43] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley, 1996.

[44] Mohammad R. Haghighat and Constantine D. Polychronopoulos. Symbolic analysis for parallelizing compilers. *ACM Transactions on Programming Languages and Systems*, 18(4):477–518, July 1996.

[45] Luddy Harrison and Sharad Mehrotra. A data prefetch mechanism for accelerating general-purpose computation. Technical Report CSRD Technical Report 1351, Dept. of Computer Science, University of Illinois at Urbana-Champaign, May 1994.

[46] J. Hicklin, C. Moler, P. Webb, R.F. Boisvert, B. Miller, R. Pozo, and K. Remington. Jama: A Java matrix package. URL: http://math.nist.gov/javanumerics/jama, 2000.

[47] Mark D. Hill and Alan Jay Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12):1612–1630, December 1989.

[48] Glenn Hinton, Dave Sager, Mike Upton, Darren Boggs, Doug Carmean, Alan Kyker, and Patice Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, (Q1 2001), 2001.

[49] Richard L. Hudson, J. Eliot B. Moss, Amer Diwan, and Christopher F. Weight. A language-independent garbage collector toolkit. Technical Report Technical Report 91-47, University of Massaschusetts, Dept. of Computer Science, September 1991.

[50] Christopher J. Hughes and Sarita Adve. Memory-side prefetching for linked data structures. Technical Report UIUCDCS-R-2001-2221, University of Illinios at Urbana-Champaign, Department of Computer Science, May 2001.

[51] Intel Corporation. *Intel Pentium 4 and Intel Xeon Processor Optimization Reference Manual*, 2002.

[52] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, 1996.

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

[54] 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, April 1990.

[55] John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the Association for Computing Machinery*, 23(1):158–171, January 1976.

[56] Magnus Karlsson, Fredrik Dahlgren, and Per Stenstrom. A prefetching technique for irregular accesses to linked data structures. In *Proceedings of the 6th International Symposium on High Performance Computer Architecture*, pages 206–217, Toulouse, France, January 2000.

[57] Richard E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, Mar/Apr 1999.

[58] Gary Kildall. A unified approach to global program optimization. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 194–206, Boston,MA, October 1973.

[59] Thomas Kistler and Michael Franz. Automated data-member layout of heap objects to improve memory-hierarchy performance. *ACM Transactions on Programming Languages and Systems*, 22(3):490–505, May 2000.

[60] Alexander C. Klaiber and Henry M. Levy. An architecture for software-controlled data prefetching. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 43–53, Toronto, Canada, May 1991.

[61] Nicholas Kohout, Senugryul Choi, Dongkeun Kim, and Donald Yeung. Multi-chain prefetching: Effective exploitation of inter-chain memory parallelism for pointer-chaising codes. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 268–279, Barcelona, Spain, September 2001.

[62] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 81–87, May 1981.

[63] David J. Kuck, R.H. Kuhn, David Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the Eigth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 207–218, Williamsburg, VA, January 1981.

[64] Monica Lam, Edward R. Rothberg, and Michael E. Wolf. The cache performance and optimization of blocked algorithms. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, Santa Clara, CA, April 1991.

[65] Sorin Lerner, David Grove, and Craig Chambers. Composing dataflow analyses and transformations. In *Proceedings of the Twenty Ninth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 270–282, Portland, OR, January 2002.

[66] M. H. Lipasti, W. J. Schmidt, S. R. Kunkel, and R. R. Roediger. SPAID: Software prefetching in pointer- and call-intensive environments. In *Proceedings of the 28th Annual IEEE/ACM International Symposium on Microachitecture*, pages 231–236, 1995.

[67] Edward S. Lowry and C.W. Medlock. Object code optimization. *Communications of the ACM*, 12(1):13–22, January 1969.

[68] Chi-Keung Luk. *Optimizing the Cache Performance of Non-Numeric Applications*. PhD thesis, University of Toronto, Department of Computer Science, 2000.

[69] Chi-Keung Luk. Tolerating memory latency through software-controlled pre-execution in simulteneous mulithreading processors. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 40–51, Goteborg, Sweden, June 2001.

[70] Chi-Keung Luk and Todd C. Mowry. Compiler-based prefetching for recursive data structures. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, Cambridge, MA, October 1996.

[71] Chi-Keung Luk and Todd C. Mowry. Automatic compiler-inserted prefetching for pointer-based applications. *IEEE Transactions on Computers*, 48(2):134–141, February 1999.

[72] Nathaniel McIntosh. *Compiler Support for Software Prefetching*. PhD thesis, Rice University, May 1998.

[73] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[74] Sharad Mehrotra. *Data Prefetch Mechanisms for Accelerating Symbolic And Numeric Computation*. PhD thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, April 1996.

[75] David A. Moon. Garbage collection in a large Lisp system. In *Proceedings of the 1984 ACM Conference on Lisp and Functional Programming*, pages 235–246, Austin, TX, August 1984.

[76] Jose Moreira, Sam Midkiff, Manish Gupta, and Pedro Artiga. Numerically intensive java: Multiarrays. URL: http://www.alphaWorks.ibm.com/tech/ninja, 1999.

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

[78] Todd C. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Stanford University, Department of Electrical Engineering, March 1994.

[79] Todd C. Mowry, Monica S. Lam, and Annop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–72, October 1992.

[80] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[81] Toshihiro Ozawa, Yasunori Kimura, and Shin'ichiro Nishizaski. Cache miss heuristics and preloading techniques for general-purpose programs. In *Proceedings of the 28th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-28*, pages 243–248, Ann Arbor, Michigan, November 1995.

[82] Vijay S. Pai and Sarita Adve. Code transformations to improve memory parallelism. In *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-32*, pages 147–155, November 1999.

[83] Vijay S. Pai and Sarita V. Adve. Comparing and combining read miss clustering and software prefetching. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 292–303, Barcelona, Spain, September 2001.

[84] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. RSIM reference manual (version 1.0). Technical Report Technical Report 9705, Rice University, Dept. of Electrical and Computer Engineering, August 1997.

[85] 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, April 1994.

[86] Michael Paleczny, Christopher Vick, and Cliff Click. The Java HotSpot server compiler. In *Java Virtual Machine Research and Technology Symposium*, Monterey, California, April 2001.

[87] Parthasarathy Ranganathan, Vijay S. Pai, Hazim Abdel-Shafi, and Sarita V. Adve. The interaction of software prefetching with ILP processors in shared-memory systems. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 144–156, Denver, CO, June 1997.

[88] M. B. Reinhold. Cache performance of garbage-collected programs. In *Proceedings of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 206–217, Orlando, Florida, June 1994.

[89] Amir Roth, Andreas Moshovos, and Gurindar S. Sohi. Dependence based prefetching for linked data structures. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 115–126, San Jose, CA, October 1998.

[90] Amir Roth and Gurindar Sohi. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 111–121, Atlanta, GA, May 1999.

[91] Amir Roth and Gurindar S. Sohi. Speculative data-driven multithreading. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, pages 191–202, Monterrey, Mexico, January 2001.

[92] Shai Rubin, David Bernstein, and Michael Rodeh. Virtual cache line: A new technique to improve cache exploitation for recursive data structures. In *Proceedings of the 8th International Conference on Compiler Construction*, pages 259–273. Springer, March 1999.

[93] Vatsa Santhanam, Edward H. Gornish, and Wei-Chung Hsu. Data prefetching on the HP PA-8000. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 264–273, Denver, CO, June 1997.

[94] Charles W. Selvidge. *Compilation-Based Prefetching for Memory Latency Tolerance*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, September 1992.

[95] Olin Shivers. Control flow analysis in Scheme. In *Proceedings of the 1988 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 164–174, 1988.

[96] Alan Jay Smith. Sequential program prefetching in memory hierarchies. *IEEE Computer*, 11(12):7–21, December 1978.

[97] Alan Jay Smith. Cache memories. *Computing Surveys*, 14(3):473–530, September 1982.

[98] Burton Smith. Architecture and applications of the HEP multiprocessor computer system. In *SPIE Real Time Signal Processing IV*, pages 241–238, 1981.

[99] Yan Solihin, Jaejin Lee, and Josep Torrellas. Using a user-level memory thread for correlation prefetching. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 171–182, Anchorage, AK, May 2002.

[100] James W. Stamos. Static grouping of small objects to enhance performance of a paged virtual memory. *ACM Transactions on Computer Systems*, 2(2):155–180, May 1984.

[101] Artour Stoutchinin, Jose Nelson Amaral, Guang R. Gao, James C. Dehnert, Suneel Jain, and Alban Douillet. Speculative prefetching of induction pointers. In *Proceedings of the 10th International Conference on Compiler Construction*, pages 289–303, Genova, Italy, April 2001.

[102] Sun Microsystems. *UltraSparc III Cu User's Manual*, version 1.0 edition, May 2002.

[103] J.M. Tendler, J.S. Dodson, Jr. J.S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–26, January 2002.

[104] Dan Truong. Considerations on dynamically allocated data structure layout optimization. In *Workshop on Profile and Feedback Directed Compilation*, Paris, France, October 1998.

[105] Dan N. Truong, Francois Bodin, and Andre Seznec. Improving cache behavior of dynamically allocated data structures. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, pages 322–330, Paris, France, October 1998.

[106] Dean M. Tullsen and Susan J. Eggers. Effective cache prefetching on bus-based multiprocessors. *ACM Transactions on Computer Systems*, 13(3):57–88, August 1995.

[107] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, Ligure, Italy, June 1995.

[108] David M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, April 1984.

[109] Steven P. VanderWiel and David J. Lilja. A compiler-assisted data prefetch controller. In *Proceedings of the 1999 International Conference on Computer Design*, pages 372–377, Austin, TX, October 1999.

[110] Steven P. VanderWiel and David J. Lilja. Data prefetch mechanisms. *ACM Computing Surveys*, 32(2):174–199, June 2000.

[111] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Effective static-graph reorganization to improve locality in garbage-collected systems. In *Proceedings of the 1991 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 177–191, Toronto, Ontario, Canada, June 1991.

[112] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Caching considerations for generational garbage collection. In *Proceedings 1992 ACM Conference on Lisp and Functional Programming*, pages 32–42, San Francisco, CA, June 1992.

[113] Micheal E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proceedings of the 1991 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 30–44, Toronto, Canada, June 1991.

[114] Peng Wu, Albert Cohen, Jay Heflinger, and David Padua. Monotonic evolution: An alternative to induction variable substitution for dependence analysis. In *Proceedings of the 2001 International Conference on Supercomputing*, pages 78–91, Sorrento, Italy, June 2001.

[115] Peng Wu, Albert Cohen, and David Padua. Induction variable analysis without idiom recognition: Beyond monotonicity. In *Proceedings of the 14th International Workshop on Languages and Compilers for Parallel Computing*, Cumberland Farms, KY, August 2001.

[116] Youfeng Wu. Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 210–221, Berlin, Germany, June 2002.

[117] Youfeng Wu, Mauricio Serrano, Rakesh Krishnaiyer, Wei Li, and Jesse Fang. Value profile guided stride prefetching for irregular code. In *Proceedings of the 11th International Conference on Compiler Construction*, pages 307–324, Grenoble, France, April 2002.

[118] Yoji Yamada, John Gyllenhaal, Grant Haab, and Wen-mei W. Hwu. Data relocation and prefetching for programs with large data sets. In *Proceedings of the 27th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-27*, pages 118–127, San Jose, CA, November 1994.

[119] Chia-Lin Yang and Alvin R. Lebeck. Push vs. pull: Data movement for linked data structures. In *Proceedings of the 2000 International Conference on Supercomputing*, pages 176–186, Santa Fe, NM, May 2000.

[120] Chia-Lin Yang and Alvin R. Lebeck. A programmable memory hiearchy for prefetching linked structures. In *Proceedings of the 4th International Symposium on High Performance Computing*, pages 160–174, Japan, May 2002.

[121] Kenneth C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–40, April 1996.

[122] Lixin Zhang, Sally A. McKee, Wilson C. Hsieh, and John B. Carter. Pointer-based prefetching within the Impulse adaptable memory controller: Initial results. In *Proceedings of the Workshop on Solving the Memory Wall Problem*, Vancouver, BC, Canada, June 2000.

[123] Zheng Zhang and Josep Torrellas. Speeding up irregular applications in shared-memory multiprocessors: Memory binding and group prefetching. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 188–200, S. Margherita Ligure, Italy, June 1995.