RICE UNIVERSITY

Compiler Support for Software Prefetching

by

Nathaniel McIntosh

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Ken Kennedy, Noah Harding Professor
Department of Computer Science
Rice University

Keith Cooper, Associate Professor
Department of Computer Science
Rice University

Sarita Adve, Assistant Professor
Department of Electrical and Computer Engineering
Rice University

Houston, Texas

May, 1998

Compiler Support for Software Prefetching

Nathaniel McIntosh

Abstract

Due to the growing disparity between processor speed and main memory speed, techniques that improve cache utilization and hide memory latency are often needed to help applications achieve peak performance. Compiler-directed software prefetching is a hybrid software/hardware strategy that addresses this need. In this form of prefetching, the compiler inserts cache *prefetch* instructions into a program during the compilation process. During the program's execution, the hardware executes the prefetch instructions in parallel with other operations, bringing data items into the cache prior to the point where they are actually used, eliminating processor stalls due to cache misses.

In this dissertation, we focus on the compiler's role in software prefetching. In a set of experimental studies, we evaluate the performance of current software prefetching strategies, first for sequential benchmark programs running on a simulated uniprocessor machine, and then for a set of parallel benchmarks on a simulated distributed shared memory (DSM) multiprocessor. In these experiments, we employ a variety of enhanced efficiency metrics that allow us to focus on the compiler-related aspects of software prefetching. Based on the results of our experiments, we propose and experimentally evaluate a series of new compiler techniques for software prefetching. Our contributions include more powerful forms of compile-time reuse analysis, to reduce the frequency of useless prefetches, and new strategies for scheduling prefetches that reduce penalties incurred due to late prefetches. In the area of prefetching for DSM multiprocessors, we present a novel data-flow framework for analyzing communication patterns within parallel programs. We show how a compiler can use the information generated by this framework to provide better prefetching for long-latency coherence misses.

Acknowledgments

I would like to acknowledge the generous support of DARPA, the NSF, and IBM Corporation, whose financial assistance made my graduate studies possible. In addition, I would like to particularly thank the following people:

- my thesis committee, Ken Kennedy, Keith Cooper, and Sarita Adve, for their constant support for my research efforts
- Kathi Fletcher, for making her simulator available for my experiments
- Paul Havlak, for developing the CFG, SSA, and Value Numbering packages in the D system software
- Seema Hiranandani, for helping me obtain a collection of compiler-parallelized benchmark programs
- Kathi Fisler and Moshe Vardi, for unselfishly donating CPU cycles on their UltraSparc machine for overnight simulations
- my fellow students Kathryn McKinley, Chau-Wen Tseng, Jerry Roth, Taylor Simpson, Nenad Nedeljković, Dejan Mirčevski, Phil Schielke, and others, for their inspiration, help, and advice along the way
- Don Baker, Tasshi Dennis, Reinhard von Hanxleden, and Mike Paleczny, my bicycling partners, for accompanying me on countless grueling 7:30 AM rides
- Amy Pullen, my wife, for her patience and encouragement during my many years of graduate study

My thanks to all the people above, and many others, for making my time at Rice a productive and enjoyable one.

Contents

Illustrations

Chapter 1

Introduction

One of the major challenges that computer architects have faced during recent years is the widening gap between processor speed and main memory latency. The rate at which processors execute computations has grown much more quickly than the speed with which items can be fetched from memory. Cache memories are the primary tool that designers use to cope with this mismatch. As a result, cache design has been intensively researched in recent years, and cache hit rates are now a critical component of overall program performance. Cache memories are not perfect, however. In order to effectively exploit the cache, a program has to exhibit locality of reference, and even programs with good locality must incur cache misses now and then. For this reason, techniques that improve cache utilization and hide the latency of cache misses are very important.

This dissertation concentrates on a particular latency hiding technique: compiler-directed software prefetching. In this form of prefetching, the compiler inserts non-binding cache prefetch instructions into a program as it is being compiled. At run-time, the program issues prefetch operations to fetch data items into the cache before they are explicitly loaded, effectively hiding the latency that would ordinarily be incurred due to cache misses.

Previous researchers have established software prefetching as a viable method for improving cache hit rates. This research has explored the architectural issues surrounding prefetching in considerable detail. In our work we choose to focus on the compiler-related aspects of software prefetching. Our goal is to advance the state of the art with respect to compiler design in this area, and to provide useful information to future compiler designers.

The chief contributions in this thesis are as follows. First, we perform detailed performance evaluations of software prefetching for both sequential and parallel programs. Throughout these evaluations, our emphasis is on gauging the efficiency of the compiler algorithms employed. Based on the data from our evaluations, we propose a series of new compiler techniques for software prefetching.

For uniprocessor architectures, our contributions are in two main areas: new methods for detecting and eliminating useless prefetches, and new techniques for improving prefetch scheduling. We develop a pair of reuse analysis strategies that are more powerful than those applied in previous work, and we experimentally evaluate their effectiveness. Our results show that our more general methods are useful primarily for programs that make extensive use of imperfect loop nests, which are more difficult to analyze than perfect loop nests. In the area of prefetch scheduling, our new optimizations are effective in reducing late prefetch penalties. Performance improvements due to scheduling optimizations are fairly modest, however, since relatively few programs incur significant numbers of late prefetches on our simulated uniprocessor.

For multiprocessor architectures, we focus on the issues surrounding software prefetching for parallel programs running on distributed shared-memory (DSM) machines. These architectures present a series of additional challenges to the compiler, due to their increased overall memory latency and their more variable cache miss latencies. The results from our experimental study of compiler-parallelized benchmark programs running on a simulated DSM show that late prefetches are a much more serious problem than in case of uniprocessor architectures. To address this problem, we develop a new form of compiler analysis designed to predict the loops and references that cause coherence activity within a parallel program. We show how our compiler can use the information gathered by this analysis to provide better prefetch scheduling, and we present a set of partial experimental results that demonstrate the effectiveness of these techniques in practice.

The foundation for many of our analysis techniques is a new compiler framework for detecting *cross-loop* reuse: repeated data accesses that take place between loop nests, as opposed to within a single loop nest. This framework incorporates both data-flow analysis, to handle intra-procedural control flow, and array-section analysis, to compactly summarize regions of arrays accessed within loops. The information generated by the framework is useful for a variety of compiler tasks, including detecting and eliminating useless prefetches, and predicting the profitability of locality-enhancing loop transformations, such as loop fusion and loop reversal. Finally, it forms the basis of the analysis techniques we use for optimizing prefetching within parallel programs.

In order to support our experiments, we introduce a collection of enhanced metrics for characterizing and quantifying various aspects of prefetching performance. These metrics incorporate refinements designed to provide information on compiler behavior that is more useful than the data generated by existing methods.

Finally, we contribute a new approach to developing the compiler infrastructure needed to support research on software prefetching. By emulating prefetches using stores to compiler-generated "shadow regions", we avoid the prohibitively expensive step of developing an optimizing low-level compiler for the target architecture/instruction set. Our experiments demonstrate that using shadow arrays to emulate prefetches is a viable method for conducting research on software prefetching.

An outline of this dissertation is as follows. We begin in Chapter 2 by surveying the spectrum of latency-hiding techniques and other strategies with goals similar to those of software prefetching. We also summarize previous research on software prefetching, and highlight the unique contributions of our work relative to prior studies. Chapter 3 describes the internal architecture of our compiler, detailing the various analysis and transformation techniques we employ. In this chapter we also discuss the use of shadow arrays to implement prefetches. In Chapter 4, we present the system of efficiency metrics that we use to evaluate the performance of prefetching. Chapter 5 presents the results of our work on uniprocessor prefetching, including an experimental study of prefetching for a set benchmark programs running on a simulated uniprocessor architecture, and evaluations of our new uniprocessor compiler techniques. Chapter 6 describes our compiler framework for detecting cross-loop reuse. In Chapter 7 we address the problem of prefetching for DSM multiprocessors. We present our new compiler strategy for analyzing parallel programs, and we show how it can be used improve prefetching for this class of architectures. Finally, in Chapter 8 we summarize the contributions in this thesis and offer our concluding remarks.

Chapter 2

Related Work

In this chapter we summarize previous research on software prefetching, and we discuss other topics in the area of memory hierarchy management and memory latency hiding. An outline of this chapter is as follows. We begin in Section 2.1 by briefly discussing developments in cache design relevant to prefetching and to other latency-hiding techniques. In Section 2.2 we provide a survey of previous work on software prefetching, first for uniprocessor architectures and then for multiprocessors. Section 2.3 covers a variety of other hardware and software methods that hide memory latency and improve cache utilization. Section 2.4 contains a summary of our conclusions.

2.1 Cache design

The design of cache memories has been researched intensively over the last several decades [88]. We mention just a few of the important aspects of cache design that relate to software prefetching.

Multi-level cache hierarchies interpose multiple levels of cache between the processor and main memory. They provide a means of keeping overall miss penalties low without resorting to long cache hit latencies. Multi-level memory hierarchies typically enforce the *inclusion* principle, that is, the hardware requires that the contents of the cache at level K be a subset of the contents at K+1. The inclusion requirement is essential for efficient implementation of hardware cache coherence in multiprocessors [81]. In keeping with the widespread acceptance of multi-level hierarchies in modern workstation and multiprocessor designs, we assume a 2-level cache hierarchy for all of our simulation studies.

A *write buffer* is a small queue (2-32 entries) that stores pending write operations, typically placed between a cache and main memory or between two levels of cache in a multi-level hierarchy [88]. When a write is issued, the write buffer stores the written value and allows execution to resume before the write completes, thereby masking the latency of the write. When combined with other architectural features such as a relaxed memory consistency model [1, 32] and a write-through primary cache, an aggressive write buffer will allow the compiler to issue prefetches only for load instructions, since most or all store latency will be hidden.

A *lockup-free* cache provides a mechanism for the processor to continue issuing instructions even after a cache miss, provided that the processor does not use the register that is the destination of the load instruction that caused the miss [58]. Software prefetching requires that the cache be lockup-free (at least with respect to prefetch operations), since prefetch operations must execute in parallel with other cache accesses in order to be effective. Lockup-free caches are often implemented through use of Miss Status Holding Registers (MSHR's), which keep information about pending requests. A lockup-free cache is typically more effective as a latency-hiding mechanism if it is combined with other advanced hardware features, such as multiple instruction issue [49], dynamic branch prediction [64], dynamic scheduling [33], register renaming [51], and/or speculative execution [76]. Although the latency-hiding mechanism provided by lockup-free caches is very general-purpose, it is limited by the amount of instruction-level parallelism in the program, as well as the ability of the processor to exploit that parallelism [7, 28].

We categorize cache misses using the following classification scheme [47]:

- a *compulsory* or *cold start* miss corresponds to the very first access to a given block of data made by a program, causing the block to be brought into cache
- *capacity* misses are due to situations where the cache is not large enough to hold all of the blocks referenced by the program, causing a given cache block to be evicted and then subsequently retrieved.
- *conflict* misses result when blocks are evicted from the cache due to limited associativity. Given an N-way set associative cache, if the program uses a group of K data items (K > N) that map to the same cache set, when one item from the group is brought into the cache, it will displace another item within the group, causing a miss when the evicted item is subsequently referenced.
- *coherence* misses occur on multiprocessors with invalidation-based cache coherence protocols.[1] These misses take place when a given line in one processor's

[1] Invalidation-based cache coherence protocols are discussed in Section 7.2.

cache is invalidated by a second processor, then subsequently referenced again by the first processor.

2.2 Prior work on software prefetching

In this section we give a summary of previous work on compiler-directed software prefetching, first for uniprocessors and then for multiprocessors. Our work differs from the studies mentioned below in three important respects. First, we provide an experimental infrastructure that is at least comprehensive and complete as those employed in prior studies (and in some cases more comprehensive). In particular, we apply prefetching to realistic-sized complete application programs (not kernels or individual loop nests), we provide a compiler implementation that integrates most of the techniques developed by previous researchers (as well as our own), and we use a very detailed simulator, capable of gathering useful statistics not available when running on real hardware. This combination allows us to provide new insights into the behavior of compiler-directed software prefetching. Second, our work is the first to apply reuse analysis techniques that look beyond the level of individual loop nests, gathering information about memory access patterns across an entire procedure or program. Finally, our compiler is the first to provide a comprehensive framework for analyzing the coherence patterns in a parallel program, allowing it to begin addressing some of the challenges that shared-memory multiprocessors present to software prefetching.

2.2.1 Uniprocessor architectures

Porterfield, Callahan, and Kennedy produced some of the first results on software prefetching [18, 78]. Their compiler targeted array accesses within loops, issuing prefetches for array data an iteration in advance. Since the simulator they used (`PFCSim`) operated at the level of subscripted array accesses and not instructions, it was difficult for them to accurately characterize instruction overhead and scheduling efficiency. Their work introduced the idea of an *overflow iteration* for predicting the cache behavior of loops.

Klaiber and Levy explored a form of software prefetching in which prefetched data are stored in a separate "fetch buffer", distinct from the cache [55]. They developed the idea of computing a *prefetching distance* for each loop, as a means of determining how far in advance to issue prefetches. The prefetching distance is calculated as $\lceil \frac{C}{M} \rceil$, where C is the number of instructions during a single iteration of the loop, and M is the expected memory latency in cycles. They did not implement a compiler as part of their work; for their experiments, they used the Livermore Loops [71], which they instrumented by hand with prefetch instructions.

Chen, Mahlke, Chang, and Hwu performed a series of studies that compared prefetching into a separate buffer with prefetching directly into cache [22]. They used trace-driven simulation to model a hypothetical multiple-issue processor equipped with support for software prefetching. Their simulation results indicated that prefetching into a separate prefetch buffer outperformed prefetching into the main cache in most cases. The use of a separate prefetch buffer for prefetching has seen little acceptance in commercial processor designs, however, most likely due to the additional cost and design complexity required.

Mowry, Lam, and Gupta provided a comprehensive study of software prefetching, for both uniprocessors and multiprocessors [72, 74]. In contrast to most previous implementations (in which prefetches were inserted into benchmark programs entirely by hand), they integrated their support for prefetching into an optimizing compiler for a MIPS-like architecture [91]. They simulated their compiled programs using a cycle-level simulator that modeled the cache and memory of the target machine in detail. To experimentally validate their techniques, they applied software prefetching to a set of kernels and small programs drawn from the SPEC, SPLASH, and NAS benchmark suites [10, 86, 96]. In addition to issuing prefetches for array references with affine subscript functions, their compiler also issued prefetches for references that used indirection arrays. They found that prefetching was quite effective for hiding memory latency; in most cases stalls due to memory latency were reduced by 50% or more. We discuss Mowry's work in more detail in Section 2.2.3.

Bernstein, Cohen, Freund, and Maydan described their compiler implementation of software prefetching for the `PowerPC` architecture, and provided performance results for the SPEC92fp benchmarks [14]. Their compiler strategy was similar in spirit to that of Mowry *et al.*, although they targeted only references with affine subscript functions within inner loops. They were able to achieve performance improvements of (5-20%), which are significant given the relatively short memory latency (30 cycles) of the target architecture. They showed only total execution time improvement due to prefetching, since they did not employ simulation for their experiments.

Santhanam, Gornish, and Hsu evaluated the performance of software prefetching for the Hewlett Packard PA-8000 architecture [85]. In an experimental study, they applied their compiler implementation of software prefetching to the programs in the the SPEC95fp benchmark suite, resulting in execution time improvements of up to 100%, with a harmonic mean value of 26%. They also included discussions of a variety of issues related to prefetching performance, including cache conflicts, memory bank conflicts, instruction overhead, and instruction scheduling. As with the work by Bernstein *et al.*, the authors reported primarily execution times, as opposed statistics from simulations.

2.2.2 Multiprocessor architectures

Gornish, Granston, and Veidenbaum implemented a form of software prefetching as part of their compiler for the Cedar machine, a shared-memory multiprocessor [36]. They focused on using prefetching to gather blocks of data into a processor's local memory, as opposed to into the cache. They employed sophisticated compiler analysis to handle the problem of determining the safety of accessing locally cached items, because their target architecture provided no support for keeping the contents of the local memories up to date with the contents of global memory. Since prefetched data were not at risk of being displaced (as is the case with prefetching into a cache), their compiler issued prefetch operations as early as possible. They observed that the additional traffic produced by prefetching sometimes caused delays for other memory operations.

Mowry and Gupta reported results on the effects of software prefetching for parallel programs running on the DASH shared-memory multiprocessor [73]. In this particular study, they inserted prefetch instructions by hand, instead of using a compiler. The goal of their work was to assess the effectiveness of software prefetching in reducing memory latency penalties for programs running on shared-memory multiprocessors with hardware cache coherence. They concluded that prefetching can indeed be beneficial, provided that there is enough memory bandwidth available.

In subsequent work, Mowry performed studies using a similar set of parallel programs (the SPLASH benchmarks [86]), but with a compiler implementation of software prefetching [72]. The compilation strategy used was similar to that used in the Mowry's uniprocessor experiments, but with the addition of *exclusive-mode* prefetching for write references (see Section 7.2 for a description of this enhancement). This work provided additional data on the overall effectiveness of prefetching and demonstrated that exclusive-mode prefetching can be beneficial in terms of reducing network traffic on machines such as the DASH [65].

Tullsen and Eggers examined prefetching in the context of a bus-based, bandwidth-limited multiprocessor [94, 95]. Their study was based on off-line trace analysis, in which prefetch instructions were inserted into a previously generated program trace by consulting an "oracle". Their results were in three general areas. First, they found that for multiprocessors with very limited memory bandwidth, software prefetching will seldom produce any significant improvement, and can easily produce a degradation in performance. Second, they found that prefetching can exacerbate cache conflicts, but that the problems can often be overcome through the use of a victim cache [50]. Finally, they concluded that for the programs they considered, a significant fraction of the cache misses were due to coherence activity (*i.e.*, invalidations), and that this source of misses should be considered in the prefetching strategy. There is some question as to whether results derived from oracle-inserted prefetch studies can be compared with studies in which a compiler is employed, however. The oracle used by Tullsen and Eggers was a simple uniprocessor cache simulator. The trace being studied was run through the cache simulator, and instructions that caused misses with the simple simulator were marked for prefetching during the multiprocessor simulation run. This method of inserting prefetches may generate prefetches that are substantially different from those generated by a compiler. For example, in the oracle method, prefetches are inserted for conflict misses, which are for all practical purposes undetectable at compile time (using existing techniques). Prefetching conflict misses may improve some programs, but it may make others worse, depending on the nature of the conflicts. Similarly, if cache miss behavior is very "bursty" during the oracle simulation, then prefetches will also be issued in a bursty fashion (which again may not necessarily the case when prefetches are inserted by compiler). In the absence of a comparative study, we are uncertain whether oracle prefetching studies provide reliable information to a compiler developer.

More recently, Ranganathan, Pai, Abdel-Shafi, and Adve examined the performance of software prefetching for a class of advanced multiple-issue shared-memory multiprocessor architectures [84]. Their simulator modeled a processor that is designed to take advantage of instruction-level parallelism (ILP). These processors exploit ILP by means of features such as dynamic instruction scheduling, branch prediction, and non-blocking reads, which overlap instructions and hide latency. The authors found that in spite of the processors' hardware-supported latency-hiding features, programs still incurred significant memory latency penalties, and that software prefetching was found to be helpful in hiding this latency. The results also showed that prefetching placed additional strain on the processor's resources, creating significant problems in certain cases. The authors of this work inserted prefetch instructions into their benchmark programs by hand, simulating the various compiler techniques used by Mowry *et al.* Some of the optimizations they applied to improve performance were fairly aggressive; current-generation compilers would probably be unable to duplicate them automatically.

2.2.3 Mowry, Lam, and Gupta's prefetching algorithms

In this section we provide additional detail on some of the compiler techniques developed by Mowry, Lam and Gupta [74], since we have integrated many of them into our software prefetching scheme. The chief compiler-related contributions in the work of Mowry *et al.* are the use of *locality analysis* for selecting prefetch targets, *software pipelining* for effective scheduling of prefetches, and *loop unrolling* to reduce or eliminate the need for guarded prefetch instructions. The example in Figure 2.1 illustrates each of these steps. Part A of Figure 2.1 shows a loop nest containing references to an array. In this example, the cache line size is 16 bytes and each array element is 8 bytes long.

In part B of Figure 2.1, prefetch operations have been inserted and the loop has been split into three new loops. The first loop, the *prolog*, performs no computation but simply fetches data needed in the first few iterations of the original loop. Note that only a single prefetch is present for the array "`a`": the compiler's locality analysis has determined that the two references to "`a`" always access the same cache line (and thus only a single prefetch needs to be issued for the pair). The next loop, the *steady-state*, fetches data for later iterations while performing computation on array elements that have already been prefetched. The number of iterations in advance that items are fetched is known as the prefetching distance ("`PD`"), and in this example is equal to 8 iterations. The prefetching distance is calculated as $\lceil \frac{C}{M} \rceil$, where C is the number of instructions during a single iteration of the loop, and M is the expected memory latency in cycles. The third new loop is the *epilog* or *cleanup*; it executes the final iterations of the original loop without performing any prefetching. Initially, the prefetches in the prolog and steady-state are protected by guards to insure that they execute only on every other iteration (without the guards, two prefetches would be

A) Original loop nest

```
do i = 1, 1024
   ··· = x + a(i)
   ··· = y + a(i)
enddo
```

B) After prefetch pipelining

```
PD = 8
do i = 1, PD
   if (i mod 2 = 0)
      prefetch a(i)
   endif
enddo
do i = 1, 1024-PD
   if ((i+PD) mod 2 = 0)
      prefetch a(i+PD)
   endif
   ··· = x + a(i)
   ··· = y + a(i)
enddo
do i = 1024-PD+1, 1024
   ··· = x + a(i)
   ··· = y + a(i)
enddo
```

C) After loop unrolling

```
PD = 8
do i = 1, PD, 2
   prefetch a(i)
enddo
do i = 1, 1024-PD, 2
   prefetch a(i+PD)
   ··· = x + a(i)
   ··· = y + a(i)
   ··· = x + a(i+1)
   ··· = y + a(i+1)
enddo
do i = 1024-PD+1, 1024
   ··· = x + a(i)
   ··· = y + a(i)
enddo
```

Figure 2.1 Prefetch pipelining

issued for each cache line, effectively wasting half the prefetches). In part C of Figure 2.1, the prolog and the steady-state loops have been unrolled, allowing each guarded prefetch to be replaced by a single unconditional prefetch. The unrolling factor is selected based on the minimum effective stride of the references in the loop; the compiler uses the smallest unrolling factor that will allow all prefetches to be issued unconditionally. In combination, these techniques schedule prefetches appropriately, avoid unnecessary prefetches, and avoid overhead due to guards.

In order to reduce the fraction of prefetches issued that are useless (*i.e.*, those that hit in cache), Mowry uses temporal reuse analysis to recognize situations where a section S of a given array is repeatedly used within a given loop nest. It exploits this information by applying *loop peeling*, splitting a given loop nest into an initial "peel" loop (in which prefetches are issued for S), followed by the remainder of the loop (in which no prefetches are issued for S). Mowry's compiler also uses a form of multistage pipelining to support prefetching of references that access data through indirection arrays. This is accomplished by adding additional stages to the prefetch pipeline, to allow time to fetch the indirection array value before issuing the prefetch for the indirectly accessed location. For a single level of indirection, for example, the first preloop stage prefetches the indirection array at distance 0, the second preloop stage prefetches the indirection array at distance D and the indirectly accessed data at distance 0, and then the steady state stage of the pipeline issues prefetches for both types of data, with indirection arrays prefetched at distance $2 * D$ and indirectly accessed data prefetched at distance D. See Mowry's thesis for the details [72].

Mowry's study was limited in certain ways, and we have tried to address some of these limitations in our work. First, some of the the programs he used were fairly small. In his uniprocessor studies, ten of thirteen uniprocessor benchmarks had 3 or fewer subroutines and were under 200 lines. Larger programs can often be difficult to analyze, and they tend to have more control flow and more procedure calls than kernels or small benchmarks. The benchmark programs used in our experiments are complete applications, and are of a size that is more representative of scientific and numerical programs used in practice.

Second, Mowry's compiler used only intra-loop reuse analysis, with an emphasis on perfectly nested loop nests. As part of this dissertation we develop more general intra-loop reuse analysis techniques, and we apply them to the problem of detecting useless prefetches. We also develop a framework for analyzing *cross-loop* reuse, that is, reuse of memory locations that takes place across outer loop nests. We provide experimental results on how useful these techniques are in practice.

Finally, Mowry's compiler targeted explicitly parallel programs when compiling for multiprocessors. Because of the nature of the application programs that he targeted, his compiler had no specific knowledge about communication patterns within the programs. It was therefore difficult for the compiler to target specific sets of references or loop iterations that incur long-latency cache misses. We address this limitation in our work through the development of a compiler framework for detecting and exploiting coherence activity in compiler-parallelized programs.

2.3 Related areas

In this section we discuss various developments in compiler optimization and in computer architecture that are particularly relevant to software prefetching. Our goal is not to provide an exhaustive history of computer architecture and latency-hiding techniques, but rather to provide enough background material to allow the reader to understand the relative strengths and weaknesses of software prefetching compared to other methods.

In Section 2.3.1 we discuss hardware prefetching, and describe the advantages and disadvantages of hardware prefetching as compared to software prefetching. In Section 2.3.3 we discuss multithreaded architectures, which take an entirely different approach to memory latency hiding. Finally, we discuss a number of compiler techniques related to software prefetching, including instruction scheduling (Section 2.3.4) and compiler techniques for improving cache utilization (Section 2.3.5).

2.3.1 Hardware prefetching

We use the term "hardware prefetching" to refer to hardware support for bringing items into cache prior to any actual request for the items on the part of the processor. Hardware prefetching schemes are advantageous in that they do not require complicated compiler support, they do not require additional instructions to be added to a program, and they are capable of exploiting information that is available only at run time. Most forms of hardware prefetching rely on the assumption that future memory access patterns can be predicted from past memory access behavior. As a result, they are best suited to applications with fairly simple access patterns that can be easily detected at run-time.

The chief drawback of hardware prefetching is simply that it requires additional hardware, increasing the total system cost and possibly slowing down access to the primary cache. In addition, hardware schemes place more bandwidth demands on the memory subsystem than software schemes, and they tend to fetch more data items that go unused, compared with software schemes. A number of researchers have performed studies to compare the performance of hardware prefetching and software prefetching (see Section 2.3.2); in general, most have concluded that both techniques are roughly equal in terms of overall benefit.

Hardware prefetching has unfortunately not gained widespread acceptance in industry; few contemporary computer designs provide hardware prefetching of the sort described in the works discussed below.

Prior research on hardware prefetching

Some of the first work on hardware prefetching was by Smith [87, 88]. Smith concentrated on a simple sequential prefetching scheme called *one block lookahead*, in which the hardware fetches block $(i+1)$ on an access to (or cache miss on) block i. He used a cache simulator that analyzed traces generated from a set of programs running on an IBM mainframe; his results showed that for the machines of the day, hardware prefetching based on one-block lookahead was fairly successful at lowering the miss ratio, provided that the cache line size was small enough.

Chen *et al.* studied a variety of hardware prefetching schemes that are more aggressive than simple sequential prefetching [9, 20, 21]. Chen divided hardware prefetching schemes into two categories: *spatial* schemes, which base prefetching decisions only on current access patterns, and *temporal* schemes, in which the prefetching unit tries to predict future accesses by looking ahead in the program's instruction stream for hints on what to prefetch [20]. Temporal schemes require that the hardware maintain a *lookahead program counter*, which uses branch prediction to anticipate future values of the program counter. Both schemes associate state information about access streams with the particular load instruction that causes the access. This information is stored in a Reference Prediction Table, or RPT, in which entries are indexed by the actual address of the load instruction.

Jouppi proposed a form of hardware prefetching in which data items are brought into one of several hardware units called *stream buffers*, distinct from the cache [50]. This idea was subsequently extended and elaborated on by Kessler [77]. A stream buffer is a small FIFO queue, where each entry in the queue contains the address and data for a cache line. On a cache miss, the hardware first checks the head of the stream buffer. If the desired line is not present in the stream buffer, then the stream buffer begins fetching lines subsequent to the address in question. If the line is present in the buffer, then it is transferred to the cache, the entries in the queue are shuffled up one, and the buffer continues to fetch subsequent addresses. In the Jouppi's original work, sequential access is required: a non-sequential miss will cause the buffer to be flushed. Subsequent researchers extended stream buffers to handle strided accesses.

Dahlgren, Dubois, and Stenström examined the performance of sequential prefetching for DSM (distributed shared memory) multiprocessors. The simplest scheme they considered, fixed-degree sequential prefetching, fetches the next K blocks ahead of a particular location when a cache miss takes place for that location, where K is a fixed value referred to as the prefetching degree. In adaptive sequential prefetching, a set of hardware counters are used to record the number of useless/useful prefetches that take place within a given interval, and the degree of prefetching is increased or decreased depending on the values in the counters. In both cases, data was prefetched into the second level cache only. They performed a study of both schemes in using simulation of the SPLASH benchmarks [86], for finite and infinite second level caches. Both schemes were found to be effective in eliminating read miss penalties, and the adaptive scheme is effective in reducing the amount of additional interconnect bandwidth consumed by prefetching.

In a subsequent paper, Dahlgren and Stenström compare simple sequential prefetching with a variety of more complex schemes that detect accesses with non-unit strides [26]. The stride prefetching schemes they examined included a method similar to the spatial RPT-based technique proposed by Chen, as well as a method proposed by Hagersten [42] that detected strides by analyzing a cache miss history list. Surprisingly, the simplest scheme (sequential prefetching) outperformed the other forms of hardware prefetching in all but a small number of cases.

2.3.2 Comparisons of hardware and software prefetching

Chen and Baer performed a study that compared the performance of hardware and software prefetching for a given set of programs [21]. They found roughly equivalent performance for the two techniques, and their observations tended to confirm previous findings on the relative strengths and weaknesses of the two areas. In addition to their comparative study, they also proposed a hybrid form of prefetching, in which the compiler inserts prefetches at the granularity of entire variables or objects to bring items into the second level cache, and the hardware is then given the task of prefetching items into the primary cache.

Gornish considered both hardware and software prefetching in his dissertation [37]. He used two constructs for supporting hardware prefetching. The first was an instruction table, similar in nature to Chen's RPT, in which each entry is indexed by the address of a particular load instruction, storing any state information for the stream generated by that load. The second scheme used a stream table, in which state information on access streams is stored without any regard to the instructions that generate them. Stream table entry IDs are associated with each cache line; when a line is initially brought into the cache due to a miss, a new stream table entry is allocated for it and a prefetch is generated for the block immediately following it. Subsequent prefetches are triggered when prefetched lines arrive in the cache; the hardware determines the stream ID associated with each new arrival by looking the line up in the stream table. Gornish also examined the idea of using a combination of hardware and software prefetching, in which compiler-issued prefetches are used to handle irregular and non-constant streams, and to "jump start" hardware prefetching for sequential streams. The idea is to have the compiler issue prefetches for the first few lines (presumably just prior to the beginning of a loop) in a sequential or small-stride stream in order to get the hardware mechanism started, at which point the hardware takes over and does the prefetching during the bulk of the loop's execution.

Poulsen evaluated the performance of two types of software prefetching for shared-memory multiprocessors, and also compared prefetching with sender-initiated data forwarding [79, 80]. He used a pipelining-based prefetching scheme similar to that developed by Mowry, and a vector prefetching scheme, in which each prefetch operation inserted by the compiler fetches a fixed-length, strided vector of data. For his experiments, he used execution-driven simulation of parallelized versions of the Perfect benchmarks; the target machine is a uniform-access-time shared-memory multiprocessor with hardware cache coherence. In the work on data forwarding, he employed a profile-based scheme in which forwarding write operations were identified through the use of an initial profiling simulator run. His results showed that neither of the two prefetching schemes had an overwhelming advantage over the other, and he found that for some programs, data forwarding was considerably more effective than prefetching, given the particular architectural parameters used.

2.3.3 Multi-threaded architectures

Multi-threaded architectures hide latency by means of context-switching between different processes or threads of control within a program [3, 6, 56, 60, 97]. The type of scheduling varies depending on the design. Context switches can be made as often as every cycle, or they can be triggered by a particular event, such as an access to memory or a cache miss.

In order for a multithreaded architecture to be effective for a single program, the program in question must be parallelized; a single-threaded program would defeat the whole purpose of this type of architecture. Thus multithreaded architectures are currently attractive primarily as platforms for running scientific codes and other computationally intensive applications. The question of whether multithreaded architectures can provide a cost-effective general-purpose computing platform is still an open one. At least for the present, these architectures have not gained widespread acceptance.

2.3.4 Instruction scheduling

We use the term "instruction scheduling" to refer to a class of compiler techniques that attempt to reorder operations within a region of the program (typically a basic block) in order to increase utilization of processor resources and to hide the latencies of multi-cycle instructions [34, 57]. While instruction scheduling is capable in theory of hiding arbitrary latencies, it is generally limited in practice by the average basic block length relative to the memory latency. This form of scheduling can only hide the latency of an instruction X if there are other useful instructions that can be placed between X and the subsequent uses of values produced by X.

More powerful forms of instruction scheduling have been developed, including trace scheduling [31] and software pipelining [62]. These techniques use more advanced forms of compiler analysis that can exploit scheduling opportunities over larger regions of the program (loops bodies or frequently-executed sequences of blocks). Almost all of these techniques operate under the assumption that instructions have fixed and relatively small latencies, however. Kerns and Eggers developed an enhanced form of instruction scheduling that significantly improves scheduling for variable-latency load instructions [54]. This work explicitly takes into account the fact that load latency is uncertain, as opposed to using a fixed assumption that all loads either miss the cache or hit the cache. Nevertheless, this form of scheduling works best for short load latencies (2-10 cycles), and provides relatively little help in hiding very large cache miss latencies.

2.3.5 Compiler management of cache

A number of compiler techniques have been developed to help programs make better use of cache, including loop tiling, unroll-and-jam, loop interchange, and copy optimizations [19, 90, 98]. These methods work by analyzing the program's access patterns and then reordering portions of the program to improve the program's locality of reference, by arranging for computations that access the same memory locations to be executed closer together in time.

For the most part, these techniques are orthogonal and complimentary to prefetching. They reduce capacity misses, but they do not eliminate them altogether. As a result, prefetching is still plays a useful role even after locality-enhancing transformations.

2.4 Summary

Previous research has shown that software prefetching is a viable technique for memory latency hiding on both uniprocessor and multiprocessor architectures. Hardware prefetching offers comparable performance improvements, but at a considerably higher price, both in terms of additional bandwidth requirements and total system cost. Various other hardware and software techniques exist to hide latency, many of them complementary to software prefetching (as opposed to acting as a substitute for it).

In comparison with previous studies of software prefetching, our work focuses primarily on the compiler's role. We use a robust experimental infrastructure that incorporates many of the strengths of previous approaches, including detailed simulation, representative application programs, and a comprehensive compiler implementation. Our work also extends the compiler analysis used in previous studies. These extensions include more aggressive forms of reuse analysis, and compiler analysis of coherence activity within parallel programs.

Chapter 3

Compiler Overview

In this chapter we describe the implementation details of our compiler framework for software prefetching. Section 3.1 explains some of the difficulties of developing compiler infrastructure to support software prefetching, and outlines the novel approach we have taken to leverage existing vendor compilers for our work. Section 3.2 presents the various stages in our compiler at a high level. In Section 3.3, we discuss the handling of loops in detail, including target loop selection, reuse analysis, loop peeling, prefetch distance calculation, and ordering of transformations. Section 3.4 describes the new uniprocessor techniques that are part of our compiler, including enhancements to group-temporal reuse analysis, enhanced loop peeling, loop strip-mining to moderate code expansion, cross-loop reuse analysis to eliminate useless prefetches, and two transformations to improve prefetch scheduling: prolog overlap and outer loop pipelining. In Section 3.5, we wrap up with a summary of the material covered in this chapter.

3.1 Prefetch implementation

3.1.1 A researcher's dilemma

One of the critical components of a good software prefetching package is a robust optimizing compiler for the target instruction set and architecture. Without the support of an industrial-strength optimizer and register allocator, the loop transformations that prefetching is based on can cause enough instruction overhead to swamp any gains derived from memory latency hiding.

This need creates a problem for researchers working on software prefetching. Writing an optimizing compiler for even a single machine requires a great deal of work. Architectures and instruction sets change very rapidly, however, which means that much of the work invested in developing a good compiler back-end can be lost in just a few years (or less). Because of the intense competition and short design cycles in the present-day computer industry, it may well be that by the time a researcher finishes writing a compiler for a particular architecture, the machine has already become obsolete. If the researcher is truly unlucky, the entire company that created the machine may have folded.

Prefetches can be implemented in a more portable and compiler-independent fashion by using subroutine calls, but this solution typically comes at the cost of very high instruction overhead, due both to the procedure call itself and to the undesirable effects on optimization of the inner loop that contains the call.

Faced with this dilemma, we have chosen an intermediate solution. Our strategy provides prefetches at a reasonable level of instruction overhead, but also gives our framework some degree of machine- and compiler-independence, allowing us to use vendor compilers without modifications. It does, however, require that we simulate our benchmark programs instead of executing them directly on a machine that supports prefetching. We describe our strategy in the remainder of this section.

3.1.2 Shadow array implementation of prefetches

We implement our approach by associating a "shadow region" with each variable that might be the target of a prefetch. The shadow region is a section of the program's address space that is used to signal prefetches for a given variable. When the compiler decides to issue a prefetch for a variable V, instead of actually generating a prefetch instruction, it instead generates a store to V's shadow region. At run-time, the simulator maintains a mapping between shadow regions and variables; when the program issues a store to a location known to be within a shadow region, the simulator interprets the operation as a prefetch of the corresponding variable.

Figure 3.1 shows an abstract view of a program's address space, with a variable "`xyz`" and its corresponding shadow region. When the simulator sees a store to the location S+4, for example, it treats the instruction not as a store but as a prefetch of the location K+4.

In order to support this approach, the compiler must identify all of the variables in the program that might be prefetched, allocate appropriately-sized shadow regions for them, and generate calls to the runtime library to inform the simulator about the shadow regions. Shadow regions for collections of related variables can be aggregated in certain circumstances, simplifying the management of the mapping information.

There are a variety of tactics that can be used to create and manage shadow regions. One possibility is to create a single shadow region for the entire data segment

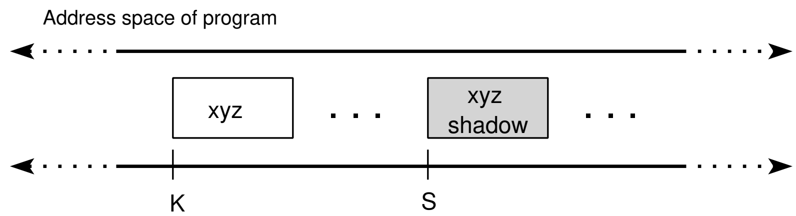


Figure 3.1 Implementing prefetches with shadow regions

of the process. The compiler can then generate pointers into the shadow region by adding an offset to the address of the target variable. We use a slightly different method in our compiler, however.

3.1.3 Shadow regions for Fortran programs

For programs written in Fortran 77, the task of creating shadow regions is simplified, since Fortran 77 does not support dynamic memory allocation (all variables must be declared statically at some point in the program). Shadow regions themselves can thus be declared as new array variables within the program, reducing the level of runtime support needed. Our compiler adds shadow regions by creating a shadow array of the appropriate size for each array variable that might be the target of a prefetch. For arrays that appear in common blocks, we create a single large shadow region for the entire block.

Figure 3.2 shows an example of the transformations made by our compiler to support shadow array prefetching. In the original subroutine there are two arrays that need to be prefetched, "`a`" and "`b`"; the first is a formal parameter and the second is declared in a common block. The transformed program, shown in an abstracted form, contains shadow regions for all arrays that are targets of prefetches. Note that a shadow region for the array "`b`" has been added to the common block "`cb`", and a call to a runtime routine has been added to inform the simulator about the shadow region.

When an array is passed to a function as a parameter, such as the array "`a`" in Figure 3.2, the compiler is generally unable to statically determine the shadow region for the array, since there may be many different callers of the function, each passing a different array. We solve this problem using interprocedural analysis: the compiler

Original subroutine

```
subroutine mumble(a, n, t)
   common /cb/ b
   integer i, t, n
   integer a(n), b(100)
   do i = 1, n
      t = t + a(i) + b(i)
   enddo
end
```

Transformed subroutine

```
subroutine mumble(a, n, t, aShadow)
   common /cb/ b, bShadow
   integer i, t, n
   integer a(n), b(100)
   integer aShadow(n), bShadow(100)
   call $declareshadow(b, bShadow, 100*4)
   ...
   do i = 1, n
      aShadow(i+P) = 0     prefetch a(i+P)
      bShadow(i+P) = 0     prefetch p(i+P)
      t = t + a(i) + b(i)
   enddo
end
```

Figure 3.2 Shadow array prefetching example

identifies all situations where a calling procedure passes an array as a parameter to a callee, and patches the call site to pass the array's shadow region in addition to the array itself. The details of this process are given in Appendix A.

3.1.4 Tradeoffs involved in the use of shadow arrays

The chief disadvantage of using shadow arrays, aside from the time required to implement the interprocedural analysis and transformations to support them, is that they can result in a slightly higher level of instruction overhead, relative to actual prefetch instructions. This additional overhead is created because the compiler must generate an address to an entirely separate array, as opposed to an offset from the address of the real array. With an actual prefetch instruction, the optimizer can sometimes recognize situations where the address used by a load of an array and the address used by a prefetch of the array share the same base register. The compiler can often exploit this redundancy, allowing it to use fewer instructions when generating the address of the prefetch. According to Mowry, this effect can result in a significant reduction in instruction overhead, if properly exploited by the optimizer [72]. Although our framework does not permit this sort of optimization, we have been able to achieve reasonably low levels of instruction overhead, as demonstrated in Section 5.5.2.

3.2 Compilation stages

Figure 3.3 gives a high-level outline of the stages of our compiler. The compiler operates in a source-to-source fashion, reading and writing Fortran code. The most interesting aspects of the compilation process have to do with loop analysis (including array-section analysis) and loop transformation. These phases phases are described in Figures 3.4, 3.5, and 3.6.

Phase	Remarks
1. Front end processing	Read and typecheck Fortran source; build AST (abstract syntax tree).
2. Control flow analysis	Build Control-Flow Graph (CFG) and Static Single Assignment (SSA) graph for procedure.
3. Array-section analysis	Construct array-section summaries for references and loops. Apply section-based intraprocedural data-flow analysis (see Chapter 6).
4. Loop analysis	Described in Figures 3.4 and 3.5.
5. Transformations	Described in Figure 3.6.
6. Output	Generate output Fortran source based on transformed AST.

Figure 3.3 Compilation stages

3.3 Loop analysis and transformation

Our compiler analyzes loops from innermost to outermost. Figures 3.4 and 3.5 show the stages of the analysis for inner and non-inner loops, respectively. Once the analysis phases have determined the proper sequence of transformations to use, applying them to the code is a complex but fairly mechanical process.

3.3.1 Target loop selection

We currently insert prefetches only for references contained within inner loops, since these references account for the bulk of the cache misses in our benchmark suite.

Stage	Remarks
1. Preliminaries	Determine loop step and loop trip count, if possible.
2. Reference partitioning	Partition references into equivalence classes based on array name/extent. Analyze reference subscript functions. Identify references accessed via indirection arrays.
3. Temporal reuse analysis	Identify useful self-temporal and group-spatial reuse via dependence analysis. Store reuse information for later use in stages 5 and 8.
4. Spatial reuse analysis	Use symbolic analysis to identify references with self-spatial reuse. Build sets of references that share group-spatial reuse.
5. Mark leading references	Use temporal and spatial reuse information to select initial set of references that need prefetching ("leading references").
6. Loop pipelining decision	Decide whether to apply pipelining to the loop (loop may contain no leading references at this point).
7. Loop stride analysis	Compute "effective stride" for all leading references. Compute unroll amount based on smallest effective stride within loop.
8. Peeling analysis	Initial loop peeling analysis: identify set of leading references for which loop peeling might be profitable.
9. Prefetching distance analysis	Compute prefetching distance for loop.

Figure 3.4 Analysis stages for inner loops

Phase	Remarks
1. Peeling analysis	Determine if it is profitable to apply loop peeling to the current (outer) loop.
2. Pipeline level selection	Determine if this loop should be pipelined instead of the inner loop it contains (see Section 3.4.5 for more information).
3. Prolog overlap analysis	Determine whether prolog overlap transformation is legal and/or profitable (see Section 3.4.4 for a description of this transformation).

Figure 3.5 Analysis stages for non-inner loops

Some existing compiler implementations apply prefetching to non-inner-loop references, and others do not. In particular, in the work by Mowry *et al.*, the compiler does target these references, whereas in the frameworks used by Santhanam *et al.* and by Bernstein *et al.*, the compiler does not target non-inner loop references [14, 74, 85]. In Section 5.5.1, we present empirical data on the percentage of cache misses due to this class of references.

In our base compilation strategy we suppress prefetching for inner loops containing procedure calls (with the exception of Fortran intrinsic and generic functions) and for loops that contain I/O.

3.3.2 Reuse analysis

We use Wolf's terminology to classify reuse between array references into one of four categories: self-spatial reuse, group-spatial reuse, self-temporal reuse, and group-temporal reuse [98]. All four of these types of reuse are important for software prefetching; our compiler exploits each type of reuse in a variety of ways.

Dependence analysis forms the basis of our temporal reuse detection algorithm. We cast the problem of finding useful reuse in terms of interrogating the dependence graph for the candidate loop nest. We configure our dependence tester to add *input* dependence edges to the dependence graph in addition to true- and anti-dependence edges [16]. Our compiler utilizes a dependence testing package developed by Goff, Kennedy, and Tseng [35]. Not every dependence corresponds to reuse that the compiler can exploit. Our framework looks for dependences that the analyzer has flagged as *consistent* [35]. In addition, we consider only loop-carried dependences with non-symbolic dependence distances [82].

Dependence information is exploited as follows. Dependences carried at the innermost loop level[2] are used to select the set of "leading" references for the particular inner loop. If a reference R has a self-dependence at the innermost loop level, or if R is the sink of a loop-carried or loop-independent dependence at the innermost level, then reference R is marked as non-leading (i.e. not requiring prefetching). Dependences not carried at the innermost loop level are used to drive loop peeling. If a reference R has a self-dependence at level K, or if R is the sink of a loop-carried or loop-independent dependence at level K, where K is some non-inner loop level, then we flag R and the loop at level K for peeling. In our default loop peeling strategy, we only consider dependences between references that are enclosed by exactly the same set of loops, in order to be consistent with the reuse analysis methods used by Mowry *et al.* In Section 3.4.1 we describe how to relax this restriction.

To detect self-spatial and group-spatial reuse, our compiler uses a symbolic analysis package developed by Havlak [46]. This package greatly simplifies the analysis of array references by expressing loop bounds and subscript functions in a canonical sum-of-products form. After the compiler applies symbolic analysis, detecting self-spatial reuse is simply a matter of classifying the induction variable use in each subscript position and then analyzing the stride of the reference. We identify group-spatial reuse by symbolically subtracting the subscript functions of pairs of references; our method is loosely based on that developed by Carr, McKinley, and Tseng [70].

When applying reuse analysis, the compiler needs to take into account the cache line size and whether the cache is write-back or write-through. In the case of a write-through cache, we consider only input dependences (edges between two read references) when performing the reuse analysis, and we suppress prefetches for writes/stores.

Multi-level cache hierarchies make things particularly complicated, since they introduce the question of whether to use the L1 cache size or the L2 cache size when performing the analysis. If a reference always hits in the L1 cache, then eliminating the prefetch for the reference is a win. If the compiler identifies a reference that always misses in the L1 cache and always hits in the L2 cache, however, then the decision of whether to suppress the prefetch depends on the L2 miss penalty and on

[2]We use Allen and Kennedy's terminology for classifying dependences with respect to loop levels [5]

the cost of issuing the prefetch. Mowry studied this problem in his dissertation; he characterized prefetch cost in terms of instruction overhead, increased conflicts, and increased contention for the primary cache tag array [72]. In general, if the L2 miss penalty takes K cycles and the cost of issuing a prefetch is P cycles, then it makes sense to issue the prefetch when $P < K$. In our particular simulated architecture, the cost of issuing a prefetch is nearly always less than the cost of an L2 cache hit. Because of this fact, we target the L1 cache and not the L2 cache when performing our reuse analysis.

3.3.3 Loop peeling analysis

Our default loop peeling implementation is based on the algorithm developed by Mowry *et al.* The compiler performs the analysis in two stages, as follows. When analyzing a reference within an inner loop, an initial phase (described in Section 3.3.2) determines whether loop peeling is profitable for the reference in question. The compiler ignores cache size when making this first peeling profitability decision. It adds the IDs of candidate references to a list associated with the loop that carries the dependence (the "peel reference list"), along with a preliminary peeling degree. Once all of the loops enclosed in an outer loop have been processed, it selects a single peeling degree for the loop by taking the maximum of the distances recorded for each of the entries on its peel reference list. Note that a reference may be marked for peeling at multiple loop levels.

The second phase of the peeling analysis takes into account the cache size. The compiler computes an estimate of the cache volume for the loop (the number of lines the loop will bring into the cache), and disables loop peeling in situations where the it predicts that the cache does not have enough capacity to hold the entire section in question throughout the loop's execution.

3.3.4 Prefetch distance calculation

Since our compiler is operating at the level of Fortran statements and not assembly language instructions (or some other low-level intermediate form), calculating the prefetching distance presents a problem: our compiler cannot statically count the number of instructions in the bodies of the inner loops being compiled.

We solve this problem by using off-line trace analysis. We compile the program and run it using our simulator, which counts the number of instructions in each inner loop body and writes this information to a database. The information gleaned from the instruction count database is then used to calculate an appropriate prefetching distance.

A final note: our compiler assumes that the memory latency is unknown at compile-time. Rather than computing the prefetching distance directly, it generates a distance-computation function and adds it to the transformed program. When the transformed program is executed, the runtime library calls the distance-computation function, passing it the memory latency of the machine. The function then computes and stores prefetching distances on the fly for all of the loops in the program. This mechanism allows us to use a single program executable on set of machines with different memory latencies, provided that the machines have similar cache configurations.

3.3.5 Transformation ordering

Figure 3.6 gives an algorithm that captures the order in which transformations are applied. The compiler walks through the loops in the procedure from outermost to innermost, applying loop peeling and loop pipelining to loops that have been marked as candidates.

3.3.6 Source code size

Figure 3.7 shows the approximate number of lines of source code (with comments) devoted to various major tasks in our compiler. The numbers shown reflect only prefetching-specific source code, and do not include any of the base compiler infrastructure (parser, abstract syntax tree, symbol table, control-flow graph, static single assignment support, symbolic expression analysis, dependence analysis, or scalar interprocedural analysis).

3.4 New techniques

The base compiler algorithms discussed in Section 3.2 (loop pipelining, loop peeling, marking of leading references, etc.) were developed by Mowry *et al.*, although we do not always use the same set of tools to implement them. In the remainder of this section we introduce some of the new uniprocessor compilation techniques that are part of this dissertation.

```
function      TransformLoop(L)
parameters:   L – AST of header node of the original loop
returns:      transformed version of L (list of AST nodes)
{
              if (L marked for loop peeling) {
                  enable prefetching for references on L's peel reference list
                  PL = TransformAST(L.body)
                  disable prefetching for references on L's peel reference list
                  ML = TransformAST(L.body)
                  return MakeList(PL, ML)
              } else if (L is marked for pipelining) {
                  generate prolog, steady-state, and epilog stages for L
                  return MakeList(prolog, steady-state, epilog)
              }
}

function      TransformAST(N)
parameters:   N – AST node in original function
returns:      transformed version of N
{
              if (N is a loop node) {
                  return TransformLoop(N)
              } else {
                  for each descendant D of N {
                      compute TransformAST(D)
                  }
                  return copy of N with transformed descendants
              }
}
```

Figure 3.6 Transformation overview

Component	Lines of C++ source code
prefetching analysis and transformations	49,000
array-section analysis	20,000
shadow array support	5,000
simulator glue code	1,000
total	75,000

Figure 3.7 Compiler source code size

3.4.1 Enhanced loop peeling

The base method we use for detecting group-temporal reuse (described in Section 3.3.2) is designed to provide the same functionality as the system used in the compiler developed by Mowry *et al.* Their reuse analysis strategy is oriented primarily towards perfectly nested loop nests. When applied to imperfect nests, it may not recognize certain types of group-temporal locality. Consider the imperfectly nested loop nest in Figure 3.8. For this loop nest, the analysis used by Mowry *et al.* would detect reuse between the two references in the "`do i`" loop, but it would fail to detect the reuse between the references in the "`do i`" and "`do m`" loops.

```
do k = 1, n
   do j = 1, n
      do i = 1, 100
         z = z + a(i,j,k) + a(i,j+1,k)
      enddo
      ...
      do m = 1, 100
         z = z + a(m,j,k)
      enddo
   enddo
enddo
```

Figure 3.8 Reuse analysis for imperfectly nested loops

The difficulty in extending group-temporal reuse analysis to handle non-identically nested references is that references in different loops may be controlled by different loop bounds. For example, in Figure 3.8, if the bounds on the "`do m`" were 99 to 100 instead of 1 to 100, there would be very little useful reuse, but there would still be a dependence present. In this case, dependence analysis provides "may access" information instead of "must access" information, which is what we need for this application. We solve this problem by applying array-section analysis.

Array-section analysis

Array-section analysis is a technique for summarizing the region(s) within an array that are accessed during some portion of the program [11, 15, 17, 45]. These summary representations provide a compact way of capturing the array access patterns, making them attractive for applications in which large portions of the program need to be considered, as opposed to single loop nests. Our particular implementation represents array accesses using *Data Access Descriptors* [12], or DADs.

The region within an array accessed by a given subscripted reference depends on the context that surrounds the reference. For example, consider the first array reference `a(i,j,k)` in Figure 3.8. If we consider this array reference in isolation, then it can be thought of as accessing the single element at `a(i,j,k)`. If we take the "`do i`" loop into account, then the reference can be thought of as accessing the vector `a(1:p,j,k)`, and so on.

We use the term *array section at level M* to refer to the region within an array that is accessed at a given loop level. More formally, given a subscripted reference nested within N loops, the array section at level K for the reference is the region accessed within the array when *A)* the loop induction variable(s) at levels 1 through K inclusive are viewed as invariants, and *B)* the induction variables at levels K+1 and above are allowed to vary.

Relaxed group-temporal reuse analysis

We employ array-section analysis as follows. When we find a dependence that meets our criteria, we apply a section-based bounds check to insure that the regions accessed by the source and sink of the dependence are conformable, *i.e.*, overlap significantly. An algorithm for the region check is shown in Figure 3.9. By incorporating this additional step, we can safely capture useful group-temporal reuse in imperfect loop nests, while excluding dependences that do not correspond to useful reuse. We refer to the this new scheme as "Relaxed" group-temporal reuse analysis, and the default scheme as "Restricted".

```
function      CheckGroupTemporalDependence(R1, R2, L)
parameters:   R1 – reference at source of dependence
              R2 – reference at sink of dependence
              L – level of loop that carries dependence
returns:      true if dependence is useful, false otherwise
{
              Build Data Access Descriptor D1 for R1 (with respect to level L)
              Build Data Access Descriptor D2 for R2 (with respect to level L)
              If (D1 contains D2) or (D1 is a shifted copy of D2) {
                 return true
              } else {
                 return false
              }
}
```

Figure 3.9 Section-based group temporal dependence check

When "Relaxed" group-temporal reuse is in effect, we also use a different algorithm for the second phase of loop peeling analysis, in which cache capacity is taken into account. The algorithm used by Mowry for determining the loop traffic for a given nest operates by tallying the contributions of each leading reference. This algorithm handled imperfectly nested loops, but the reuse analysis method he used did not. For example, although the imperfect loop nest shown in Figure 3.8 contains group-temporal reuse carried by the "`do j`" loop, this reuse would not be considered by his compiler. The compiler would create an equivalence class for each inner loop, and would mark one reference from each loop as "leading". As a result, the loop traffic for the loop would be over-estimated, potentially disabling loop peeling, increasing the fraction of useless prefetches.

The approach we use for calculating the cache traffic for a loop is based on array-section analysis. Similar to Mowry's algorithm, we visit loops from innermost to outermost. For each loop, the compiler builds a summary of the region accessed by every array reference, again employing Data Access Descriptors. Once the set of `DAD`s is constructed for the loop, the compiler then compares the sections for each array, coalescing together `DAD`s that are substantially contained in each other. Finally, the compiler sum the volumes of the `DAD`s that remain. This process is described in more detail in Appendix B.

There are two advantages to our algorithm. The first is that it recognizes reuse in imperfect nests, as in Figure 3.8. The second is that the use of array-section analysis allows us to recognize a larger class of references that have group-temporal reuse. These advantages yield an algorithm that produces more precise loop traffic estimates.

3.4.2 Loop strip-mining to reduce code expansion

The cache line size of the target machine is the main factor used to determine the degree unrolling when applying prefetching transformations. If a loop contains a unit-stride access to an array with element size K, then the unrolling degree will be set to $\frac{CLS}{K}$, where CLS is the cache line size. For machines with large line sizes (64-128 bytes), the optimal unrolling degree can often be in the 16-32 range.

Large unrolling degrees can have a number of unpleasant consequences. First, although the loop body produced by a large unrolling degree may enhance optimization opportunities and reduce loop overhead, it can also increase register pressure. If the increase pressure forces the register allocator to spill values within the loop, then the larger unroll degree can actually increase execution time. Second, increases in code size can change instruction cache behavior, with unpredictable and occasionally dramatic results. Finally, the code expansion from unrolling can result in very long compile times and drastically increased object file size. A doubling or tripling of the compile time and executable size may not be acceptable to every user. For these reasons, our compiler includes support for a "cutoff" in situations where the unrolling degree is predicted to cause problems. Our goal is to create a loop nest that still derives some benefit from unrolling, but also yields a more moderate compile time.

Our algorithm works as follows. We estimate the register pressure in the loop by scanning the statements in the loop body, looking for subscripted array references and other values that are live throughout the entire loop's execution (scalars, constants, etc.). Array references are grouped into equivalence classes if the compiler predicts that they will share the same base register (example: `a(i,j)` and `a(i+1,j)`, and each equivalence class is only counted once. References are also flagged if the compiler predicts that they will not need additional registers when the loop is unrolled. For example, the register requirements for a collection of loop-invariant references are unlikely to increase as a result of unrolling. We use the data gathered by this phase to estimate the the maximum unrolling degree possible before register pressure becomes a major problem. We call this degree the "adjusted" unrolling degree.

A) Pure unrolling

```
do i = 1, m, 16
   prefetch a(i+PD)
   w = w + a(i)
   w = w + a(i+1)
   w = w + a(i+2)
   w = w + a(i+3)
   w = w + a(i+4)
   ...
   w = w + a(i+13)
   w = w + a(i+14)
   w = w + a(i+15)
enddo
```

B) Limited unrolling with strip-mining

```
do ii = 1, m, 16
   prefetch a(ii+PD)
   do i = ii, min(ii+3,m), 4
      w = w + a(i)
      w = w + a(i+1)
      w = w + a(i+2)
      w = w + a(i+3)
   enddo
enddo
```

Figure 3.10 Strip-mining in combination with unrolling

Once we have computed the adjusted unrolling degree, we use a combination of strip-mining and loop unrolling in the steady-state stage of the prefetch pipeline. Figure 3.10 gives an illustration. Part A of the figure shows a loop nest where only unrolling is used, with an unroll degree of 16. In part B, the inner loop is unrolled using the adjusted unrolling degree (4), and then surrounded by an auxiliary "strip" loop. We measure the results of this technique in practice; the results are given in Section 5.6.3.

Other researchers have implemented loop unrolling cutoff mechanisms as well. Mowry's compiler limits unrolling for a given loop when the loop body reaches a certain size, as opposed to estimating register pressure [72].[3] Bernstein *et al.* also use heuristics to limit the unrolling factor for a loops that require prefetching [14].

[3]Although not specifically discussed or evaluated in his thesis, Mowry's compiler also supports a similar form of loop strip-mining to reduce code expansion [75].

3.4.3 Cross-loop reuse analysis for prefetching

In this section we give a high-level description of how we use cross-loop reuse analysis to augment intra-loop analysis during software prefetching. The details of our cross-loop reuse analysis framework are described in Chapter 6.

```
do i = 2, m
   r = bn(i) / an(i-1)
   an(i) = an(i) - r * cn(i-1)
   fn(i) = fn(i) - r * fn(i-1)
enddo
gn(m) = fn(m) / an(m)
do i = 1, (m-1)
   gn(m-i) = (fn(m-i) - cn(m-i) *
            gn(m-i+1)) / an(m-i)
enddo
```

Figure 3.11 Example of cross-loop reuse

Consider the example program fragment in Figure 3.11, taken from the SPEC95 benchmark `apsi`. When analyzing the second loop nest, a compiler that relies only on intra-loop reuse analysis would apply prefetching to all of the references in the second loop nest ("`fn(m-i)`", "`an(m-i)`", etc.). Depending on the cache capacity and the loop bounds, however, some of the sections accessed in the second loop nest may already be in the cache (as illustrated by the boxes). If the compiler can detect this type of reuse, then it can disable prefetching for the references in question, effectively reducing the number of useless prefetches.

We exploit cross-loop reuse as follows. Give a reference R within a loop L, we construct an array-section descriptor (DAD) S_R that summarizes the region accessed by R. When deciding whether to issue prefetches for R, we try to find another section S_Q that is already resident in the cache when L begins its execution, where the S_R's region is contained in S_Q's region. This is accomplished by solving a series of data-flow equations over the function's control-flow graph; during the analysis, sets of array sections are propagated along edges in the CFG, taking into account the size of the cache and the reuse among the various arrays. If the data-flow analysis can establish that

1. the section S_Q is accessed every path in the CFG flowing into L, and
2. S_Q is not displaced from the cache between the point at which it is brought into cache and the start of L, and
3. the geometric region of S_Q contains the region of S_R

then the prefetch for R in L is useless and can be eliminated. We evaluate the effectiveness of this form of reuse analysis in Section 5.6.1.

3.4.4 Prolog overlap

We have developed a new transformation called *prolog overlap* that is intended to provide better scheduling for prefetches issued during the prolog stage of the prefetch pipeline. In this section we describe the prolog overlap transformation and discuss the compiler support needed to implement it.

Prolog overlap is a form of rotation or shifting in which the three stages in the prefetch pipeline (prolog, steady-state, and epilog) are rearranged in order to provide better latency-hiding for prefetches issued in the prolog stage. Figure 3.12 gives an illustration of the technique. The original loop nest is shown in part A; part B shows a standard software prefetching pipeline for the loop. Part C shows the loop after the compiler has applied prolog overlap. In this scenario, the prolog stage for outer loop iteration K+1 is placed between the steady-state loop and the epilog loop for outer loop iteration K. The prolog loop is also duplicated and placed just before the outer loop, in order to cover the first outer loop iteration. Prolog overlap does require an enclosing outer loop, which may or may not be available.

This transformation has the potential to provide two benefits. First, it can improve the scheduling of the prefetches issued in the prolog stage of the prefetch pipeline. Since the prolog loop contains no other computation, the prefetches issued at this point can't be overlapped with other useful computation (floating point calculations, for example) prior to the point where the data is actually needed, in the first few iterations of the steady state. By placing the prolog between the steady state and cleanup stages of a previous pipeline, we increase the opportunities for latency hiding.

A) Original loop nest

```
do j = 1, m
   ... a(1,j) ...
   do i = 1, 1024
      ... b(i,j) ...
   enddo
enddo
```

B) Standard pipeline

```
do j = 1, m
   prefetch a(1,j+1)
   ... a(1,j) ...
   do i = 1, 8, 2
      prefetch b(i,j)
   enddo
   do i = 1, 1016, 2
      prefetch b(i+8,j)
      ... b(i,j) ...
      ... b(i+1,j) ...
   enddo
   do i = 1017, 1024
      ... b(i,j) ...
   enddo
enddo
```

C) Prolog overlap

```
do i = 1, 8, 2
   prefetch b(i,1)
enddo
do j = 1, m
   ... a(1,j) ...
   do i = 1, 1016, 2
      prefetch b(i+8,j)
      ... b(i,j) ...
      ... b(i+1,j) ...
   enddo
   prefetch a(1,j+1)
   do i = 1, 8, 2
      prefetch b(i,j+1)
   enddo
   do i = 1017, 1024
      ... b(i,j) ...
   enddo
enddo
```

Figure 3.12 Prolog overlap

Second, prolog overlap can be used to schedule prefetches for references that are not enclosed in an inner loop. The main problem for such references is the selection of a prefetching distance. If a prefetching distance of 0 is used (*i.e.*, prefetches are issued for the current outer loop iteration's data), then there will be very little time for the prefetch to complete before the data is needed. If the reference is prefetched one outer loop iteration in advance, then the data is likely to arrive too early, since the entire inner loop must execute before it can be used, as in Part B of Figure 3.12. When prolog overlap is in effect, we can use a prefetching distance of 1 iteration, and place the prefetch within the shifted prolog, as in part C.

Prolog overlap is not always safe to apply, since it may not always be possible for the compiler to generate code that predicts the values needed by the next iteration of an outer loop, due to control flow, complex expressions, etc. Given an outer loop L_{outer} and a series of inner loops $L^1_{inner}, L^2_{inner}, \ldots L^k_{inner}$ directly contained in L_{outer}, we enforce the following conditions when applying prolog overlap:

1. L_{outer} must not contain any procedure calls or I/O

2. the step for the outer loop L_{outer} must be known at compile time

3. all inner loops contained in L_{outer} must be immediately *control dependent* only on L_{outer} [29]

4. the loop bounds and subscript functions within each inner loop L^j_{inner} must be composed of linear functions of loop induction variables and values that are loop-invariant with respect to L_{outer})

If the nest meets these criteria, then we duplicate the prolog loop for L^1_{inner} and place it prior to L_{outer}, and we place the prolog loop constructed for L^j_{inner} between the steady state and cleanup loops of loop L^{j-1}_{inner} (the prolog loop for L^1_{inner} is placed between the steady state and cleanup stages of L^j_{inner}). We examine the effectiveness of this transformation in Section 5.6.4 (page 75).

3.4.5 Outer loop pipelining

The efficiency of the pipelining transformation used during prefetching is highly dependent on the number of iterations in the target loop. If a loop executes for only a few iterations, then the pipeline will never "fill" (the steady state stage will never execute, only the prolog and cleanup stages), resulting in poor performance.

To tackle the problem of short inner loops, our compiler implements a form of outer loop pipelining, in which the compiler pipelines a loop that directly encloses an inner loop. loops. Figure 3.13 shows an example of outer loop pipelining. Because the trip count of the inner loop is very small, and because it contains very little computation, the compiler has chosen to pipeline the enclosing "`do j`" loop instead of the inner loop. In addition, the inner loop trip count is small enough that the compiler has decided to completely unroll the inner loop as well.

If the inner loop bounds are unknown at compile-time, applying outer loop pipelining is more difficult. Since the compiler cannot statically determine the inner loop trip count, it cannot compute an accurate value for the outer loop prefetching distance. If the compiler predicts that the loop will incur very long-latency references, however, then outer loop pipelining can still be applied by applying strip-mining beforehand. Figure 3.14 gives an example. Here the inner loop is strip-mined into chunks of `K` iterations, and the strip loop is then moved outside the original two innermost loops. Although the compiler may not be able to determine the value of "`m`", it can still compute a prefetching distance, since it has control over the value of `K`.

Original loop nest

```
do j = 1, n
   do i = 1, 4
      w = w + a(i,j)
   enddo
enddo
```

Outer loop pipelining

```
do j = 1, OPD
   prefetch a(1,j)
enddo
do j = 1, n-OPD
   prefetch a(1,j+OPD)
   w = w + a(1,j)
   w = w + a(2,j)
   w = w + a(3,j)
   w = w + a(4,j)
enddo
do j = n-OPD+1, n
   do i = 1, 4
      w = w + a(i,j)
   enddo
enddo
```

Figure 3.13 Outer loop pipelining

Original loop nest

```
do j = 1, n
  do i = 1, m
    ··· a(i,j) ···
  enddo
enddo
```

Strip-mine inner loop

```
do j = 1, n
  do ii = 1, m, K
    U = min(ii+K-1,m)
    do i = ii, U
      ··· a(i,j) ···
    enddo
  enddo
enddo
```

Interchange, pipeline

```
do ii = 1, m, K
  U = min(ii+K-1,m)
  prolog:
  do j = 1, OPD
    do i = ii, U, 4
      prefetch a(i,j)
    enddo
  enddo
  steady-state:
  do j = 1, n-OPD
    do i = ii, U, 4
      prefetch a(i,j+OPD)
      ··· a(i,j) ···
    enddo
  enddo
  epilog:
  do j = n-OPD+1, n
    do i = ii, U
      ··· a(i,j) ···
    enddo
  enddo
enddo
```

Figure 3.14 Outer loop pipelining with strip-mining

In our compiler we currently only apply this transformation in a restricted set of circumstances. First, the outer loop must contain only a single inner loop (we require perfect nesting), and must not contain procedure calls or I/O. Second, the inner loop bounds and subscript functions must be such that the compiler can predict the values needed several outer loop iterations in advance. Finally, we apply dependence-based tests to insure that loop interchange (blocking) is safe. In Section 5.6.4 (page 77), we give experimental results on the effectiveness of outer loop pipelining.

3.5 Summary

In this chapter, we have described the architecture of our compiler, giving high level-descriptions of the methods we employ to implement prefetching. We first outline how to use shadow arrays to implement prefetches, allowing us to avoid the difficult step of developing an entire optimizing compiler that supports prefetch instructions. We then detailed the stages of analysis and transformation used by our compiler, highlighting the differences between our implementation and that of previous researchers. We give an overview of the new techniques incorporated into our compiler, including enhanced loop peeling for imperfect loop nests, cross-loop reuse analysis, enhanced loop peeling, a register-pressure driven loop unrolling cutoff based on strip-mining, and a pair of optimizations designed to improve prefetch scheduling.

Chapter 4

Evaluating software prefetching

4.1 Introduction

In order to improve the performance of software prefetching, developers need to be able to determine not just whether prefetching fails or succeeds for a given program, but *why* it fails or succeeds. Overall execution time will always be the final arbiter of the success of a given latency-hiding technique, but it is generally not very informative when diagnosing the specific drawbacks of an implementation.

For this reason, we use variety of metrics to provide more information on prefetching performance to the compiler developer. We characterize a prefetching scheme by examining how it performs with respect to three major areas: prefetch *target selection*, prefetch *scheduling*, and prefetch instruction *overhead*. Figure 4.1 shows each area along with the specific metric used evaluate it.

Prefetch target selection is the process of determining the set of references for which prefetches need to be issued. The goal of prefetch target selection is to issue prefetches for the exact set of references that cause cache misses, without prefetching any unneeded data or issuing prefetches for lines already in cache.

Prefetch scheduling efficiency is concerned with how well the compiler is able to place a given prefetch prior to the reference that it is intended to cover. A perfectly scheduled prefetch will cause the desired line to be brought into the cache just prior to its use.

Instruction overhead is defined as the number of additional instructions added to a program as a result of prefetching. Overhead includes the prefetch instructions themselves, supporting instructions needed for address computations, as well as additional instructions that may result from loop transformation overhead and/or spill code due to increased register pressure.

Most of these metrics were originally used by Mowry as part of his dissertation, including instructions per prefetch, prefetch coverage factors, percentage of useless prefetches, and percentage of prefetched lines that are displaced before being used [72].

<table>
<tr><td colspan="2">Area   Sub-area</td><td>Metric</td><td>Comments</td></tr>
<tr><td rowspan="3">Target selection</td><td>Coverage</td><td>% miss reduction</td><td>Fraction of the cache misses that prefetching is able to turn into cache hits.</td></tr>
<tr><td>Overshoot</td><td>% unused pref'd lines</td><td>Fraction of prefetched lines that are evicted from the cache before being used.</td></tr>
<tr><td>Selectivity</td><td>% useless prefetches</td><td>Fraction of prefetches that hit in cache, i.e., don't accomplish useful work.</td></tr>
<tr><td colspan="2">Scheduling</td><td>% late prefetches</td><td>Fraction of prefetches that fail to complete before their corresponding load or store.</td></tr>
<tr><td colspan="2">Overhead</td><td>IPP</td><td>Additional dynamic instructions resulting from issuing prefetches.</td></tr>
</table>

Figure 4.1 Prefetching efficiency metrics

Our contribution is to add refinements that allow the metrics to be more useful for evaluating compiler implementations. Some of these refinements are described in this chapter, and some are presented in Chapter 5.

4.2 Prefetch target selection

We view prefetch target selection as having three components or categories: *coverage*, *overshoot*, and *selectivity*. Coverage measures the success with which prefetching is able to convert cache misses in the base program into cache hits in the transformed program. A given reference R is considered "covered" if the compiler issues a useful prefetch for the location needed by R, turning R from a cache miss into a cache hit. The prefetch *coverage factor* for a program is defined as the percentage reduction in uncovered cache misses achieved by applying prefetching to the program. Overshoot takes place when a prefetch is performed for a line that is subsequently never accessed, *i.e.*, the prefetched line is evicted from the cache before it can be used. Finally, selectivity measures how well the compiler is able to avoid issuing prefetches for lines that are already in cache (useless prefetches).

4.2.1 Coverage

The main metric for prefetch coverage is the percentage reduction in cache misses for the transformed program:

$$Coverage = 100 * \frac{M_{orig} - M_{pref}}{M_{orig}}$$

where M_{orig} and M_{pref} are the number of cache misses in the original and transformed versions of the program, respectively. The ideal coverage factor for a program is 100%. A cache miss whose latency is partially hidden (due to a late prefetch) is considered covered in our framework.

Static vs. dynamic coverage

We attribute poor coverage to two causes: *static* failures, and *dynamic* failures.

Static coverage breakdowns occur when the compiler fails to insert a prefetch instruction for a reference that causes a cache miss. There are host of circumstances that may result in static coverage failures. For example, some prefetching algorithms insert prefetches only for references within inner loops. If a reference is not contained in an inner loop (or not contained in a loop at all), then it may result in an uncovered miss. Limitations in the prefetching algorithm may also prevent the compiler from inserting prefetches even for inner loop references. For example, an inner loop may contain a subroutine call, making it impossible to calculate a prefetching distance at compile-time, or the subscript function for a given reference may be too complex to analyze. Finally, a coverage breakdown may result from imprecision in the compiler's reuse analysis: if the compiler incorrectly decides that references R_1 and R_2 both access the same cache line within a given loop, it will only insert prefetch for one of the references, resulting in an uncovered miss for the other reference.

Dynamic coverage failures occur when a prefetch is issued at run-time, but fails to cause a cache hit for the reference it is intended to cover. Late prefetches are special type of dynamic coverage failure; we treat them as a separate category (see Section 4.4). Cache conflicts are another main cause of dynamic coverage failures: it is often the case that a line is brought into the cache by a prefetch, and then displaced from the cache (due to conflicts) before it can be used.

As part of our experiments, we attempt to separate out the static and dynamic components of prefetch coverage; see Section 5.5.1 for the details. In addition, in Section 5.5.1 we perform a classification of uncovered misses according to lexical nesting, to try to identify the sources of poor coverage.

4.2.2 Overshoot

Overshoot can be measured by counting the number of prefetched lines that are evicted from the cache without being used, and expressing this number as a fraction of all lines brought into the cache by prefetches:

$$Overshoot = 100 * \frac{P_{unused}}{P_{total}}$$

As with prefetch coverage, there are static and dynamic components to overshoot. Static overshoot takes place when the compiler issues prefetches for data that is subsequently never accessed, perhaps due to control flow within a loop. Figure 4.2 shows an example of how control flow can result in prefetching overshoot. In the loop nest shown, the "`if`" statement determines whether the loop accesses the array "`c`" or the arrays "`zz`" and "`yy`". If the compiler issues prefetches for all of the references in the loop, then there is a chance that much of the data will go unused, depending on which way the branch proceeds.

```
do i = 1, n
   d(i) = x(q(i))
   if (a(i) .lt.  1)
      c(i) = 0
   else
      zz(i) = yy(i)
   endif
enddo
```

Figure 4.2 Control flow within an inner loop

Dynamic overshoot takes place when a prefetch is issued and completes, but the line in question is displaced before it can be used, typically due to cache conflicts.

4.2.3 Selectivity

A simple measure of selectivity is the percentage of all prefetches issued that hit in the cache or in the pending prefetch buffer.

$$Selectivity = 100 * \frac{P_{useless}}{P_{total}}$$

The percentage of useless prefetches depends largely on how accurately the compiler is able to identify locality of reference within the program being compiled. A high percentage of useless prefetches for a given program usually indicates that the compiler's reuse analysis is performing poorly for that program.

Useless prefetches can arise in a number of situations and circumstances. In order to isolate the most significant sources of useless prefetches, we develop and apply a more sophisticated set of selectivity metrics in Section 5.5.1.

4.3 Overhead

As mentioned previously, prefetching overhead measures the number of additional instructions that have to be added to the program to issue prefetches. Figure 4.3 illustrates how we measure the instruction overhead of a prefetching implementation. The upper portion of the figure shows the instruction stream of the base program (without prefetching). The lower portion of the figure shows the instruction stream of the transformed program; two prefetch instructions have been inserted, along with 3 other supporting instructions, for a total of 5 additional instruction, yielding an IPP (instructions per prefetch) of 2.5. More formally,

$$IPP = \frac{I_{pref} - I_{orig}}{P_{total}}$$

where I_{orig} and I_{pref} are the dynamic instruction counts for the original and transformed versions of the program, and P_{total} is the total number of prefetches issued in the transformed version of the program.

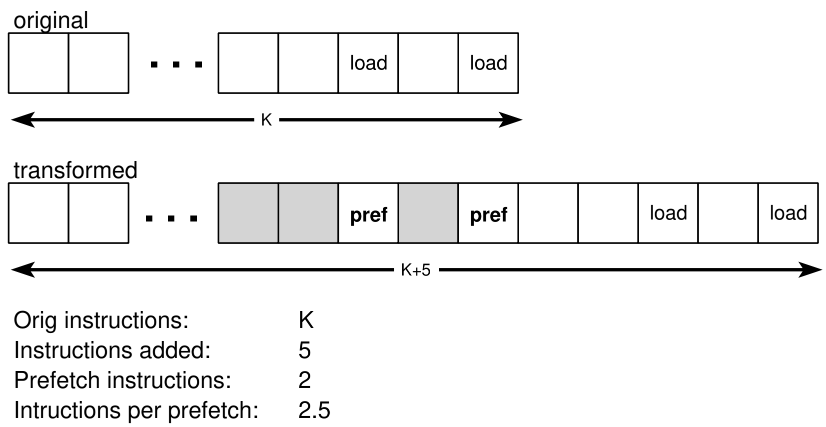


Figure 4.3 Instruction overhead

An ideal IPP value is close to 1, however IPP values are very sensitive to the loop unrolling policy used for both versions of the program. Restricting unrolling in the original version of the program while applying unrolling in the transformed version of the program can result in a low IPP value. Very high unrolling degrees in the transformed version of the program can sometimes lead to high IPP values, due to increased register pressure. High IPP values can also occur when a program has a large fraction of indirect references, which are more difficult to prefetch.

It is important to mention that for a multiple-issue processor capable of overlapping the execution of many instructions simultaneously, the notion of IPP as a measure of instruction overhead becomes considerably more hazy. For example, in a compute-bound loop with a large number of floating point instructions, it may be possible to add prefetches to the loop without any visible increase in execution time, since the processor's integer unit can independently issue and retire the prefetch instructions while other functional units are stalled waiting the floating point operations to complete.

4.4 Scheduling

Scheduling is concerned with the arrival time of a prefetched line in the cache, relative to the load or store that the prefetch is intended to cover. If the prefetch is placed too early, then there is a risk that the prefetched line will be displaced before it can be used. If the prefetch instruction is scheduled too close to the actual reference, then it may be the case that the prefetch is still in progress when the reference executes, resulting in partial memory latency penalties.

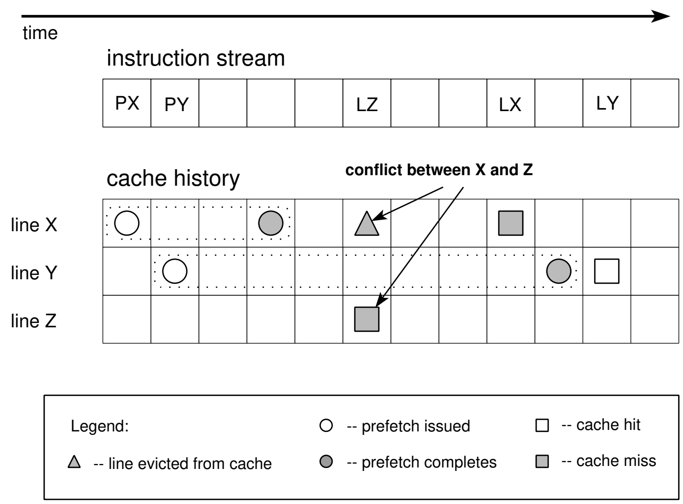


Figure 4.4 Prefetch scheduling

Figure 4.4 illustrates some of the issues of prefetch scheduling. The top portion of the figure shows the program's instruction stream. The program issues prefetches for two lines (`X` and `Y`), performs some other instructions (including a load of line `Z`), then loads locations `X` and `Y`. The bottom portion of the figure shows the activity taking place in the cache. In the scenario depicted, the prefetch for line `X` completes well before its corresponding load. Due to a cache conflict between line `X` and line `Z`, however, the line `X` is displaced from the cache before it can be used, thus the subsequent load of `X` results in a cache miss. If the prefetch for `X` had been issued (and completed) later, then the load of `X` might have resulted in a cache hit instead of a miss. Note that we are assuming that evictions caused by prefetches only occur when the prefetch completes (not at the time it is issued).

Prefetches that are scheduled too late can be easily counted, provided that simulation is being employed:

$$Sched_{late} = \frac{P_{late}}{P_{total}}$$

where P_{late} is the number of times that the processor encounters a cache miss on a line for which there is currently a prefetch operation in progress. Another metric for scheduling efficiency is the average number of cycles that the processor has to stall when waiting for a prefetch to complete.

Prefetches that are scheduled too early are more difficult to measure, since the undesirable effects of an early prefetch are more variable. If a prefetched line X arrives in the cache too early,

1. it may have no ill effects (if the displaced line was no longer being used), or
2. it may be displaced by some other line before it can be used, or
3. it may displace some other line Y that will be used again shortly

The second scenario above will result in an increase in overshoot for the L1 cache (but typically not for the L2 cache), and a decrease in coverage, since the subsequent access to line X will result in a cache miss. The third scenario will result in a decrease in coverage (since the subsequent access to line Y will result in a cache miss), and possibly an increase in overshoot as well (if Y was a prefetched line that had not yet been used).

4.5 Pipeline characteristics

As described in previously, prefetch scheduling is accomplished using a form of pipelining, in which the loop is split into a prolog loop, an unrolled steady-state loop, and an epilog loop. The relative amount of time (iterations) spent in each stage depends on a number of factors, including the trip count of the loop, the amount of computation in the loop body, and the memory latency. Ideally, we would like to see programs spending most of their time executing within the steady-state stage of the prefetch pipeline, for two reasons.

First, the more time the program spends in the steady state, the more it benefits from loop unrolling, since the steady state is unrolled (whereas the epilog is not). If the trip count of the loop is very small, and/or the prefetching distance is quite large, then the epilog stage of the pipeline will perform most of the computation, resulting in increased instruction overhead.

Second, prefetches issued during the steady-state stage are more likely to be scheduled properly. Since the prolog loop contains no other computation, the prefetches it issues will be spaced very close together in time, and will often wind up being buffered due to limited cache and memory bandwidth. Once the prolog is complete, the steady-state loop immediately begins accessing the data prefetched in the prolog; this allows very little time for the prolog prefetches to complete. In contrast, prefetch instructions issued during the steady-state are generally spaced further apart and are overlapped with other useful computation, increasing the likelihood that they will be successful. The end result of these pipeline "start-up" effects is that prefetches issued during the prolog will be less likely to complete on time.

Categories	Definition
`SteadyMajIter`	A majority of the prefetches in the loop were issued during the steady-state stage of the pipeline.
`SteadyMinIter`	A majority of the prefetches in the loop were issued during the prolog stage, but some of the prefetches were issued during the steady-state.
`NoSteady`	All of the prefetches in the loop were issued during the prolog stage; the steady state did not execute.
`WorstCase`	Same as `NoSteady`, but in this case the the trip count of the loop was less than the prefetching distance.

Figure 4.5 Pipeline efficiency categories

The fraction of prefetches issued in each stage of the pipeline is dependent on two factors: the prefetching distance for the loop (which is in turn a function of the number of instructions in the loop body and the memory latency), and the number of iterations in the loop. If T is the number of iterations in the loop and P is the prefetching distance in iterations, then the fractions of prefetches issued in the prolog will be roughly $\frac{\mathtt{min}(P,T)}{T}$. In the case that $P > T$, the steady state portion of the pipeline will not execute at all: only the prolog and cleanup loops will run. In such situations we can expect that prefetch scheduling efficiency will suffer. Instruction overhead will also increase, since the non-unrolled epilog loop has higher instruction overhead than the steady-state loop.

Figure 4.5 shows our classification scheme for the pipeline characteristics of a given execution of a pipelined loop. There are four categories, ranging from `SteadyMajIter` (good pipeline characteristics) to `WorstCase` (very poor pipeline characteristics). By identifying the dominant pipeline characteristics of a given application, we can get a sense for the overall potential of pipelining with respect to scheduling and instruction overhead. In Chapter 5, we use simulation to gather empirical data on the pipeline characteristics of our benchmark applications; this data is presented in Section 5.5.3.

4.6 Summary

This chapter has outlined a set of metrics that are useful for gauging the effectiveness of prefetching for a given compiler, target program, and simulated architecture. We characterize the efficiency of a given prefetching scheme by measuring its performance in three major areas: target selection, scheduling, and instruction overhead. Target selection metrics are designed to indicate whether the compiler is issuing prefetches for precisely the set of references that cause cache misses. They include include coverage (fraction of cache misses whose latency is hidden using prefetching), selectivity (fraction of useless prefetches), and overshoot (fraction of prefetched data that is subsequently unused). Scheduling metrics are designed to measure whether prefetch instructions are being issued at the right time, relative to the accesses they are intended to cover. Prefetches that are scheduled too late can be counted as a measure of scheduling efficiency. In addition, the average number of cycles stalled per late prefetch is a good way to measure the severity of the penalties incurred by late prefetches. Early prefetches are more difficult to detect, and must generally be measured indirectly by looking for decreases in coverage and increases in overshoot.

Most of the metrics described in this chapter were originally developed by Mowry et al. Our contributions are to provide enhancements that allow the metrics to be more useful for evaluating the compiler components of prefetching performance. These enhancements include static and dynamic coverage factors, static and dynamic overshoot, and more detailed scheduling metrics. Additional refinements of these metrics are presented in Chapter 5.

Chapter 5

Uniprocessor Prefetching

In this chapter we explore the area of prefetching for single-processor architectures. In the first half of the chapter, we present a series of experiments designed to determine the overall effectiveness of prefetching for a set of benchmark programs. Our focus for this work is on the compiler; we have structured our experiments to isolate compiler behavior, and to distinguish between situations where the compiler is performing poorly and situations where performance is poor due to application-related or architectural factors. In the second half of the chapter, we experimentally evaluate the new compiler techniques we have developed.

An outline of this chapter is as follows. Sections 5.1, 5.2, and 5.3 describe the benchmark programs that we have chosen for our experiments and give the details of our compilation and simulation methodology. Sections 5.4 gives raw data on the performance improvement provided by prefetching for each of our benchmarks. In Section 5.5 we give a more detailed evaluation of prefetching performance, using the metrics developed in Chapter 4. Finally, Section 5.6 covers the evaluation of the new uniprocessor techniques outlined in Chapter 3.

5.1 Uniprocessor benchmark programs

Figure 5.1 gives some of the important characteristics of our benchmark programs. These programs were selected from the SPEC 95 benchmarks, the NAS benchmarks, the Rice Compiler Evaluation Program Suite, and the PERFECT benchmarks [10, 25, 78, 96]. In order to yield more reasonable simulation times, we reduced the number of timesteps executed for many of the programs, and in some cases we had to reduce the sizes of the arrays used. In Figure 5.1, “Functions” is the number of procedures in the program; “Lines” is the number of non-comment source lines. “Data” is the total size of all the arrays used by the program, in megabytes. “Instructions” is the approximate dynamic instruction count for the program. The L1 and L2 miss rates shown are for the cache configuration specified in Figure 5.2. The miss rates are

Program	Origin	Functions	Lines	Data (MB)	Instructions	L1 miss rate	L2 miss rate
appbt	NAS	20	2734	1.2	55,103,834	13.1%	22.7%
applu	SPEC	16	1883	33.1	69,851,918	15.0%	23.2%
appsp	NAS	26	2120	0.3	45,247,365	10.5%	9.3%
apsi	SPEC	96	4872	9.6	169,836,484	8.9%	32.0%
cgm	NAS	14	381	2.9	77,141,945	14.2%	24.0%
erl	RiCEPS	16	494	8.1	387,659,431	16.8%	26.5%
flo52	PERFECT	36	2235	1.0	89,212,405	16.9%	1.9%
hydro2d	SPEC	40	1565	2.2	164,286,859	24.7%	30.5%
mgrid	SPEC	13	445	1.1	129,600,506	5.7%	26.5%
nasbuk	NAS	7	181	3.2	259,816,923	35.2%	13.6%
ocean	PERFECT	37	2051	0.8	150,291,613	18.3%	4.5%
su2cor	SPEC	36	1753	24.4	561,211,029	25.3%	15.6%
swim	SPEC	6	261	14.7	207,196,621	74.9%	11.5%
tomcatv	SPEC	1	125	14.8	369,712,137	20.8%	21.2%
wave5	SPEC	104	7372	42.8	372,647,825	41.9%	3.8%

Figure 5.1 Uniprocessor benchmark programs and their vital statistics

calculated based on accesses to arrays and global (common) data only; we exclude cache activity due to stack-allocated scalars, the constant pool, spill code, etc.

5.2 Simulator

For these experiments we use an execution-driven simulator derived from RPPT [27]. We simulate a pipelined SPARC processor with functional unit latencies that are equivalent to those of the Sun `UltraSparc` [92]. The simulator models the cache and memory subsystems in detail. Data collected by the simulator include overall execution time, cache miss rates, average miss latency, total prefetches issued, useless prefetches, and late prefetches, among others. Figure 5.2 shows the parameters we use for our uniprocessor simulations. Since we are using reduced data set sizes for many of our benchmark programs (to yield shorter simulation times), we selected L1 and L2 cache sizes that are correspondingly reduced as well.

We use a blocking-load model for the processor. Both levels of cache are lockup-free, however, in the sense that they can support multiple outstanding prefetch and/or write requests, in concert with a single pending cache miss. Multiple requests for a given line are coalesced, thus there can only be one outstanding type of request for a given line at a time. Inclusion between the L1 and L2 cache is enforced. When the processor tries to issue a prefetch but encounters a full prefetch buffer, it stalls until a slot is available in the buffer, as opposed to dropping the prefetch.

Processor parameters	
Clock speed	300 MHz
Cache parameters	
Cache line size	32 bytes
L1 cache	8 Kbytes, 2-way set-associative, write-through
L1 hit time	1 cycle
L1 write buffer	16 entries
L2 cache	128 Kbytes, 2-way set-associative, write-back
L2 hit time	10 cycles
Pending prefetch buffer	16 entries
Memory parameters	
Main bus	100 Mhz, 256 bits wide
Peak bus bandwidth	3.2 Gbytes/sec
Memory speed	200ns / 32 bytes
Memory interleaving	4-way by 32-byte chunks
Peak memory bandwidth	640 Mbytes/sec
Miss-to-memory	83 cycles

Figure 5.2 Architectural parameters for uniprocessor simulations

5.3 Compiler parameters

We used a prefetch latency of 175 cycles for these experiments. This value was arrived at empirically, by trying a wide range of latencies and selecting the value that produced the best improvement for all of the programs. The programs were compiled using the compilation strategy described in Chapter 3, including selective prefetching, loop peeling, and multi-stage pipelining for indirect references, but without any of the new techniques discussed in Section 3.4. Only references enclosed in inner loops are considered for prefetching. We refer to this as the "default" compilation strategy. Since the target architecture features a write-through L1 cache, our compiler does not insert prefetch instructions for writes (stores).

Our compiler operates in a source-to-source fashion, reading and writing Fortran; both the original Fortran program and the transformed Fortran program are then run through the Sun `f77` compiler (version SC4.0) to produce a SPARC executable.

The compiler flags given to `f77` were "`-dalign -O3 -cg92`" for the original program and "`-dalign -O3 -cg92 -unroll=1`" for the transformed program.

5.4 Execution time

Figure 5.3 shows the overall improvement in execution time provided by software prefetching for our benchmark programs. There is a wide variation in the effectiveness of prefetching, ranging from 43.4% improvement (`hydro2d`) to -6.2% (`flo52`). The programs that derived the most benefit from prefetching were generally those with high L1 and/or L2 miss rates (as shown in Figure 5.1), with some exceptions. In the next few sections we will attempt to explore these results in more detail, and to determine the reasons why prefetching behaves the way it does for these programs. Section 5.5.1 provides empirical data on prefetch target selection; Section 5.5.2 presents data on instruction overhead, and Section 5.5.3 presents data on prefetch scheduling.

Program	Reduction in execution time
appbt	8.3%
applu	8.6%
appsp	2.4%
apsi	-5.1%
cgm	27.0%
erl	14.8%
flo52	-6.2%
hydro2d	43.4%
mgrid	20.2%
nasbuk	22.2%
ocean	5.6%
su2cor	21.9%
swim	17.0%
tomcatv	38.5%
wave5	-0.2%

Figure 5.3 Execution time reduction due to prefetching, 1 processor

5.5 Experimental data on prefetching performance

In this section we use the metrics described in Chapter 4 to look at various aspects of prefetching performance. We first examine prefetch target selection (Section 5.5.1), then consider instruction overhead (Section 5.5.2) and finally prefetch scheduling (Section 5.5.3).

5.5.1 Prefetch target selection

Figure 5.4 contains data on prefetch coverage for our benchmark programs. We report coverage factors for both levels of cache. To simplify our measurements of target selection, we modified our simulator (for these experiments only) to treat prefetches as zero-latency operations: fetched data is brought into the cache on the next cycle. We use a prefetching distance of a single loop iteration for all pipelined loops. In combination, these modifications remove imprecision due to poor prefetch scheduling (i.e. very early or very late prefetches), and allow us to focus solely on the issue of coverage.

	Coverage factors			
	L1 cache		L2 cache	
Program	Base	Virtual	Base	Virtual
`appbt`	58.0%	66.7%	61.7%	61.6%
`applu`	91.0%	94.0%	96.2%	96.4%
`appsp`	84.4%	89.1%	82.4%	82.8%
`apsi`	41.8%	90.5%	42.1%	90.6%
`cgm`	84.1%	84.5%	66.9%	67.0%
`erl`	89.6%	90.3%	91.3%	91.3%
`flo52`	76.0%	79.3%	87.7%	84.0%
`hydro2d`	92.6%	94.5%	94.5%	94.5%
`mgrid`	73.8%	88.0%	93.0%	93.1%
`nasbuk`	98.4%	100.%	99.9%	100.0%
`ocean`	61.7%	63.0%	86.9%	86.6%
`su2cor`	70.1%	96.2%	99.1%	99.4%
`swim`	4.1%	72.2%	69.0%	84.1%
`tomcatv`	59.0%	99.7%	99.6%	99.7%
`wave5`	26.9%	54.6%	88.8%	88.9%

Figure 5.4 Prefetch coverage factors

Virtual coverage

Prefetch coverage factors provide only limited information about compiler behavior, since it is difficult to determine whether poor coverage is due to static or dynamic coverage breakdown (see Section 4.2.1 for a definition of these terms). To deal with this problem, we introduce the notion of the "virtual" coverage factor, also shown in Figure 5.4.

To calculate virtual coverage, we add a 64-entry *victim cache* to our hypothetical simulated machine. This cache is a separate, fully-associative cache used to store prefetched lines that have been recently evicted from the main cache [50]. During the simulation, when a line is displaced from a given cache, we insert it into the victim cache for that level. When the program incurs a cache miss, the simulator checks for the line in the victim cache; if the line is present, then the miss is considered a cache hit for the purposes of computing the virtual coverage factor. By comparing the virtual coverage factor with the base coverage factor, we can get a sense for both the static and dynamic components of prefetch coverage.

The results in Figure 5.4 show that for the first level of the memory hierarchy, cache conflicts have at least some effect on most of the programs. For 3 of the programs, "`apsi`", "`swim`", and "`tomcatv`", cache conflicts cause significant reductions in prefetch coverage. In these cases the difference between the base and virtual coverage factors is more than 40 percentage points.

Although some of the programs have virtual L1 coverage factors approaching 100%, there are a few programs for which L1 and L2 coverage factors are roughly 50-70%. For these programs, it seems logical to conclude that the compiler is failing to insert prefetches for a significant numbers of references (see Section 4.2.1 for a description of the situations where our compiler will disable prefetching for an inner loop).

To try to learn more about the coverage failures exhibited in Figure 5.4, we designed an experiment to learn more about the locations of uncovered cache misses, and to see what factors are responsible for the low coverage rates.

Lexical nesting of uncovered misses

In this experiment, we instrumented our benchmark programs with calls that mark the beginning and end of each function and loop. The simulator then uses a stack to keep track of dynamic loop nesting at runtime. The simulator's book-keeping information

Ref category	Definition
`NoLoop`	Reference is not enclosed in any loop.
`IPLoop`	Reference is not enclosed in any loop within the current procedure, but is dynamically enclosed in a loop in some calling routine.
`MidLoop`	Reference is enclosed in a loop, but not an inner loop.
`InnerLoop`	Reference is enclosed in an inner loop.

Figure 5.5 Reference categories by lexical/interprocedural loop nesting

is used to place each reference into one of four classes, shown in Figure 5.5. Each L1 cache miss is then assigned to a category based on the class of the reference that incurred the miss.

Program	No loop	IP loop	MidLoop	Inner loop
appbt	-	-	43.9%	56.0%
applu	-	-	0.1%	99.8%
appsp	-	-	27.0%	72.9%
apsi	-	0.4%	10.9%	88.5%
cgm	-	-	3.2%	96.6%
erl	-	-	-	99.9%
flo52	2.8%	-	4.0%	93.0%
hydro2d	-	-	0.1%	99.8%
mgrid	-	-	-	99.9%
nasbuk	-	-	-	100.0%
ocean	-	-	0.5%	99.3%
su2cor	-	0.5%	0.5%	98.9%
swim	-	-	-	99.9%
tomcatv	-	-	0.1%	99.8%
wave5	0.5%	-	-	99.4%

Figure 5.6 Loop nesting breakdown for uncovered cache misses

Figure 5.6 shows the percentage of each category of uncovered cache miss for each of the programs. Since we are primarily interested in the compiler's contribution to prefetch coverage, we use *virtual* cache misses for this experiment. The data indicate that in general, the majority of uncovered cache misses in fact take place within the inner loops of these programs. Four of the programs exhibit non-trivial fractions of `MidLoop` misses; for these programs, our default policy of applying prefetching only to references in inner loops is falling short. For the remaining programs, however, we find that reduced coverage is not due to unprefetched outer loop misses, however. In addition, uncovered misses in the `NoLoop` and `IPLoop` categories are almost nonexistent. This suggests that efforts to improve prefetch coverage should continue to focus on loops, and that there is little benefit to be gained from prefetching references not enclosed in loops.

Selectivity

Figure 5.7 shows the percentage of useless prefetches exhibited for the programs in our benchmark suite. The results show that with the exception of a few programs (`erl`, `swim`, and `tomcatv`), the total percentage of useless prefetches is above 50%. This suggests that the compiler's locality analysis is not entirely successful in predicting and/or exploiting locality of reference in these programs. We report the percentage of prefetches that hit in the L2 cache, but it should be noted that a high fraction of useless prefetches at this level is not generally an indication of a problem with the compiler analysis, since prefetches that hit in the L2 cache still provide benefit to the program.

Classification of useless prefetches

In order to pinpoint the places where the compiler analysis is falling short, we performed the following experiment. We conceptually we divide the program's execution into a sequence of intervals or *epochs*, where each epoch corresponds to a loop or loop nest. In our model, there are two types of epochs, *inner loop* epochs and *outer loop* epochs. A new inner loop epoch begins each time an inner loop starts to execute, and a new outer loop epoch begins each time a lexically outermost loop starts to execute. Figure 5.8 gives an illustration.

The compiler makes epoch transitions visible to the simulator by instrumenting the benchmark programs, inserting calls to simulator runtime routines at the beginning and end of each loop. The simulator maintains counters that record the current inner and outer epoch number, incrementing them at the appropriate points. Finally, the simulator associates epoch information with each line in the cache, by storing some state information with each cache line. Each line is tagged with four values, shown

Program	% useless prefetches L1 cache	L2 cache
appbt	68.4%	37.5%
applu	67.2%	34.3%
appsp	50.5%	77.1%
apsi	71.5%	46.8%
cgm	55.1%	65.1%
erl	19.6%	14.9%
flo52	71.1%	95.6%
hydro2d	27.2%	11.7%
mgrid	26.0%	48.6%
nasbuk	34.9%	77.6%
ocean	85.0%	82.9%
su2cor	58.3%	49.0%
swim	5.2%	30.4%
tomcatv	1.5%	35.3%
wave5	60.6%	81.9%

Figure 5.7 Useless prefetches as a total of all prefetches

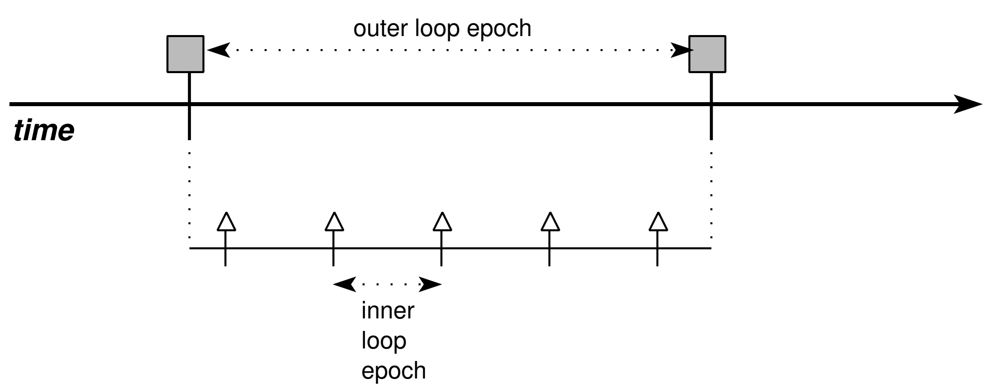


Figure 5.8 Inner and outer loop epochs

in 5.9. These values record the epoch numbers during which the line was *A)* brought into the cache, and *B)* most recently used, for each type of epoch.

Counter	Interpretation
`InnerEpochIn`	inner epoch number during which the line was brought into the cache
`InnerEpochUse`	most recent inner epoch number during which the line was used (set on every access to line)
`OuterEpochIn`	outer epoch number during which the line was brought into the cache
`OuterEpochUse`	most recent outer epoch number during which the line was used (set on every access to line)

Figure 5.9 Epoch values stored with each cache line

When a useless prefetch was issued for a given line, the line's epoch values are then used to classify the prefetches into one of several categories, shown in Figure 5.10. A "Cross-Loop Useless" (`CLU`) prefetch corresponds to a prefetch issued for a line that was brought into the cache in a prior outer loop epoch, and not referenced since then; this type of prefetch is only possible to eliminate using cross-loop reuse analysis. A "Mid-Loop Useless" (`MLU`) prefetch corresponds to a line that was brought into the cache in the current outer epoch, but not used or prefetched during the current inner epoch. This type of useless prefetch may be detected with intra-loop analysis, and can often be eliminated by loop peeling. An "Inner Loop Useless" (`ILU`) prefetch corresponds to a line that was brought into the cache in the current outer epoch, and was prefetched and/or used during the current inner epoch. Intra-loop reuse may be able to detect and directly eliminate this type of useless prefetch.

Figure 5.11 shows the fractions of all useless prefetches that fall into these three categories for our benchmark suite. For the programs that make heavy use of index arrays (`cgm`, `nasbuk`, `su2cor`, and `wave5`), we note that the bulk of the useless prefetches fall into the `ILU` category. We attribute this to the fact that since the compiler can't determine the contents of an index array, the compiler cannot predict reuse among indirect accesses, and therefore can't eliminate useless prefetches that

Type	Definition	Conditions
Cross Loop Useless (CLR)	Prefetch hits a line L that arrived in the cache during a previous outer epoch, but has not been used or prefetched since then.	L.OuterEpochIn $<$ curOuterEpoch and L.OuterEpochUse $<$ curOuterEpoch
Mid Loop Useless (MLU)	Prefetch hits a line L, where L has been used during the current outer epoch, but has not been used within the current inner epoch.	L.OuterEpochUse $=$ curOuterEpoch and L.InnerEpochUse $<$ curInnerEpoch
Inner Loop Useless (ILU)	Prefetch hits a line L, where L has been used during the current outer epoch and during the current inner epoch.	L.OuterEpochUse $=$ curOuterEpoch and L.InnerEpochUse $=$ curInnerEpoch

Figure 5.10 Categories of useless prefetches

Program	CLU %	MLU %	ILU %
appbt	1.5%	65.3%	33.0%
applu	2.2%	68.4%	29.3%
appsp	2.1%	78.1%	19.6%
apsi	49.3%	15.6%	34.9%
cgm	-	37.8%	62.1%
erl	-	92.6%	7.3%
flo52	9.7%	29.6%	60.6%
hydro2d	0.2%	64.4%	35.3%
mgrid	0.9%	82.9%	16.1%
nasbuk	0%	0.1%	99.8%
ocean	0.4%	13.8%	85.7%
su2cor	1.9%	3.4%	94.6%
swim	0.1%	91.8%	8.0%
tomcatv	-	87.1%	12.8%
wave5	1.7%	8.2%	89.9%

Figure 5.11 Useless prefetch breakdown

are caused by the reuse. The program `ocean` does not use indirection arrays, but contains many references with symbolic strides, effectively hiding all spatial locality from the compiler. For many of the remaining programs that have high rates of useless prefetches (`appbt`, `applu`, `appsp`, `erl`, `hydro2d`, and `mgrid`), the bulk of the useless prefetches fall in the `MLU` category, which suggests that the compiler is successfully recognizing and exploiting inner loop locality, but is failing to properly detect and exploit outer loop locality. In the case of the program `apsi`, almost half the useless prefetches fall into the `CLU` category, which cannot be detected by using loop level reuse analysis.

In general, these results suggest that there is some room for improvement with respect to the reuse analysis employed by the compiler. As part of Section 5.6.1, we evaluate the extent to which more advanced forms of reuse analysis can be used to reduce the frequency of useless prefetches.

	Overshoot	
Program	Base	Virtual
appbt	6.7%	5.8%
applu	5.0%	4.4%
appsp	4.8%	3.8%
apsi	44.8%	0.6%
cgm	0.7%	0%
erl	0%	0%
flo52	13.4%	11.3%
hydro2d	1.3%	0.2%
mgrid	0.8%	0.1%
nasbuk	0.6%	0%
ocean	1.5%	0.8%
su2cor	27.5%	3.6%
swim	88.8%	4.0%
tomcatv	20.7%	0%
wave5	54.7%	5.9%

Figure 5.12 Overshoot and Virtual Overshoot

Overshoot

Figure 5.12 provides data on overshoot for our benchmarks programs. The values in this table show the percentage of all lines brought into the cache by prefetch instructions that are subsequently evicted before being used. All values are with respect to the L1 cache. Figure 5.12 also shows the "virtual" overshoot percentage, measured using the addition of a victim cache to our simulated architecture (as described in Section 5.5.1). For virtual overshoot we count the number of unused lines evicted from the victim cache, not the main L1 cache.

The data indicate that most overshoot is due to cache conflicts– the programs with high degrees of overshoot are exactly those that suffer from conflicts (`apsi`, `su2cor`, `swim`, `tomcatv`, and `wave5`). The same programs also exhibit elevated rates of virtual overshoot; we attribute this to cache conflicts that take place over longer periods of time, which the victim cache is unable to ameliorate. Based on a source code inspection, we find that inner loop control flow does not appear to be a factor influencing the level of overshoot. For example, the program with the highest level of overshoot, "`flo52`", has no control flow in the critical loop nests that generate nearly all of the prefetches. Our conclusion is that there is little to be gained from the development of optimizations that attempt to reduce static overshoot from control flow, as discussed in Section 4.2.2.

5.5.2 Instruction overhead

Figure 5.13 provides data on prefetching instruction overhead for our benchmarks, in the form of IPP values (as defined in Section 4.3). For most of the programs, less than 10 instructions are required per prefetch. Although comparisons with previous studies are problematic, due to differences in the target instruction set, compiler, etc., the IPP values shown are roughly in line with those produced by previous software prefetching implementations [72].

In general, we find that programs that use indirection arrays have slightly higher IPP values, which we attribute to the additional overhead of multistage pipelining. Several of the programs in our benchmark suite have fairly high levels of instruction overhead, in spite of the fact that they don't use indirection arrays. In particular, `apppt`, `applu`, and `appsp`, have IPP values above 5. We attribute this to the fact that these codes have very short inner loops, which means that they spend very little time in the "steady-state" stage of their pipelines, gaining very little from unrolling. Section 5.5.3 examines this phenomenon in more depth.

Two of the programs, `erl` and `flo52`, have elevated levels of IPP due to an artifact of the compilation process related to loop unrolling. When we run the final transformed Fortran program through the Sun `f77` compiler, we disable `f77`'s loop unrolling feature, since unrolling has already been applied as part of prefetching. If our compiler has decided that no prefetching is required for a given loop, however, then it will not unroll the loop at the Fortran level, whereas the corresponding loop in the untransformed version of the program may well be unrolled by `f77`. This effect can be greatly magnified for loops containing calls to Fortran intrinsic functions, due to a quirk in the current `f77` optimizer, resulting in artificially large IPP values.

Program	% increase in instructions	**IPP**	IPP_R
appbt	25.8%	6.36	5.93
applu	34.4%	7.59	7.22
appsp	34.7%	5.58	4.85
apsi	43.8%	6.97	5.35
cgm	131.3%	6.73	4.65
erl	1.8%	11.05	-8.99
flo52	41.7%	7.39	4.18
hydro2d	5.9%	1.57	-0.87
mgrid	-0.1%	-0.13	-3.49
nasbuk	6.0%	2.59	-0.94
ocean	27.9%	2.46	0.22
su2cor	16.0%	2.34	1.73
swim	-1.9%	-1.59	-9.22
tomcatv	-2.1%	-3.74	-4.86
wave5	29.5%	3.99	2.15

Figure 5.13 Instruction overhead from prefetching

It should be noted that we use a slightly different loop unrolling policy than that used in previous studies. In Mowry's work, the untransformed versions of the programs are compiled without loop unrolling, while the prefetched versions of the program receive the benefits of unrolling as a side effect of the transformations used to implement prefetching. Since loop unrolling often has beneficial effects for programs even without prefetching, we allow loop unrolling when compiling the untransformed versions of the benchmarks. We accomplish this by directing the `f77` compiler to apply loop unrolling, as opposed to unrolling loops at the Fortran level. This policy results in IPP values that are higher than with a more restrictive loop unrolling policy. By way of comparison, the column labeled "IPP_R" shows the IPP values when no unrolling is applied during compilation of the untransformed versions of the programs. The data show that suppressing unrolling can yield excellent IPP values for the transformed codes (-9 for `erl` and for `swim`, for example).

Effects of cache organization on overhead

The default cache line size used for our experiments is 32 bytes; in this section we examine the impact of using a larger line size on prefetching overhead. The main change that a larger line size causes in the compilation strategy is to increase the degree of loop unrolling needed in order to exploit spatial locality. If a loop has to be unrolled by a factor of 4 when using 32-byte lines, the same loop must be unrolled by a factor of 16 when targeting a machine with 128-byte cache lines. To examine the effects of line size on instruction overhead, we performed an experimental study. Figure 5.14 shows the IPP values for each of the programs as the line size is increased from 32 to 128 bytes.

	IPP		
Program	LS=32	LS=64	LS=128
appbt	6.36	9.92	9.45
applu	7.59	8.85	9.06
appsp	5.58	6.29	6.23
apsi	6.97	7.21	8.63
cgm	6.73	5.69	6.53
erl	11.05	16.76	64.13
flo52	7.39	6.47	8.48
hydro2d	1.57	-0.43	-4.12
mgrid	-0.13	-2.95	-13.40
nasbuk	2.59	1.92	1.70
ocean	2.46	2.27	2.62
su2cor	2.34	2.01	1.86
swim	-1.59	-6.01	-11.55
tomcatv	-3.74	-9.16	-21.35
wave5	3.99	3.64	3.43

Figure 5.14 Effect of cache line size on IPP

Our interpretation of these results is as follows. First, based on an inspection of the transformed programs, we found that excess register pressure (due to loop unrolling) did not play a key role in determining IPP; the register allocator employed by the Sun SC4.0 `f77` compiler is quite robust and seems to be able to handle the expanded loop bodies produced by large unrolling degrees.

Instead, we found that programs with good pipeline characteristics, (*i.e.*, those that spend the bulk of their time in the steady-state portion of the prefetch pipeline) tend to benefit from larger line sizes, whereas programs with poor pipeline characteristics tend to show an increase in prefetching overhead. We look at the pipeline characteristics of the various benchmark programs in a subsequent portion of this chapter (Section 5.5.3); we note that there is a strong correlation between pipeline characteristics and IPP benefits from increased unrolling. The program "`erl`" exhibits artificially high IPP levels in spite of modest overhead in the 32-byte line case (1 to 4 percent). This is due to an artifact of our loop unrolling policy, as described in Section 5.5.2,

It should be noted that compile times increase fairly drastically when larger unrolling degrees are used; in Section 5.6.3 we provide empirical data on compile times and executable sizes, and show how strip-mining can be employed to produce more moderate compile times while still retaining most of the benefits of unrolling.

5.5.3 Experimental data on prefetch scheduling

We now consider the issue of prefetch scheduling. We begin with an empirical study that examines the loop characteristics of our benchmark programs with respect to pipelining, to try to determine how effective inner loop pipelining will be. We then provide empirical data on the frequency of late prefetches for our benchmark suite.

Pipeline efficiency

As previously described in Section 4.5, the pipeline characteristics of a given inner loop can be characterized according to the fraction of time spent in the steady state stage, as opposed to the prolog and epilog stages. In this section we present the results of an experimental study in which we determined the dominant pipeline characteristics of the loops in each of our benchmark programs.

The experiment was structured as follows. Our compiler instrumented the inner loops in each program to record the number of iterations executed. We then combined the data on the average inner loop trip count with the number of instructions in each inner loop body, effectively computing off-line the fraction of time spent in each pipeline stage for each inner loop. Based on this information, we placed each inner loop into one of four categories, shown in Figure 4.5. Each of these categories is intended to represent a particular level of scheduling efficiency, ranging from excellent (`SteadyMajIter`) to poor (`WorstCase`). We then computed a pipeline category distribution for each program by weighting each loop by the fraction of the total prefetches issued in each loop, and summing the weighted values for each category to produce a total.

Figure 5.15 shows the result of this study, conducted using a prefetch latency of 175 cycles. The data show that there is quite a wide variation in the pipeline characteristics of the various applications. Programs such as `tomcatv` and `swim` have almost perfect pipeline characteristics, whereas for `appbt` or `applu`, most of the prefetches in the program are issued in loops with worst-case pipeline characteristics (*i.e.*, loops with very short trip counts), greatly reducing the opportunities for overlapping prefetches with other computation. At the very least, these data suggest that there are significant opportunities for improvement for about half the programs in our suite. We present a pair of optimizations that address these opportunities in Section 5.6.4.

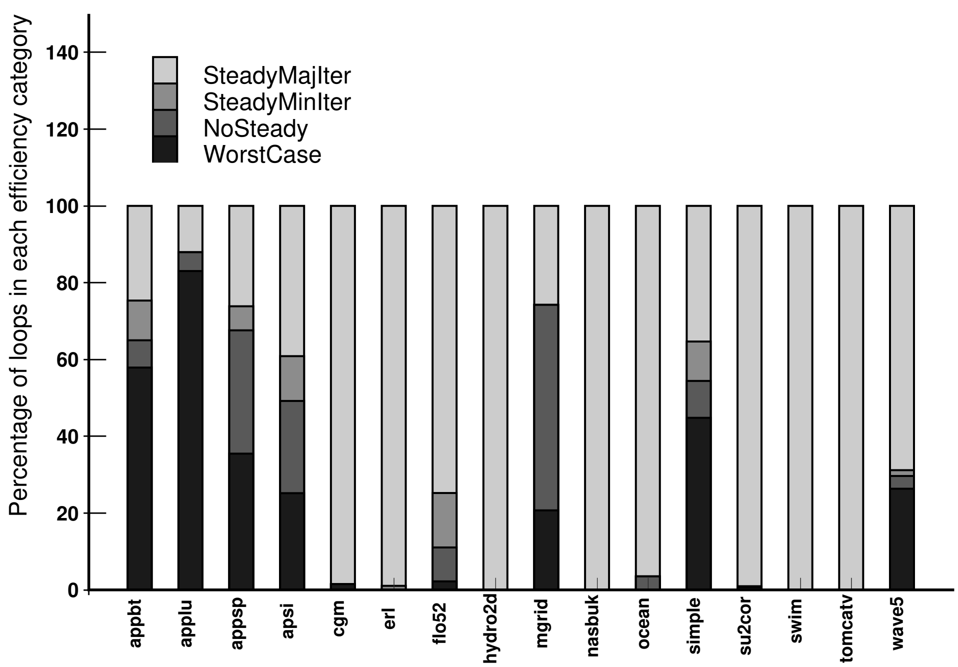


Figure 5.15 Pipeline efficiency metrics

Empirical data on late prefetches

Program	Percent late prefetches	Average number of cycles late
appbt	8.6%	41
applu	9.7%	39
appsp	1.2%	17
apsi	2.6%	36
cgm	1.2%	15
erl	7.2%	32
flo52	0.2%	22
hydro2d	3.9%	66
mgrid	1.3%	32
nasbuk	--	--
ocean	7.7%	7
su2cor	0.1%	17
swim	0.5%	69
tomcatv	0.6%	35
wave5	0.2%	39

Figure 5.16 Late prefetches as a percentage of useful prefetches

Figure 5.16 shows the fraction of non-useless prefetches that failed to complete on time, for all of our benchmark programs. As expected, the programs that exhibit large fractions of late prefetches are those that have poor pipeline characteristics, as we saw previously in Figure 5.15. Figure 5.16 also shows the average number of cycles the processor had to stall to wait for fetched data to arrive in cache. The average stall is around 30 cycles. In combination with the low level of late prefetches, this suggests that the payoff provided by scheduling optimizations is likely to be fairly small, for the given architectural parameters. In a subsequent portion of this chapter (Section 5.6.4) we give the results of a study in which we measure the effects of a pair of new optimizations that attempt to reduce the frequency of late prefetches.

5.6 New uniprocessor compiler techniques

In this section we evaluate the new methods we have developed to improve uniprocessor prefetching. We present new methods related to prefetch target selection, instruction overhead, and prefetch scheduling.

5.6.1 Techniques for improving prefetch target selection

The data from our experiments on prefetch target selection in Section 5.5.1 show that low levels of prefetch coverage are due primarily to cache conflicts (which are difficult to attack using compiler techniques), and that most of our programs exhibit very little overshoot. As a result, we chose to focus primarily on techniques to reduce the number of useless prefetches. We evaluate a series of more powerful reuse analysis methods to determine how effective they are in reducing useless prefetches. In addition, we examine the question of whether it is profitable to apply group-spatial reuse analysis to references that use indirection arrays.

Method	Description
`IntraLoop-Restricted`	intra-loop analysis only; compiler looks for group-temporal reuse between identically nested references
`IntraLoop-Relaxed`	intra-loop analysis only; compiler detects and exploits group-temporal reuse between non-identically nested references
`CrossLoop`	compiler employs cross-loop reuse analysis in addition to relaxed intra-loop reuse analysis

Figure 5.17 Spectrum of reuse analysis techniques

Improved reuse analysis

In this section we evaluate a series of reuse analysis techniques that are progressively more powerful than those employed in previous frameworks for software prefetching. Figure 5.17 shows the three schemes used, in order of increasing generality. We start with the "IntraLoop-Restricted" scheme, in which the compiler employs only intra-loop analysis, and is restricted to considering reuse only among identically nested references. The second scheme is "IntraLoop-Relaxed", in which the compiler also considers non-identically nested references. The third scheme is "Cross-Loop", in which the compiler performs cross-loop reuse analysis (described in Chapter 6) in addition to intra-loop techniques.

Program	Intra-loop Restricted Cov.	Intra-loop Restricted Usel.	Intra-loop Relaxed Cov.	Intra-loop Relaxed Usel.	Cross-loop Cov.	Cross-loop Usel.
appbt	0%	8.1%	-0.8%	11.4%	-0.2%	14.8%
applu	-0.1%	13.8%	-0.1%	14.4%	-0.1%	14.9%
appsp	0.1%	7.1%	0.9%	11.6%	1.7%	16.8%
apsi	-3.4%	0.4%	-0.2%	0.9%	-2.6%	-0.6%
cgm	0.1%	0.2%	NC	NC	-0.1%	0.1%
erl	2.4%	42.0%	NC	NC	NC	NC
flo52	-0.3%	2.3%	-0.6%	2.7%	-0.5%	2.4%
hydro2d	0%	3.6%	-1.1%	6.9%	NC	NC
mgrid	0.5%	33.0%	NC	NC	NC	NC
nasbuk	0%	0%	NC	NC	NC	NC
ocean	0.1%	0.1%	NC	NC	NC	NC
su2cor	0%	0.1%	-0.4%	0.1%	-1.2%	0.2%
swim	0%	1.2%	-3.9%	4.8%	NC	NC
tomcatv	0%	0%	NC	NC	NC	NC
wave5	-2.3%	-0.4%	-5.4%	1.3%	-6.4%	0.7%

Figure 5.18 Effects of more general reuse analysis methods

Figure 5.18 shows how each of these techniques compare. For each technique, we measure the percentage of useless prefetches eliminated relative to a baseline compilation strategy in which loop peeling is disabled. In addition to displaying the reduction in useless prefetches, we also show the change in L1 virtual coverage. This is intended to identify situations where the reuse analysis actually eliminating useful prefetches as opposed to useless prefetches. In the table, "`NC`" indicates that the method in question did not produce any change in the particular metric relative to the method shown in the previous column. All values are with respect to the baseline, not with respect to the previous method.

We interpret the data as follows. The simplest form of reuse analysis, when combined with loop peeling, provides the biggest payoff in terms of eliminating useless prefetches. This is particularly the case for programs such as "`mgrid`" and "`erl`", in which there are essentially no imperfect loop nests. Relaxed intra-loop reuse provides benefits only for programs that make heavy use of imperfect loop nests, primarily "`appbt`", "`applu`", and "`appsp`". For these programs, it provides a 2-4% reduction in useless prefetches above the "IntraLoop-Restricted" scheme. Cross-loop reuse provides another incremental 2-4% reduction in useless prefetches for these programs, but generally has little effect on the rest of the benchmarks. Overall, only two programs show significant benefits from the more general forms of reuse analysis, but for these programs, we are able to double the fraction of useless prefetches eliminated. Most programs show little or no reduction in prefetch coverage; the exceptions are "`apsi`", "`swim`"and "`wave5`". Inspection of the L2 coverage factors (not shown) indicated that these effects are due to L1 cache conflicts, not to any loss of static coverage. We conclude that for the benchmarks we use, the more general forms of reuse analysis provide fairly small benefits, due in part to the fact that the analysis targets the 8K L1 cache, which is unable to exploit program locality over longer periods of program execution.

Group-spatial reuse analysis for irregular references

Group-spatial reuse analysis seeks to determine situations where two different array references always access the same cache line for a given point within the iteration space. This type of reuse analysis is fairly straightforward for regular (affine) subscript functions; the compiler can symbolically subtract the subscript functions of the two references, and check to see if the result is 0 for all subscript positions except the column, which must be a constant c related to the cache line size. Question arise when applying this sort of analysis to irregular references. The example in Figure 5.19 illustrates.

```
do i = 1, n
   a(i) = y(i) - y(i+1)
   d(i) = x(q(i)) - x(q(i)+1)
enddo
```

Figure 5.19 Group-spatial reuse among irregular references

The two references to array "`y`" share group-spatial reuse according to our definition. The two references to "`x`" also seem share group-spatial reuse, but since the values in the index array "`q`" are unknown at compile-time, we can't be sure. In particular. it may be possible that the array element `x(q(i))` falls on the last word in a cache line, and the element `x(q(i)+1)` falls on the first word of the next cache line. In such a case, it might be detrimental to eliminate either of the prefetches.

	Change in coverage				Change in	Change in
	L1 cache		L2 cache		useless prefs	exec time
Program	Base	Virtual	Base	Virtual		
cgm	-	-	-	-	-	-
nasbuk	-	-	-	-	-	-
wave5	2.5%	-3.4%	-6.4%	-6.9%	-10.5%	-1.2%
su2cor	-3.0%	-2.7%	-3.8%	-4.2%	-0.4%	0.6%

Figure 5.20 Effects of group-spatial reuse analysis on selected irregular applications

Figure 5.20 shows the increase or decrease in coverage and in the useless prefetch percentage when group-spatial reuse is applied to irregular references (we show only the 4 programs that use indirection arrays). Group-spatial reuse analysis appears to be detrimental to "`su2cor`"; there is very little change in the useless prefetch percentage, and coverage factors for the L1 and L2 cache decrease, indicating that useful prefetches are being eliminated. For "`wave5`", the results are less conclusive: fewer useless prefetches are issued, but most of the coverage factors decrease as well. In terms of actual execution time, one programs is degraded very slightly and one program is improved. We conclude that at least for the programs in our benchmark suite, there is little to be gained by eliminating prefetches in these situations. For faster processors, where instruction overhead is less of an issue and prefetch coverage is more important, there may be a more pronounced penalty in execution time.

5.6.2 Techniques related to instruction overhead

5.6.3 Applying strip-mining to moderate loop unrolling

As described in Section 3.4.2, our compiler also includes support for using a combination of strip-mining and unrolling in situations where a large unroll degree is deemed to be undesirable. In this section we examine the effects of this transformation on compile times, executable sizes, and instruction overhead.

We find that from the point of view of the performance of the generated program, it is always a win to use the unroll degree dictated by the cache line size, even when it results in very long compile times. Our goal in applying strip-mining is rather to create a final program that makes a good compromise between compile time and execution time.

	Percentage increase in:			
	Compile time		Executable size	
Program	LS=64	LS=128	LS=64	LS=128
`appbt`	0.3%	-33.6%	0%	-14.5%
`applu`	-3.0%	-3.0%	3.1%	0%
`appsp`	5.1%	-7.4%	1.8%	-1.8%
`apsi`	13.6%	38.5%	9.0%	20.7%
`cgm`	10.5%	52.6%	0%	12.5%
`erl`	12.5%	35.4%	8.3%	12.5%
`flo52`	58.6%	477.7%	15.6%	43.7%
`hydro2d`	14.8%	75.3%	3.0%	18.1%
`mgrid`	52.1%	147.8%	17.6%	23.5%
`nasbuk`	0%	14.2%	0%	0%
`ocean`	11.8%	23.7%	4.8%	4.8%
`su2cor`	15.1%	60.4%	8.1%	26.5%
`swim`	26.3%	115.7%	6.2%	12.5%
`tomcatv`	27.2%	109.0%	4.4%	7.3%
`wave5`	19.3%	65.7%	4.5%	10.3%

Figure 5.21 Compile time and executable size increase with longer cache lines

Figure 5.21 shows the effects of longer line sizes on the compile times and executable sizes of the programs to which prefetching has been applied. The data show percentage increase in executable size and compile time relative to the default line size of 32 bytes; compile times are based on the wall clock time to compile and link the transformed programs using the Sun `f77` compiler on an unloaded UltraSparc workstation with 64 megabytes of memory. Most of the programs show an increase in compile time and executable size; the exceptions are `appbt`, `appsp`, and `applu`. For these programs, the larger unroll factors cause the compiler to suppress more of the steady state loops in the programs (if the compiler can prove that the steady state loop never executes, due to a small loop trip count, then no steady state will be generated). As a result, compile time actually decreases for these three programs as the line size is increased. For many of the other programs, however, compile time and executable size increase drastically when larger line sizes are used, due to the larger loop unrolling factors. Compile-time increases of this magnitude (50% to 400%) are not always acceptable, given that there may not be a corresponding increase in program performance when using larger line sizes.

	Percentage increase				Absolute increase in IPP	
	Compile time		Executable size			
Program	LS=64	LS=128	LS=64	LS=128	LS=64	LS=128
appbt	1.4%	-34.7%	0	-12.5%	0.0	0.0
applu	-2.2%	-2.2%	3.1%	3.1%	0.0	0.0
appsp	4.0%	-27.8%	0%	-9.2%	0.2	0.2
apsi	4.0%	11.2%	6.4%	9.0%	0.3	0.4
cgm	5.2%	15.7%	0%	6.2%	0.8	0.3
erl	-14.5%	-20.8%	4.1%	0%	5.9	-2.1
flo52	3.5%	4.4%	1.5%	0%	0.7	0.6
hydro2d	8.6%	18.5%	3.0%	9.0%	1.0	3.6
mgrid	23.9%	26.0%	11.7%	11.7%	3.6	3.8
nasbuk	0%	14.2%	0%	0%	0.8	1.4
ocean	8.9%	6.9%	2.4%	0%	0.0	0.2
su2cor	1.4%	2.8%	4.0%	4.0%	3.7	3.3
swim	-10.5%	-15.7%	0%	0%	9.4	15.7
tomcatv	18.1%	9.0%	4.4%	4.4%	3.9	4.7
wave5	-1.4%	5.9%	1.6%	2.9%	1.1	1.4

Figure 5.22 Compile time and executable size increase, with strip-mining

Figure 5.22 shows the same statistics with the compiler's register pressure driven loop strip-mining option (described in Section 3.4.2) is enabled. With strip-mining of the steady-state loop, we are to hold the `f77` compile time increase down to a maximum of 26% in the case of "`mgrid`" for the 128-byte line case (as opposed to an increase of 148% when no strip-mining is used). Strip-mining does result in increased overhead– the programs for which strip-mining has the most benefit in terms of compile-time moderation (`hydro2d`, `mgrid`, `swim`, and `tomcatv`, for example) show increased IPP values. It should be noted that IPP is already very low for these programs when compiled for larger line sizes, however (see Figure 5.14), for most of these programs IPP is still below 0 even with strip-mining enabled, indicating that the strip-mined version of the code is still getting most of the benefits of unrolling.

5.6.4 Techniques to improve prefetch scheduling

We now evaluate the effects of our two new scheduling optimizations, prolog overlap and outer loop pipelining. These transformations are described in Sections 3.4.4 and 3.4.5. Figure 5.23 shows the number of loops selected for each transformation, assuming a prefetching latency of 175 cycles. In these experiments prolog overlap was applied to every loop nest that met the restrictions on loop bounds, subscript expressions, etc., whereas we only applied outer loop pipelining to loop nests where the inner loop had "`WorstCase`" pipeline characteristics (see Section 5.5.3). We see that prolog overlap is more widely applicable than outer loop pipelining, which is to be expected given the more restrictive criteria required for the latter technique. In the following sections we provide simulation data on each method.

Program	Loops	Inner loops	Outer loop pipelining	Prolog overlap
appbt	207	98	23	48
applu	182	83	32	41
appsp	251	110	27	71
apsi	389	228	2	51
cgm	31	26	1	1
erl	82	38	2	27
flo52	152	85	21	61
hydro2d	161	104	0	50
mgrid	65	30	6	14
nasbuk	10	10	0	0
ocean	227	130	0	17
su2cor	130	90	0	4
swim	24	16	0	6
tomcatv	16	9	0	5
wave5	414	301	4	70

Figure 5.23 Applicability of scheduling optimizations

Prolog overlap

Figure 5.24 shows the effects of prolog overlap on program performance. The first column shows the percentage of late prefetches that were eliminated, relative to the default prefetched version of the program. The second column shows the percentage increase in overshoot, *i.e.*, the number of prefetched lines that arrive in the L1 cache but are displaced before they can be used. The final column shows the effect on execution time; positive numbers indicate an improvement in performance.

Program	Late prefs eliminated	Increase in overshoot	Execution time improvement
appbt	93.2%	52.4%	1.2%
applu	68.7%	2.2%	-0.2%
appsp	81.2%	87.6%	-1.8%
apsi	11.7%	0.7%	-0.6%
cgm	1.0%	-1.7%	-0.2%
erl	95.7%	273.9%	-0.2%
flo52	87.3%	0.6%	-1.2%
hydro2d	18.1%	51.6%	0.5%
mgrid	81.2%	-0.5%	0.1%
nasbuk	0%	0%	0%
ocean	-0.2%	-0.1%	-0.1%
su2cor	57.8%	3.6%	-1.9%
swim	99.4%	2.9%	0.4%
tomcatv	99.7%	4.2%	-1.4%
wave5	-2.7%	-4.0%	-0.8%

Figure 5.24 Effects of prolog overlap on performance

The data indicate that prolog overlap is effective in reducing the number of late prefetches for many of the programs, yet the overall impact on performance was minimal. Four programs also show an increase in overshoot, however: `appsp`, `appbt`, `erl`, and `hydro2d`. In the case of `erl` the huge increase is something of a red herring, since in the standard prefetching version there is very little overshoot to begin with. For the other three programs, however, the increased level of overshoot is significant, effectively cancelling out some or all of the gains seen due to the reduction in late prefetches. The overall effect on execution time due to prolog overlap is relatively minor. We attribute this to the fact that in the uniprocessor case, the average penalty incurred due to a late prefetch is fairly small (see Figure 5.16). In Chapter 7 we take another look at scheduling optimizations for distributed shared memory multiprocessors, where the average penalty for a late prefetch is much higher.

Outer loop pipelining

Our results indicate that outer loop pipelining can be somewhat unpredictable in its effects. We have found several factors that account for this unpredictability. First, when outer loop pipelining is applied, the prefetching distance granularity becomes much higher, since the distance must be in terms of outer loop iterations instead of inner loop iterations. As a result, prefetching distance over- and under-estimates are more common. Second, outer loop pipelining can have a number of effects on instruction overhead, both positive and negative. When outer loop pipelining is applied to loops with very little computation, it can convert the loop's large inner-loop prefetching distance to a much smaller outer-loop prefetching distance. This means that more time will be spent in the unrolled steady state portion of the pipeline, reducing instruction overhead. On the other hand, the use of strip-mining can introduce additional instruction overhead, since the transformation effectively adds another loop level.

Program	Late prefs eliminated	Increase in replacements	Execution time improvement
`appbt`	49.7%	-98.7%	1.5%
`applu`	46.6%	-95.3%	6.9%
`appsp`	-31.6%	-50.5%	3.8%
`apsi`	-0.2%	-4.8%	-2.1%
`flo52`	-567.0%	-3.7%	0.2%
`mgrid`	7.5%	740.4%	-2.1%

Figure 5.25 Effects of outer loop pipelining on performance

Figure 5.25 shows the effects of outer loop pipelining for our benchmark suite. Only six of the programs showed any noticeable changes; we show the results for these programs. As in the previous section, we give data on percent reduction in late prefetches, percent increase in overshoot, and overall improvement (decrease in execution time). The three programs with short inner loops (`appbt`, `appsp`, and `applu`) show modest improvements in performance. In the case of `appsp`, however, most of the gains come from a reduction in instruction overhead and not from scheduling effects. In addition, two of the other programs (`apsi` and `mgrid`) are degraded by outer loop pipelining. In the case of `apsi`, the degradation is from increased instruction overhead as a result of strip-mining. In the case of `mgrid`, the slowdown is due to additional conflicts between the current working set and the prefetched lines that are arriving earlier. The program `flo52` actually shows in increase in late prefetches (something of a red herring, since it had few late prefetches to begin with), but gains a little in terms of instruction overhead, resulting in a small improvement.

5.7 Summary

In this chapter we give the results of a detailed evaluation of compiler performance for uniprocessor prefetching, using the metrics developed in Chapter 4. Our use of simulation allows us to provide a clear view of the compiler's contributions and to precisely quantify the effects of various compiler strategies.

In the area of prefetch target selection, we find that in most cases poor prefetch coverage is due to cache conflicts and not due to our prefetching policy (*i.e.*, inserting prefetches only for references nesting within inner loops). By providing a breakdown of uncovered cache misses by lexical nesting, we show that in programs where prefetching has been applied, the bulk of the remaining cache misses are still within inner loops, suggesting that compiler developers should continue to focus on loops, as opposed to references not enclosed in loops. Our findings indicate that many programs exhibit a significant percentage of useless prefetches, indicating that existing compiler analysis techniques are often unable to effectively recognize and/or exploit the available reuse. We describe several improvements to our default reuse analysis strategy, experimentally characterizing their effects on useless prefetches. We also provide empirical data on prefetching overshoot; the data from these experiments suggest that for this particular benchmark suite, there is little to be gained by adding optimizations designed to reduce overshoot.

With respect to instruction overhead, we experimentally characterize the instruction overhead for our compilation scheme; the results show that a prefetching strategy based on shadow arrays provides a viable platform for studying compiler-directed software prefetching. In the area of prefetch scheduling, we provide an experimental study of the pipeline characteristics for the programs in our benchmark suite, as well as giving data on the frequency of late prefetches and the penalties they incur. A number of the programs in our benchmark suite have very poor pipeline characteristics, leading to high levels of instruction overhead and non-optimal prefetch scheduling. We perform experimental evaluations of two new compiler techniques for improving scheduling: outer loop pipelining and prolog overlap. Our results show that these techniques are effective in reducing the number of poorly scheduled prefetches, although the performance improvements are very modest, due to the relatively small penalty incurred by late prefetches on our simulated uniprocessor configuration.

Chapter 6

Cross-loop Reuse Analysis and Transformations

6.1 Introduction

In this chapter we describe the design of a data-flow framework for detecting *cross-loop reuse*. Cross-loop reuse takes place when a set of data items or cache lines is accessed in a given loop nest and then accessed again within some subsequent portion of the program, usually another outer loop nest. In contrast to *intra-loop* reuse, which occurs during the execution of a single loop nest, cross-loop reuse is hard to analyze using traditional dependence-based techniques [13, 99]. Dependence analysis can provide very detailed information about the memory access patterns within a loop, but applying it to larger regions within a procedure is difficult, especially if the region in question contains control flow or procedure calls.

Rather than operating on the granularity of individual statements or array references, as is the case with dependence analysis, our analysis framework uses array-section analysis to reason about reuse in terms of entire regions within an array. The array section information is then used as input to a data-flow solver, which deals with intra-procedural control flow. The framework is designed to account for cache size when gathering reuse information, and when used in an interprocedural setting, it also provides a mechanism for summarizing the effects of procedure calls.

Cross-loop reuse analysis provides information that is useful for compiler-directed software prefetching. As with intra-loop reuse analysis, cross-loop reuse analysis allows the compiler to identify situations where the prefetch for a given reference can be eliminated, decreasing prefetching instruction overhead. Section 3.4.3 outlines how our compiler uses cross-loop reuse analysis to eliminate redundant prefetches, and in Section 5.6.1 we give experimental data on the effectiveness of this strategy in practice. Cross-loop reuse analysis also forms the underlying basis of our framework for compile-time prediction of coherence misses in parallel programs, described in Chapter 7.

Additionally, cross-loop reuse information can also be used to directly drive a number of transformations that enhance locality and improve cache utilization, including loop fusion and loop reversal [99]. Although these transformations are not new, their impact on cache behavior has not always been easy to predict, making them difficult to apply.

An outline of this chapter is as follows. In Section 6.2 we describe the details of our framework. In Section 6.3 we outline how the information provided by the framework can be applied, and in Section 6.4 we describe the results of some experiments using our framework to directly drive locality-enhancing transformations. In Section 6.5 we discuss related work, and finally in Section 6.6 we summarize our findings.

6.2 Analysis framework

In this section, we describe the details of our analysis. We begin by we outlining some of the key design goals for the problem we are solving, and discuss how we accomplish these goals. Section 6.2.1 introduces the control-flow representation used by our data-flow solver. In Section 6.2.2, we describe the domain over which our data-flow solver operates (*i.e.*, the sets propagated during the analysis). Sections 6.2.3, 6.2.4 and 6.2.5 describe the actual data-flow equations used, along with their inputs and the details of the solver. In Section 6.2.6, we discuss incorporation of cache size constraints. In Section 6.2.7, we briefly cover the algorithmic complexity of some of the operations performed in the framework. In Section 6.2.8, we describe how this framework can be used in an interprocedural setting.

Important design considerations

At a high level, the goal of our framework is to determine the set of array regions accessed on all paths that reach a point X within a given program. In many respects, this problem is similar to the well-known data-flow problem of "Available Expressions" [4]. We formulate a solution to our problem, therefore, using data-flow analysis.

We do not use a traditional data-flow solver, however, since our problem is unusual in some important respects. First, the problem requires that we explicitly take into account the loop structure of the program, as opposed to treating all control flow in an identical fashion. This is necessary because a given subscripted reference will access a different region in its array depending on how much of the surrounding loop context is taken into account (see Section 3.4.1 for an example).

A second aspect of the problem complicates the analysis: we want to take into account the size of the cache. The tools we have developed for this sub-task are difficult to combine with traditional flow analysis techniques. The solver we use allows us to develop a framework for detecting reuse without considering cache size constraints, and then factor in the cache size if necessary (see Section 6.2.6).

We use array-section analysis extensively in this framework [11, 15, 17, 45]. See Section 3.4.1 for a brief introduction to this form of analysis.

6.2.1 Control flow representation

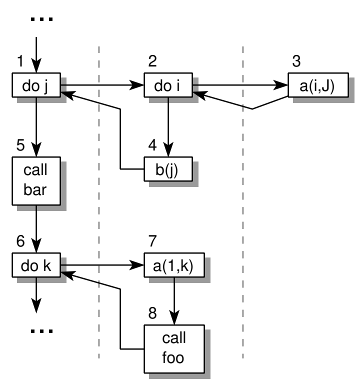
Rather than using a standard control-flow graph (CFG), this framework uses an *interval-flow graph* (IFG), developed by von Hanxleden and Kennedy [44]. Our use of the interval-flow graph allows us to take the loop structure of the program into account explicitly. The IFG is constructed by starting with a normal CFG and then partitioning the nodes and edges in the graph into categories based on Tarjan interval analysis.[89] [4]

A Tarjan interval $T(h)$ is a set of CFG nodes that corresponds to a loop within the program, where h is a unique header node (with $h \notin T(h)$). Intuitively, $T(h)$ together with h form a strongly connected region within the CFG. When $x \in T(h)$, we say that $\textsc{Header}(x) = h$.
Each interval-flow graph $G = (N, E)$ has a unique root node, $\textsc{Root}$, that can be viewed as the header node for the interval corresponding to the entire procedure. We define $\textsc{Level}(n)$ as the loop nesting level of node n, counting from the outside in, where $\textsc{Level}(\textsc{Root})$ is defined as 0. For a given interval $T(n)$, we define $\textsc{Children}(n)$ as the set of nodes $\{\ p \in T(n) : \textsc{Level}(p) = \textsc{Level}(n) + 1\ \}$, that is, the nodes in the interval headed by n that are immediate descendants of n. Figure 6.1 shows an example program fragment along with its interval-flow graph.

Each edge $(x \rightarrow y)$ in the IFG is placed into one of the following categories:

[4]The interval-flow graph (a data structure) should not be confused with interval-based data-flow analysis (an algorithm). Data-flow analysis techniques that use the interval-flow graph generally do not include the notion of collapsing intervals in the CFG into single nodes, as in interval-based data-flow analysis.

```
do j = 1, n
  do i = 1, n
    ... = a(i,j)
  enddo
  ... = b(j)
enddo
call bar(a)
do k = 1, n
  ... = a(1,k)
  call foo(a, k)
enddo
```



Intervals:
T(1) = { 2,3,4 }
T(2) = { 3 }
T(6) = { 7, 8 }

Figure 6.1 Example sub-program with interval-flow graph

Entry: *iff* $y \in T(x)$; an edge from an interval header x to a node within the interval.

Cycle: *iff* $x \in T(y)$; an edge from within an interval back to the header of that interval.

Jump: *iff* $\exists h : x \in T(h),\ y \notin \{\ T(h) \cup h\ \}$; a jump out of a loop (*i.e.*, an edge from a node in one interval to a node outside the interval that is not the h, header node).

Flow: *iff* $\forall h : x \in T(h) \iff y \in T(h)$; an intra-interval edge (*i.e.*, an edge that is none of the above).

To refer to the predecessors and successors of a given node, we use the following terminology:

Preds(n): The set of nodes $\{\ x : \exists e \in E,\ e = (x, n)\ \}$

Succs(n): The set of nodes $\{\ x : \exists e \in E,\ e = (n, x)\ \}$

The PREDS(n) and SUCCS(n) notation may be further qualified by adding a superscript containing the desired edge type. For example, PREDS$^F(n)$ is the set of nodes that are at the source of a FLOW edge whose sink is n, SUCCS$^{EJ}(n)$ is the set of nodes that are that are at the sink of an ENTRY or JUMP edge whose source is n, and so on.

After construction of the IFG, the graph is post-processed to insure that each interval has at most one CYCLE edge, *i.e.*, for each non-empty interval $T(h)$ there exists a unique $n \in T(h)$ such that $(n, h) \in E$. This sometimes requires the insertion of *synthetic* nodes and edges [43].

In addition, we define two types of partial orderings on N, as follows:

FORWARD/BACKWARD: Given a FLOW/JUMP edge (m, n), a FORWARD order visits m before n, whereas a BACKWARD order visits m after n.

UPWARD/DOWNWARD: Given $m, n \in N$ such that $m \in T(n)$, an UPWARD order visits m before n, whereas a DOWNWARD order visits n before m.

These two orderings are orthogonal and may be combined (for example, a FORWARD and DOWNWARD order).

6.2.2 Data-flow universe

The universe for this data-flow problem consists of sets of array section descriptors, where each descriptor is composed of the name of the array in question and a symbolic representation of the region accessed within the array. We call these sets *reuse summary sets*. An example of a reuse summary set might be

$$\{\ \mathtt{a(1:10)},\ \ \mathtt{a(75:n)},\ \ \mathtt{b(1:n)}\ \}$$

As can be seen from the set above[5], a reuse summary set may contain more than one region within a given array. The interpretation of the set depends on the context; it may represent a set of array sections available on entry to a given block, generated within a given loop, etc.

[5]In this example we are showing only the geometric region component of the `DAD`. The actual `DAD` contains more detailed information on the array access.

6.2.3 Initial information

For each basic block n that contains one or more array references, we compute the set $GEN_{INIT}(n)$. This reuse summary set contains an array section for each array reference within the block. The sections initially assigned to $GEN_{INIT}(n)$ are computed with respect to the innermost loop level. During the analysis, when the section is propagated up out of an enclosing loop, the region of the section is reconstructed to take the loop in question into account.

During the computation of the initial set for a block, we attempt to coalesce sections that are adjacent or identical. For example, we might try to collapse the set { `a(i,j), a(i+1,j)` } into { `a(i:i+1,j)` }. This coalescing is only performed when it will result in no loss of precision.

$$GEN_{IN}(n) = \bigwedge_{p \in PRED^F(n)} \{ \; GEN_{OUT}(p) \; \} \tag{6.1}$$

$$GEN_{LOC}(n) = \left\{ \begin{array}{ll} GEN_{OUT}(\textsc{Lastchild}(n)) & \text{if } n \text{ is an interval header} \\ GEN_{INIT}(n) & \text{otherwise} \end{array} \right\} \tag{6.2}$$

$$GEN_{OUT}(n) = GEN_{IN}(n) \vee GEN_{LOC}(n) \tag{6.3}$$

$$REACH_{IN}(n) = \left\{ \begin{array}{ll} REACH_{IN}(\textsc{Header}(n)) & \text{if } \textsc{Preds}^E(n) \neq \emptyset \\ \bigwedge_{p \in PRED^F(n)} \{ \; REACH_{OUT}(p) \; \} & \text{otherwise} \end{array} \right\} \tag{6.4}$$

$$REACH_{OUT}(n) = REACH_{IN}(n) \vee GEN_{LOC}(n) \tag{6.5}$$

Figure 6.2 Reuse equations

6.2.4 Reuse equations

Our goal for the flow analysis is to compute the set $REACH_{IN}(n)$ for all of the nodes in the IFG. The set $REACH_{IN}(n)$ corresponds to the set of array sections that reach node n on all paths from Root to n. We compute $REACH_{IN}$ using the equations in

Figure 6.2; evaluation of the equations is controlled by the algorithm in Figure 6.3. All sets are initially empty (with the exception of GEN_{INIT}).

Intuitively, $GEN_{IN}(n)$ corresponds to the set of sections accessed by the nodes prior to n within the interval that contains n. $GEN_{LOC}(n)$ corresponds to the set of sections accessed within n (if n is not a loop header) or the sections accessed within the interval headed by n (if n is a loop header). $GEN_{OUT}(n)$ combines GEN_{IN} and GEN_{LOC} to form the set of sections that flow out of n (taking into account only the nodes in the interval containing n). The GEN values are computed starting with innermost loops and working outwards.

$REACH_{IN}(n)$ corresponds to the set of sections reaching n from within n's interval and (possibly) from some previous loop outside n's interval. $REACH_{OUT}(n)$ is the set of sections flowing out of n, where the sections may be generated locally or they may reach n from some previous loop somewhere in the program.

---

procedure *ComputeReuse*

inputs: interval-flow graph $G = (N, E)$
$GEN_{INIT}(n)$ for all $n \in N$
outputs: $REACH_{IN}(n)$ and $REACH_{OUT}(n)$ for all $n \in N$

begin
  forall $n \in N$ in UPWARD and FORWARD order:
    compute equations 6.1, 6.2, and 6.3
  forall $n \in N$ in DOWNWARD and FORWARD order:
    compute equations 6.4 and 6.5
end

Figure 6.3 Procedure for computing reuse equations

6.2.5 Meet ($\bigwedge$) and join ($\bigvee$) operators

In the equations above, the $\vee$ and $\wedge$ operators play an important role. The $\wedge$ operator is used to merge together sets of sections at join points, and the $\vee$ operator is used to merge together local information with incoming information (*i.e.*, the $\vee$ operator models the effects of passing through a block).

The left hand side of Figure 6.4 illustrates a situation where the $\wedge$ operator is needed. In this case, the analysis must merge together the sections reaching the node `X` from its two predecessors `M` and `N`, taking into account the fact that the particular path to `X` is unknown.

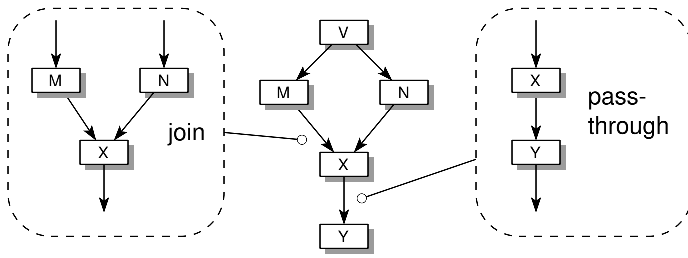


Figure 6.4 Control flow

The right hand side of Figure 6.4 shows a situation where the $\vee$ operator is needed. The sections flowing out of `X` need to be combined with the sections locally generated at `Y`, but in this case we know that flow of control must reach `Y` if it reaches `X` (unless the program diverges).

6.2.6 Incorporating cache constraints

The data-flow framework we have described in the previous sections predicts reuse without regard to cache size or other resource constraints. The resulting information does not give any hint as to distance between successive uses of array data, but rather only indicates that a set of array locations is reused. In most situations, we would like to know whether the distance between the successive uses is small enough that the reused items will be found in cache, for some fixed cache size. This section describes how we modify our framework to take cache size and organization into account. The key change we make is to the implementation of the $\vee$ and $\wedge$ operators, as will be shown shortly.

Array section age:

First, we introduce the concept of the "age" of an array section with respect to a particular point in the program. We define the age of a given section as the number of cache misses that have taken place since the first element of the section was brought into the cache. Array section age, a compile-time construct, is intended to model the finite-cache capacity effects on a particular array at run-time.

During the analysis, we associate age values with each of the sections in a reuse summary set. When an array section is first added to a reuse summary set (corresponding to the point where it is first brought into the cache) we assign it an initial age value based on its volume. As the section is propagated to other points in the CFG, other accesses will start to displace the section from the cache; when this happens, the age of the section is incremented. Eventually the age of the section reaches a cutoff, at which point we consider the section "dead" (*i.e.*, totally displaced from the cache), and it is eliminated from its reuse summary set.

In order for this scheme to work, we need to have a mechanism for determining the number of cache lines accessed by an array reference within a loop; this is in fact a research problem all by itself [30]. Our approach is to estimate the volume of the DAD for the reference, using a simple technique similar to the RefCost algorithm developed by Carr, McKinley, and Tseng [19].

Cache organization:

Our framework is not equipped to predict cache conflicts due to limited associativity. We instead conservatively assume that cache conflicts will reduce the amount of reuse that takes place by a fixed factor. We currently estimate the "effective" size of the cache used in the analysis by multiplying the actual cache size by $1 - \frac{1}{2^S}$, where S is the set associativity of the cache. We base this heuristic on the commonly used "2:1 rule of thumb", a conjecture which states that a direct-mapped cache of size N has approximately the same miss rate as a 2-way set-associative cache of size $N/2$ [47].

$\vee$ operator for finite caches:

For the finite cache case, we use a more sophisticated $\vee$ operator that models the cache effects when execution passes through a given node (shown in Figure 6.5). Given the set of sections flowing into block N (IN) and the set of sections accessed locally within N and N's descendants (LOC), this new $\vee$ operator computes the OUT set, taking into account the level of reuse and the size of the cache. The algorithm is based on the observation that a given section $S \in LOC$ will cause cache misses only if it is not contained in some section $S' \in IN$.

```
procedure FiniteCache-∨

inputs:   IN (incoming reuse summary set)
          LOC (reuse summary set for locally accessed data)
outputs:  OUT (outgoing reuse summary set)

begin
   volume ← 0
   R ← ∅
   forall x ∈ LOC:
      if ∃ y ∈ IN such that x is contained in y then
         R ← R ∪ { y }
      else
         volume ← volume + (cache line volume of x)
      endif
   endfor
   OUT ← LOC ∪ (IN - R)
   forall x ∈ OUT:
      increment age value of x by volume
      remove x from OUT if age exceeds cache size cutoff
   endfor
end
```

Figure 6.5 $\vee$ for finite cache case

$\wedge$ operator for finite caches:

We also modify the $\wedge$ operator when estimating reuse for a finite cache. Set intersection is still the basis for the operator, but the age values of the sections must be adjusted as well. In particular, when the $\wedge$ operator merges the reuse summary sets for two incoming paths, it may encounter a section that appears in both sets, but has a different age value in each one. In this case, the $\wedge$ operator chooses the larger of the two ages when forming the result.

For example, consider the graph fragment in Figure 6.4. Suppose that we are applying the $\wedge$ operator to the sections reaching block X from its predecessors M and N, and suppose that blocks V, M, and X contain array accesses (each to a different array), and that N contains no accesses. When we apply the $\wedge$ operator, both input sets will contain the section from V (we assume here that the number of accesses in M is relatively small), however the age of the sections from V that arrive at X along the edge $M \rightarrow X$ will be larger than the age of the corresponding sections flowing through the edge $N \rightarrow X$ (due to the additional data brought into the cache in M). The $\wedge$ operator selects the larger age, in order to be conservative.

6.2.7 Complexity

Generating the GEN_{INIT} set for each block requires that we build a DAD for each array reference in the procedure. This process requires $O(D^2 * N)$ time per reference, where D is the number of dimensions of the array and N is the nesting depth of the reference. Each DAD takes $O(D^2)$ space, and most operations involving DADs (union, intersection, containment, comparison) take $O(D^2)$ time.

The flow analysis framework itself considers each node and edge in the IFG exactly twice. The $\wedge$ and $\vee$ operators for the unlimited-cache case are linear in the number of sections in the sets being operated on, but for the finite-cache case, $\vee$ takes $O(N^2)$ time in the worst case (where N is the number of sections in each set), since since each section in the set may have to be compared with every other section.

In practice, we have found that the time required by the framework is comparable to the time that it takes to perform dependence analysis for the procedure.

6.2.8 Interprocedural analysis

Array-section analysis was originally conceived as a means of summarizing the effects of procedure calls within loops. As a result, it is relatively straightforward to extend our framework to an interprocedural setting.

When invoked in the context of whole-program analysis, we use the following strategy. We analyze procedures starting with the leaves in the call graph and working up to the root, visiting a procedure only after all of the procedures it calls have been visited. When a procedure call is encountered within the IFG of the subroutine being analyzed, we take the previously computed $REACH_{OUT}$ set for the callee and use it as the GEN_{LOC} set for call (applying array reshapes if necessary, and translating the summary into the name space of the caller by substituting actuals for formals, etc.).

Summarizing call sites in this fashion is generally feasible only if the framework is being run with cache size constraints taken into account (without the size constraints, the reuse summary sets grow very large in the upper regions of the call graph).

Currently we are restricted to propagating information upwards in the call graph. We do not, for example, take advantage of context within calling routines to reason about reuse within a callee. Our analysis does not currently handle programs whose call graphs contain cycles.

6.3 Applications of cross-loop reuse information

This section describes some of the ways in which cross-loop reuse information can be used by a compiler. It should be noted that exploiting cross-loop reuse information tends to be more difficult than exploiting intra-loop reuse information. The larger the region within the program in which the reuse is taking place, the more obstacles the compiler must overcome to apply restructuring.

6.3.1 Elimination of useless prefetches

Our compiler uses cross-loop reuse information to detect situations where prefetches are unnecessary. Given a reference R within a loop L, we use the following test to determine whether the prefetch for R can be eliminated. We construct an array-section descriptor S_R that summarizes the region accessed by R with respect to L. We then check to see if the section S_R is contained in some section $S_Q \in REACH_{IN}(L)$, for R's array. If S_Q exists, and if the same $S_Q \in REACH_{OUT}(L)$, (indicating that S_Q

remains resident in the cache during the execution of the L), then we eliminate the prefetch for R.

6.3.2 Locality-enhancing loop transformations

Cross-loop reuse can be used to predict the profitability of locality-enhancing transformations involving pairs of adjacent loop nests. These transformations include (but are not limited to) loop fusion and loop reversal.

Loop fusion:

Loop fusion combines two adjacent loops with identical headers into a single loop, reordering the sequence of references made by the two loops. Loop fusion has the potential to convert cross-loop reuse into intra-loop reuse, which can greatly enhance cache utilization.

Our framework can supply enough information to cheaply predict the profitability of loop fusion. The compiler can examine the $REACH_{IN}$ set for a given loop nest to see what sections reach the loop. If the intersection of the $REACH_{IN}$ set with the GEN_{LOC} set of the loop is sufficiently large and the sections in the $REACH_{IN}$ set originate from the immediately preceding loop nest, then loop fusion will be profitable (the degree of profitability will be dependent on the volume of the intersection).[6] Once it is established that the transformation is profitable, then the compiler can apply the more costly dependence-based techniques to determine whether fusion is safe [99].

Loop reversal:

A weaker but slightly more widely applicable technique is loop reversal. This optimization can provide benefits only in proportion to the size of the cache, thus it works best for very large (presumably secondary or tertiary) caches. Consider two consecutive outer loops that both access a single large vector, where vector itself is too big to fit entirely in cache. Even though the vector is reused, there is no cache reuse, since when the second loop begins execution, the first elements of the vector have been flushed from cache. However if we reverse the second loop, then the last elements of the vector from the previous loop are likely to still be in cache. This optimization relies on the *traversal order* component of the DAD representation, which

[6]This requires that we tag each section with the ID of the loop in which it originated.

captures the direction and stride of the access in each array dimension (see [12] for the details). One advantage of loop reversal is that the target loops do not have to be directly adjacent. There may be intervening code between the loops, provided that the code does not destroy the useful reuse.

6.3.3 Transformation selection

Even if the compiler can cheaply predict when a transformation is going to be profitable, there still remains the problem of deciding the sequence of transformations to apply within a procedure. A given loop nest may be optimized in several different ways (fused with its predecessor or with its successor, for example). Selecting the optimal set of transformations is a very difficult problem; optimizing temporal locality using loop fusion alone is NP-hard [52]. As a result, the compiler must resort to heuristics to choose the set of transformations to apply.

Our transformation selection procedure is as follows. For each loop nest N, we create a hash table (the "reuse score table") whose entries are tuples of the form $\langle L, R\rangle$, where L is a loop ID number and R is an estimate of the number of cache lines reused from loop L. We generate the reuse score table for a loop N while computing the value of equation 6.3 in Figure 6.2; when a section $x \in GEN_{IN}(N)$ intersects with a section $y \in GEN_{LOC}(N)$, we compute the volume of the intersection of x and y, determine the loop L' in which x originated, and update the L' entry in N's table. We calculate the total score for a loop by summing the values of all of the entries in its reuse score table.

We then use a greedy algorithm to select the loops to optimize; we first consider the loop nest with the highest reuse score value, optimize it if possible, then consider the loop with the next highest score, and so on. We first apply loop fusion, then loop reversal. We limited loop fusion to the outermost loop in each pair of adjacent loop nests in our study (for nests involving more than one loop). Fusion of inner loops often results in decreased instruction count, whereas in this study we opted to focus primarily on cache effects.

6.4 Experimental results

In this section, we report the results from an experimental study in which we apply our techniques to ten programs from the SPEC95 floating point benchmark suite [96]. The infrastructure for this experiment consists of a Fortran transformation engine, including the cross-loop reuse analysis framework, and an execution-driven simulator for gathering instruction counts and cache statistics.

6.4.1 Compilation

The compilation phases for this experiment are shown in Figure 6.6. In order to focus solely on cross-loop transformations, we disabled most of the phases of our compiler related to software prefetching (as described in Chapter 3), leaving only the cross-loop reuse analysis framework and cross-loop transformations. All of the analysis and transformation steps shown are performed automatically.

Phase	Remarks
1. Front end processing	read and typecheck Fortran source build AST (abstract syntax tree)
2. Local analysis	build IFG for each procedure compute GEN_{INIT} for all $n \in$ IFG
3. Cross-loop analysis	run cross-loop framework for each function
4. Transformations	apply loop reversal and loop fusion based on cross-loop reuse info
5. Output	generate transformed Fortran source

Figure 6.6 Compilation stages

6.4.2 Simulator

We used a very simple cache simulator for this study, based on the tool `shade`, part of the SPARC Performance Analysis Toolkit [23]. `Shade` provides an extensible mechanism for writing execution-driven simulators; it operates by interpreting a SPARC executable and passing a trace of the instructions to a user-written trace analyzer. In our case, the trace analyzer counts instructions and simulates a particular cache configuration.

After the source-to-source transformer is run, the target programs are instrumented with calls to runtime routines to tag outer loop nests and to demarcate the regions of the program's address space containing array data. The instrumented programs are then compiled using the Sun `f77` Fortran compiler (version SC4.0 18/Oct/95).

The simulator deals with *data* cache behavior only; it does not simulate an instruction cache. We use the following cache parameters for our simulations. The L1 cache is 64 Kbytes, 4-way set-associative, with a line size of 32 bytes, and the L2 cache is 1024 Kbytes, also 4-way set-associative with a line size of 32 bytes. An LRU replacement policy is used within each cache set. Both caches are write-back, with an allocate-on-write-miss policy.

6.4.3 Benchmark programs

Figure 6.7 gives some of the summary characteristics of the programs we used for our experiments. "Functions" is the number of procedures in the program; "Lines" is the number of non-comment source lines. "Data" is the total size of all the arrays used by the program, in megabytes. "Runtime" shows the approximate wall clock running time on an unloaded SPARCStation 10 with 256 megabytes of memory. The "training" input files were used for these runs, in order to yield more reasonable simulation times [96]. The programs themselves were not modified, however,

6.4.4 Results

Figure 6.8 gives the raw instruction counts and cache metrics for the original untransformed programs. All numbers are in thousands. The "Instructions" column contains the total dynamic instruction count for the program. The L1 and L2 cache metrics are for accesses to array data only (*i.e.*, they exclude accesses to scalars, compiler-generated spill code, etc.). The "P-cycles" term in the final column is an approximation of the overall execution time of the program; it combines the total instruction count with the projected stalls due to cache misses. It is computed as follows:

$$\text{P-Cycles} = \text{IC} + (M_1 * P_1) + (M_2 * P_2)$$

where "IC" is the total instruction count, M_k is the total number of misses at level k, and P_k is the additional miss penalty at level k. For the purposes of this study, we assume a level 1 miss penalty of 10 cycles and an additional level 2 miss penalty of 80 cycles, for a total miss-to-main-memory time of 90 cycles.

Figures 6.9, 6.10, and 6.11 show the results for the transformed programs using purely static reuse analysis. Figure 6.9 gives a summary of the specific transformations applied to each of the programs. The numbers show for "candidates" indicate the total

Program	Functions	Lines	Data (MB)	Runtime (secs)
applu	16	1883	32.3	25
apsi	96	4872	9.4	61
fpppp	38	2408	0.5	6
hydro2d	40	1565	8.6	182
mgrid	13	445	7.5	192
su2cor	36	1753	23.8	551
swim	6	261	14.4	15
tomcatv	1	125	14.4	181
turb3d	23	1294	25.4	337
wave5	104	7372	41.4	65

Figure 6.7 Program characteristics

Program	Instructions	L1 hits	L1 misses	L2 hits	L2 misses	P-cycles
applu	329,240	66,962	5,962	4,731	1,231	487,340
apsi	2,333,872	362,521	5,427	5,413	4	2,388,462
fpppp	240,043	23,771	38	36	2	240,583
hydro2d	5,543,080	1,186,888	218,461	16,091	202,371	23,917,370
mgrid	12,566,952	3,445,822	119,285	102,698	16,587	15,086,762
su2cor	23,341,356	6,425,969	1,215,323	1,024,815	190,508	50,735,226
swim	481,252	119,409	8,784	21	8,763	1,270,132
tomcatv	6,398,033	1,647,655	172,041	26,139	145,901	19,790,523
turb3d	14,453,171	2,568,151	84,751	34,275	50,475	19,338,681
wave5	2,700,459	564,179	38,124	30,988	7,106	3,650,179

Figure 6.8 Simulation data for original programs [thousands]

		fusion		reversal	
Program	loops	fused	candidates	reversed	candidates
applu	168	2	4	20	111
apsi	298	1	2	5	150
fpppp	39	0	0	0	6
hydro2d	163	3	14	5	136
mgrid	57	0	1	7	36
su2cor	118	0	0	3	47
swim	24	0	4	0	22
tomcatv	16	0	2	0	8
turb3d	64	0	0	0	33
wave5	377	14	27	27	212

Figure 6.9 Transformation summary

Program	Instructions	L1 hits	L1 misses	L2 hits	L2 misses	P-cycles
applu	330,029	67,138	5,814	4,582	1,232	486,729
apsi	2,333,872	360,505	5,411	5,399	4	2,388,302
fpppp	240,042	23,518	39	37	2	240,592
hydro2d	5,542,709	1,197,341	218,262	18,518	199,744	23,704,849
mgrid	12,561,752	3,448,549	116,891	100,306	16,577	15,056,822
su2cor	23,341,354	6,423,729	1,217,500	1,027,017	190,482	50,754,914
swim	481,490	119,409	8,784	21	8,763	1,270,370
tomcatv	6,397,215	1,647,651	172,044	26,143	145,901	19,789,735
turb3d	14,668,168	2,555,806	95,870	45,379	50,490	19,666,068
wave5	2,686,798	559,578	38,891	31,779	7,081	3,642,188

Figure 6.10 Simulation data for transformed programs [thousands]

Program	Instructions	L1 hits	L1 misses	L2 hits	L2 misses	P-cycles
applu	0.23	0.26	-2.48	-3.14	0.08	-0.12
apsi	0	-0.55	-0.29	-0.25	0	0
fpppp	0	-1.06	2.63	2.77	0	0
hydro2d	0	0.88	-0.09	15.08	-1.29	-0.88
mgrid	-0.04	0.07	-2.00	-2.32	-0.06	-0.19
su2cor	0	-0.03	0.17	0.21	-0.01	0.03
swim	0.04	0	0	0	0	0.01
tomcatv	-0.01	0	0	0.01	0	0
turb3d	1.48	-0.48	13.11	32.39	0.02	1.69
wave5	-0.50	-0.81	2.01	2.55	-0.35	-0.21

Figure 6.11 Percent change between original and transformed (no profile data)

number of loops in the program that were determined to be legal candidates for fusion or for reversal. Figure 6.10 shows the raw data for the transformed programs (again, all numbers are in thousands). Figure 6.11 compares the transformed programs to the original programs in each category. As can be easily seen, there is very little overall change in program performance.

After a closer inspection of the results, we found that the framework appeared to be missing a number of important transformation opportunities. Our initial investigations suggested that the compiler was frequently forced to make conservative assumptions about reuse due to unknown loop bounds. To explore this hypothesis, we ran a new set of experiments in which the compiler incorporated profiling data into the analysis framework.

We gathered profiling data by running each program on the given input file and gathering the minimum, maximum, and average value for the loop bounds, loop step, and overall trip count. For loops whose bounds were invariant at run-time, the compiler substituted in the bounds from the profiling data when computing the GEN_{LOC} set for the loop in question. It should be emphasized that profiling data was only used in the portions of the analysis that determine profitability of transformations, not the safety of the transformations.

Figures 6.12, 6.13, and 6.14 show the results with the profiling information. By using profiling data, the compiler was able to compute more accurate estimates of the number of cache lines accessed in each loop nest, and was able to detect reuse in situations where previously it had to assume no overlaps, to be conservative. The number of loops fused went up from 20 to 24, and the number of reversed loops went up from 67 to 84. The results in Figure 6.14 indicate that two programs, `tomcatv` and `hydro2d`, showed improvements in performance as a result of the transformations. In both cases, the improvement is due to improved cache behavior; both programs show significant reductions in both L1 and L2 misses overall.

6.5 Related work

A number of researchers have developed compiler techniques useful for improving cache behavior [8, 16, 19, 66, 98]. Almost all of these techniques apply to individual loop nests, however, and are not designed to detect or exploit cross-loop reuse. Two exceptions are *loop fusion* and *affinity regions*.

		fusion		reversal	
Program	loops	fused	candidates	reversed	candidates
applu	168	2	4	20	111
apsi	298	1	2	5	150
fpppp	39	0	0	0	6
hydro2d	163	5	14	20	136
mgrid	57	0	1	7	36
su2cor	118	0	0	5	47
swim	24	1	4	1	22
tomcatv	16	1	2	0	8
turb3d	64	0	0	0	33
wave5	377	14	27	26	212

Figure 6.12 Transformation summary (with profile)

Program	Instructions	L1 hits	L1 misses	L2 hits	L2 misses	P-cycles
applu	329,562	67,122	5,813	4,584	1,229	486,012
apsi	2,333,873	362,635	5,289	5,277	4	2,387,083
fpppp	241,000	23,518	39	37	2	241,550
hydro2d	5,581,854	1,195,747	209,575	30,474	179,101	22,005,684
mgrid	12,561,753	3,446,434	118,673	102,110	16,563	15,073,523
su2cor	23,342,471	6,423,530	1,217,698	1,027,185	190,514	50,760,571
swim	480,793	119,411	8,783	29	8,754	1,268,943
tomcatv	6,396,957	1,663,997	155,700	23,096	132,603	18,562,197
turb3d	14,453,171	2,567,901	84,701	34,225	50,476	19,338,261
wave5	2,705,524	559,560	38,909	31,797	7,081	3,661,094

Figure 6.13 Simulation data for
transformed programs, with profile [thousands]

Program	Instructions	L1 hits	L1 misses	L2 hits	L2 misses	P-cycles
applu	0.09	0.23	-2.49	-3.10	-0.16	-0.27
apsi	0	0.03	-2.54	-2.51	0	-0.05
fpppp	0.39	-1.06	2.63	2.77	0	0.40
hydro2d	0.69	0.74	-4.06	89.38	-11.49	-7.99
mgrid	-0.04	0.01	-0.51	-0.57	-0.14	-0.08
su2cor	0	-0.03	0.19	0.23	0	0.04
swim	-0.09	0	-0.01	38.09	-0.10	-0.09
tomcatv	-0.01	0.99	-9.49	-11.64	-9.11	-6.20
turb3d	0	0	-0.05	-0.14	0	0
wave5	0.18	-0.81	2.05	2.61	-0.35	0.29

Figure 6.14 Percent change between
original and transformed (with profile)

Kennedy and McKinley have proposed using loop fusion to improve locality and cache behavior [52]. In a subsequent study, McKinley, Tseng, and Carr included loop fusion in their repertoire of transformations for an experimental study on compiler cache optimizations [19]. This study used dependence analysis to test for the profitability of loop fusion; loop reversal was not used as a locality-enhancing transformation. Their results showed that there are relatively few opportunities for applying loop fusion, but when applied, it can be very beneficial. In contrast, we find that loop reversal is much more widely applicable, although the benefits from reversal are less pronounced.

Affinity regions are a mechanism that allows a compiler or user to give locality-improving hints to the loop scheduler for a parallel program running on a shared-memory multiprocessor. By placing a set of consecutive parallel loops within an affinity region, the user or compiler is informing the loop scheduler that cross-loop reuse exists and that it should try to assign iterations to processors in such a way that the reuse is preserved. Compile-time identification of affinity regions was proposed by Appelbe *et al.* [8].

Our data-flow framework resembles that of Gross and Steenkiste [39]. However their framework is geared towards finding parallelism as opposed to detecting useful reuse for cache optimizations. Our techniques are also similar to those developed by Gupta, Schonberg, and Srinivasan for optimizing communication placement for programs running on distributed-memory multiprocessors [41]. The problem of communications placement is quite distinct from the problems we are attacking using our framework, however. Communications placement deals primarily with the flow of values within the program, whereas our analysis is more location-oriented. Their framework also does not provide a means of incorporating resource constraints, such as cache size.

6.6 Summary

In this chapter, we have presented a framework for predicting cross-loop reuse. The framework combines two existing tools: array section analysis and data-flow analysis. By using array sections, we can exploit the characteristics of the program's array access patterns without resorting to potentially costly procedure-wide dependence analysis. By using data-flow analysis, we can gracefully handle intra-procedural control flow. Our framework also provides a mechanism for taking into account a specific cache size when predicting reuse.

This work opens up the possibility of systematically applying cross-loop transformations to improve cache utilization, since it provides a means of cheaply predicting the profitability of loop fusion and particularly loop reversal. The results in this chapter demonstrate that for programs that exhibit cross-loop reuse, our analysis framework is able to detect this reuse and apply cross-loop transformations to exploit it. For programs running on uniprocessors, our results translate into modest (up to 10%) improvements in overall execution time. We would expect to see more significant gains for shared-memory multiprocessors, where small increases in second-level cache utilization can sometimes result in significant performance improvements.

Chapter 7

Software Prefetching for DSM Multiprocessors

7.1 Introduction

Shared-memory multiprocessors are an attractive platform for small- to medium-scale parallel programming, since they provide a convenient programming model based on a hardware-supported shared address space. Recent multiprocessor designs are starting to incorporate a distributed shared-memory (DSM) architecture, in which main memory is divided up into modules and distributed across all of the processors. The processors are then linked together with a high-speed interconnection network [2, 24, 61]. Figure 7.1 shows an abstracted view of a DSM machine. When memory is distributed in this fashion, a processor that accesses an item currently being stored in another processor's memory must wait for the data to be transferred over the interconnect. Because these remote memory accesses can take hundreds and even thousands of cycles, cache utilization is a critical component of overall performance for programs running on these machines. Techniques that improve cache utilization, such as software prefetching, can be very beneficial.

DSM architectures present a series of additional challenges for software prefetching. First, coherence misses (accesses to cache lines that have been invalidated due to remote writes) may add significantly to the total miss count, placing more demands on prefetching. Second, the time required to satisfy a cache miss varies much more widely on DSMs than on uniprocessors. Cache miss latencies can be influenced by the distribution of data among the various memory modules, the sharing patterns in the program, network contention, and variety of other factors. Existing software prefetching studies employ a uniform-distance prefetch scheduling policy: the compiler tries to issue all prefetches a fixed number of cycles in advance [72, 73, 94]. Such a policy is problematic on DSM machines, where memory latency is much less uniform. If all prefetches are scheduled assuming a worst-case memory latency, a significant fraction of the prefetched data may arrive too early, causing evictions that will effectively eliminate any benefits for long-latency misses.

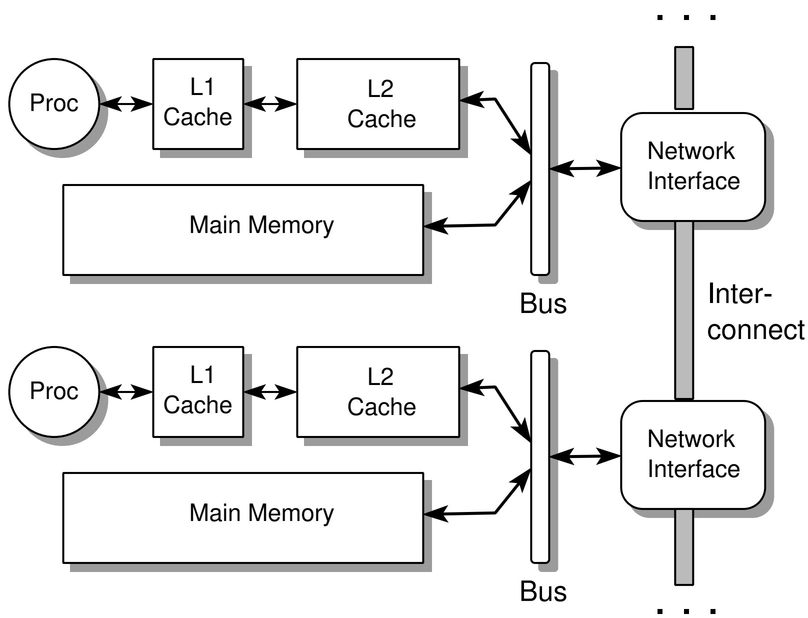


Figure 7.1 DSM multiprocessor block diagram

In this chapter we describe our work on software prefetching for DSM machines. In the first part of the chapter, we evaluate the performance of previously developed software prefetching strategies on DSM multiprocessors. Our results show that prefetching improves performance, but is not totally successful in hiding remote memory latencies. In particular, most programs exhibit a greatly increased percentage of late prefetches, relative to the levels present for the uniprocessor case, and the average penalty incurred on each late prefetch is severe. In the second part of the chapter, we outline a new form of compiler analysis that enables the prediction of one class of coherence misses, allowing the compiler to schedule prefetches to hide these long-latency misses.

An outline of the chapter is as follows. In Section 7.2 we describe DSM multiprocessors in more detail, emphasizing the architectural features that are related to prefetching. Section 7.3 presents information on the class of parallel programs targeted by our compiler, as well as giving a brief overview of the parallel code generation scheme it incorporates Section 7.4 gives the results of an experimental evaluation of software prefetching on DSM machines, for a set of parallel benchmark programs.

In Section 7.5 we describe our new compiler framework for optimizing prefetching on DSM machines, and in Section 7.6 we describe some of the optimizations that framework can be used to drive. Section 7.7 gives a set of experimental results on the effectiveness of the framework, for a small set of subroutines extracted from three of our benchmark programs. Finally, in Section 7.8, we summarize the material covered in this chapter.

7.2 DSM multiprocessors

The components of a typical DSM multiprocessor are illustrated in Figure 7.1. Each node in the DSM consists of a processor, a cache, a memory module, a local bus, and a network interface connecting the node with the rest of the machine. Pages of main memory are allocated to the processors according to some distribution function (example: round-robin, or first-touch). Each processor maintains a hardware directory for the pages it owns. The directory contains an entry for each cache line in a page, storing IDs of the remote processors that have cached copies of the line [65].

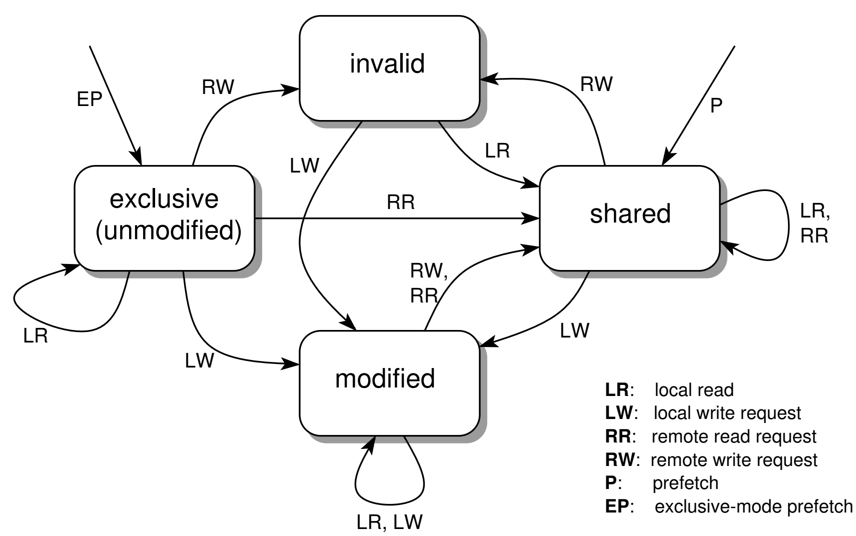


Figure 7.2 State diagram for example MESI cache coherence protocol

Invalidation-based coherence protocols

When a processor writes to a location that other processors may have cached, or when it reads a location that another processor may have recently written, the underlying hardware takes action to ensure that all processors see a consistent view of memory. In an invalidation-based cache coherence protocol, when a processor P writes to a location, the hardware brings a copy of the line containing the location into P's cache, invalidates any copies of the line cached by other processors, and places the line in a "modified" state. This state indicates that P is the only processor caching the line, and that the contents of the line are different from the contents stored in main memory. When a processor issues a read operation for a line not already in its local cache, the line is brought into the cache in a "shared" state. This state indicates that other processors may have a copy of the line, but the contents of all the copies are up to date with the values in main memory. Finally, a line in an "invalid" state indicates that the data in the line is stale, and should not be used.

Figure 7.2 shows an example coherence protocol state diagram. The labels on the edges between the various states indicate the action that causes the state transition, from the point of view of the local processor. This particular protocol also includes an "exclusive unmodified" state, which indicates that the caching processor has an exclusive copy of the line, but that the line's contents are still in sync with memory. This additional state can be useful for supporting "exclusive-mode prefetching", in which a line is prefetched into the cache in an exclusive but unmodified state [72]. This special type of prefetch is useful for read-modify-write accesses, where a processor reads a line and then immediately writes the same line. If the compiler issues an ordinary prefetch in such situations, then two remote requests will have to be performed: the first to fetch the line in a shared state, and the second to obtain exclusive ownership of the line for the write. If the compiler issues an exclusive-mode prefetch for the line in question, then only one remote request will be needed, reducing overall network traffic.

7.3 Parallel program model

Most researchers in the past have focused on applying prefetching to explicitly parallel programs, in which the compiler has no specific knowledge about parallelism within the target program, other than that the program is parallel at some level [72]. We have chosen instead to target a class of compiler-parallelized Fortran programs of the sort produced by KAP [59] or by an advanced workstation compiler. This type of parallel program consists of a sequential Fortran program that has been annotated with "DOALL" or "PDO" directives indicating the loops that can be safely be run in parallel, as well as the sets of variables that can be placed in processor-private storage. By targeting a class of programs in which information about parallelism is exposed to the compiler, and by integrating a parallel code generator into our implementation, we lay the foundation for the design of compiler techniques that exploit information about parallel loops during prefetching.

7.3.1 Parallel code generation

As part of the compilation process, our compiler uses the parallelization directives in the source code to transform the annotated sequential input program into a full-fledged shared-memory parallel program. We refer to this process as "parallel code generation". The table in Figure 7.3 shows the set of directives currently supported by our parallel code generator. Figure 7.4 shows an excerpt of an example parallel program that uses some of the directives.

The parallel code generator extracts the statements within a parallel region and relocates them into a separate subroutine, adding a call to a runtime thread-management routine in their place. The separate subroutine is then invoked on each processor at runtime. This process also provides a mechanism for implementing processor-private storage, since each parallel thread will have its own stack. Within the parallel subroutine, parallel loops are given reduced loop bounds, computed by a call to a loop scheduling function. Other directives such as barriers are converted into calls to simulator runtime routines.

Our current implementation requires that parallel loops use only barrier synchronization. We assume that the iterations of a given parallel loop are scheduled (assigned to processors) either statically by blocks, or through the use of an *affinity scheduler* [69]. Although our compiler does not currently incorporate data distribution directives (of the sort provided in High Performance Fortran [48]), our techniques could easily be extended to exploit information of this nature.

7.4 Evaluation of software prefetching for DSM machines

In this section we evaluate the performance of software prefetching on DSM machines for our benchmark suite. An outline of this section follows. Section 7.4.1 describes

Directive	Interpretation
C$PAR parallel	Marks the beginning of a parallel region within the program, containing code to be executed by all processors. This directive also lists the shared and processor-private variables used within the parallel region.
C$PAR end parallel	Marks the end of a parallel region.
C$PAR pdo	Marks the start of a partitioned loop: iterations of the loop are divided up among all processors.
C$PAR end pdo	Marks the end of a partitioned loop.
C$PAR barrier	Indicates a barrier synchronization point.
C$PAR single process	Marks the beginning of a "single-processor" section within a parallel region. The code within the section must only be executed by a single processor.
C$PAR end single process	Marks the end of a single-processor section.
C$PAR critical section	Marks the beginning of a critical section within a parallel region. Only one processor at a time is allowed to execute within the critical section.
C$PAR end critical section	Marks the end of a critical section.

Figure 7.3 Parallelization directives

```
C$PAR   parallel shared(a,b,n) local(i,j,tt)
C$PAR   pdo
        do j = 1, n
           tt = b(3,j)
           do i = 1, n
              ... tt + a(i,j) ...
           enddo
        enddo
C$PAR   end pdo
C$PAR   single process
        b(1,1) = n
C$PAR   end single process
C$PAR   pdo
        do j = 1, n
           b(2,j) = 0
        enddo
C$PAR   end pdo nowait
C$PAR   barrier
C$PAR   end parallel
```

Figure 7.4 Parallel program excerpt

how we produced the parallel programs used in the study, and provides some empirical data on the level of parallel coverage they exhibit. Section 7.4.3 covers the target architecture modeled by our simulator. In Sections 7.4.5, 7.4.6, and 7.4.7 we provide experimental data on prefetching performance, including overall execution time improvement, information about prefetch outcomes, and data on late prefetches.

7.4.1 Parallel benchmark programs

To obtain the parallel programs for this study, we selected a set of programs from our uniprocessor benchmark suite and created parallel versions of the programs by running them through KAP, the Kuck and Associates Preprocessor [59]. KAP, an automatic parallelization tool, reads a sequential Fortran program and produces an annotated version of the program that contains directives (comments) indicating loops and regions that can be safely executed in parallel. The version of KAP (V6.0.2) used for this study generates only a single level of parallelism (no nested "PDO" loops). Two uniprocessor programs in our benchmark suite, `cgm` and `nasbuk`, were excluded from this study, due to poor parallelization. KAP was unable to parallelize any of the important loops, because of the way the programs use indirection arrays.

Figure 7.5 shows the cache miss rates of these programs when run on the simulated DSM described in Section 7.4.3. Coherence activity results in miss rates for several of the parallel programs that are higher than the miss rates in the equivalent uniprocessor program (this is the case for `apsi`, `applu`, `appsp`, `flo52`, and `ocean`). The hit rates for the remainder of the programs are either roughly the same or show minor improvements, due to the relative increase in total cache available to the program.

Program	L1 miss rate	L2 miss rate
appbt	13.6%	19.2%
applu	23.5%	45.5%
appsp	10.7%	11.2%
apsi	19.9%	33.0%
erl	17.1%	26.2%
flo52	15.0%	14.7%
hydro2d	24.4%	25.6%
mgrid	6.3%	34.2%
ocean	20.8%	8.6%
su2cor	23.9%	9.2%
swim	74.9%	11.5%
tomcatv	22.5%	21.2%
wave5	36.7%	2.4%

Figure 7.5 Miss rates for parallelized benchmarks

7.4.2 Parallel coverage

To provide a sense for how much loop-level parallelism KAP has been able to identify, we experimentally measured the parallel coverage for our programs. The results are shown in Figure 7.6. These data were generated by comparing the base (untransformed) versions of the programs with the parallelized versions run on a single processor.

The first column shows the instruction overhead added during the parallelization process; this corresponds to the percentage increase in dynamic instruction count. Most of the programs have fairly reasonable overhead levels; the exceptions

Program	Parallelization overhead	Parallel coverage instructions	Parallel coverage cache accesses
appbt	1.6%	99.2%	99.5%
applu	29.8%	99.3%	99.4%
appsp	3.0%	98.9%	99.2%
apsi	14.6%	92.7%	92.6%
erl	-55.6%	99.8%	99.9%
flo52	0.5%	94.8%	96.1%
hydro2d	0.0%	76.5%	73.8%
mgrid	0.0%	95.2%	97.0%
ocean	1.0%	96.8%	97.2%
su2cor	0.6%	55.8%	62.3%
swim	0.0%	98.0%	98.7%
tomcatv	0.3%	16.2%	22.9%
wave5	3.0%	68.5%	78.8%

Figure 7.6 Parallel coverage

are "`applu`" and "`erl`". In the first case, the parallelizer selected an inner loop as the "DOALL" loop within a critical nest, greatly increasing the number of times the program enters and leaves the subroutine corresponding to a parallel region. In the case of "`erl`", the parallel version is actually faster, due to an artifact of the Sun `f77` compiler related to Fortran intrinsic functions. Note that the instruction overhead shown in Figure 7.6 does not include synchronization.

The second and third columns in Figure 7.6 refer to the parallelized version of the code; they show the percentage of all instructions and all cache accesses that are executed within a parallel region. Most programs have high parallel coverage percentages, but a few have problems. The programs "`hydr2od`" and "`tomcatv`" both read large data files as part of their initialization process; this accounts for their reduced parallel coverage. In the case of "`su2cor`" and "`wave5`", the extensive use of indirection arrays has interfered with a complete parallelization of the program.

7.4.3 Architectural parameters

We use execution-driven simulation to gather our data. Our simulated parallel machine consists of a series of RISC processors connected by a high speed interconnection network. We assume hardware-supported, directory-based cache coherence, with an invalidation protocol (shown in Figure 7.2). From the point of view of processor speed, cache configuration, and memory bandwidth, each node in the DSM machine is identical to that used in our uniprocessor simulations (see Figure 5.2).

Network parameters and remote cache miss latencies for the simulated architecture are shown in Figure 7.7. In the figure, a "2-hop" remote cache miss refers to the scenario where a processor requests a cache line from the owning processor, and the owning processor is able to return the data without any additional remote requests. In a "4-hop" remote cache miss, processor X requests a line from processor Y, but a third processor Z has the line in a modified or exclusive state, thus Y must first communicate with Z before returning the data to X.

The values we have chosen for remote memory access latencies are fairly high, relative to certain commercially available DSM multiprocessors. For example, average remote memory access latency for a 128-processor SGI Origin is quoted as 945 nanoseconds, and the ratio of local to remote memory access latency is characterized very roughly as 2:1 [63]. Other NUMA multiprocessors, such as the Sequent NUMAQ, have considerably local-to-remote ratios that are closer to the values we use in our experiments, with correspondingly higher remote memory access penalties [68].

Our simulated DSM uses release consistency [32]. Pages of main memory are allocated to the processors in a round-robin fashion by default; page size is 4 Kbytes.

Network parameters	
Topology	ring
Peak network bandwidth	4 GBits/sec per processor
Remote miss latency	608 cycles (2-hop) or 1216 cycles (4-hop)

Figure 7.7 Network parameters for DSM multiprocessor simulations

7.4.4 Compiler parameters

For these experiments, the prefetching distance for pipelined loops was calculated based on an assumed memory latency of 350 cycles. This value was arrived at experimentally by simulating our various benchmark programs at a range of compile-time latency settings and choosing the single latency that provided the best performance for all programs.

Our implementation does not currently support exclusive-mode prefetching; all lines are fetched initially in a read-shared state. As in the uniprocessor case, the compiler inserts prefetches only for read references, since the simulated architecture incorporates a write-through cache and a write buffer.

7.4.5 Performance of prefetching on DSM machines

Program	Reduction in execution time
`appbt`	26.6%
`applu`	15.4%
`appsp`	22.8%
`apsi`	27.3%
`erl`	26.1%
`flo52`	26.5%
`hydro2d`	40.3%
`mgrid`	34.9%
`ocean`	14.0%
`su2cor`	31.5%
`swim`	25.4%
`tomcatv`	47.5%
`wave5`	20.6%

Figure 7.8 Execution time reduction due to prefetching, 4 processors

Figure 7.8 shows the reduction in execution time provided by software prefetching for 4 processors, comparing the execution time of the original parallel program with that of the parallel program with prefetching. In general, the programs for which prefetching worked well on a single processor also benefit from prefetching in the multiprocessor case. Such programs include "`hydro2d`", "`mgrid`", and "`tomcatv`", among others. In addition, some of the programs that derived little benefit from prefetching on a uniprocessor show more pronounced gains when they are parallelized. For example, prefetching actually increased the execution time of the single-processor version of "`flo52`" by 6.2%, whereas the parallel version of the same program is 26.5% faster when prefetching is applied.

	Used	Overshoot	Replaced by Pref	Invalidated	Hit Cache	Hit Buffer	Full Buffer
appbt	27.2%	1.5%	1.0%	0.4%	59.5%	11.4%	0.2%
applu	35.6%	5.1%	4.6%	2.7%	47.2%	9.3%	0.0%
appsp	43.5%	1.9%	1.7%	0.4%	49.6%	4.7%	1.3%
apsi	32.1%	10.4%	6.7%	2.0%	40.0%	15.4%	4.3%
erl	83.9%	0.7%	0.7%	0.0%	10.3%	5.1%	2.4%
flo52	20.8%	5.4%	4.1%	0.8%	63.8%	9.2%	0.6%
hydro2d	72.7%	1.0%	0.5%	0.0%	25.5%	0.9%	10.5%
mgrid	73.7%	0.6%	0.5%	0.0%	18.5%	7.3%	2.1%
ocean	17.8%	0.7%	0.6%	0.1%	65.7%	15.7%	0.1%
su2cor	26.0%	10.6%	5.6%	0.1%	57.2%	6.1%	6.1%
swim	15.1%	76.7%	46.4%	0.0%	7.2%	1.0%	1.9%
tomcatv	41.1%	48.1%	27.6%	0.0%	9.9%	0.9%	28.4%
wave5	12.9%	22.2%	14.2%	0.3%	58.9%	5.7%	4.0%

Figure 7.9 Outcome breakdown for prefetches, 4 processors

7.4.6 Outcomes for prefetches

Figure 7.9 shows the outcome for each of the prefetches issued during the simulations. “Used” refers to a prefetch that partially or completely succeeds in hiding the latency of a cache miss. “Overshoot” indicates that a prefetched line arrived in the cache, but was displaced before it could be used. “Replaced by Pref” is a sub-case of “Overshoot”, where a prefetched line X is displaced by another arriving prefetched line Y (as opposed to a line brought in by a cache miss) before X can be used. “Invalidated” indicates that a prefetch arrived in the cache, but was invalidated before it could be used. “Hit cache” and “Hit buffer” are prefetches that hit in the L1 cache and in the prefetch buffer, respectively. “Full buffer” refers to a scenario where the processor tries to issue a prefetch, but the pending prefetch is full.

We interpret this data as follows. First, the percentage of prefetches that hit in the cache seem to match the single-processor results, for the most part. Similarly, some of the programs still have problems with cache conflicts, notably “`swim`” and “`tomcatv`”. Invalidation of prefetched lines does not appear to be a major problem for any of the programs, suggesting that there is relatively little false sharing behavior. The one area where there is a striking difference between uniprocessor and multiprocessor prefetching performance is the frequency of late prefetches, which we discuss in the following section.

7.4.7 Prefetch scheduling: a closer look

Program	Percent late prefetches	Cycles Late (average)	Percent dirty remote	Percent clean remote
appbt	23.5%	332	15.8%	77.4%
applu	34.5%	493	34.7%	55.1%
appsp	8.8%	598	61.0%	37.4%
apsi	34.2%	787	81.8%	17.4%
erl	31.1%	361	0.2%	96.4%
flo52	17.1%	726	92.4%	6.9%
hydro2d	21.6%	418	5.7%	91.0%
mgrid	6.8%	502	27.7%	70.5%
ocean	22.2%	433	72.1%	19.9%
su2cor	3.8%	636	35.4%	63.3%
swim	20.2%	510	1.3%	98.4%
tomcatv	23.3%	593	12.5%	87.3%
wave5	13.5%	704	85.3%	14.0%

Figure 7.10 Late prefetches as a percentage of useful prefetches, 4 processors

In this section we take a closer look at the fraction of prefetches that are late, and the effects of late prefetches on execution time. Figure 7.10 shows the percentage of the useful prefetches (prefetches that did not hit in cache or in the pending prefetch buffer) that were late. "Cycles Late" refers to the average number of cycles the processor had to stall waiting for the prefetch to complete. The data show that in comparison to our uniprocessor study, late prefetches are much more prevalent, and the average penalty for a late prefetch is approximately an order of magnitude larger. This suggests that compiler optimizations to eliminate late prefetches can potentially have a major impact on performance.

The last two columns are expressed as percentages of all *late* prefetches. A late "clean remote" prefetch refers to a scenario in which a prefetch results in a line being obtained from another processor where the line is in a clean state (*i.e.*, no processor is currently caching the line in an exclusive state). This type of late prefetch results simply from the distribution of memory among the processors. In our model, pages are assigned to processors at run-time, thus the compiler is unable to exploit information about memory distribution at compile time that might allow it to target this class of late prefetches.

A late "dirty remote" prefetch occurs when a line must be fetched from a remote processor, and the line must also be invalidated (some processor has the line in an exclusive state). A significant fraction of the late prefetches fall into this category for our benchmark programs. Unlike late "clean remote" prefetches, late "dirty remote" prefetches are a result of coherence activity, and are originally created by the program's sharing patterns, which can be analyzed at compile-time. This type of late prefetch can arise when a cache line is written on one processor and then subsequently read on another processor, for example. In the following sections, we develop a framework for statically detecting sharing patterns that are likely to result in late dirty remote prefetches.

7.5 Compiler framework

7.5.1 Predicting coherence misses: an overview

We have developed a global data-flow analysis method that predicts coherence activity within a parallel program. We give a brief outline of the procedure here and then describe each component in detail in subsequent sections of the chapter. Our compiler uses array-section analysis to identify portions of shared arrays accessed in specific program regions, typically loop nests. We augment array sections with a *parallel mapping* component that describes how a region within an array is accessed by the processors in the machine. We develop a set of equations that predicts the outcome (cache hit, cache miss, or remote miss) for specific array references within the program. Our compiler solves these equations using interval-based data-flow analysis. The solutions to the equations for a given control-flow graph node are then used to predict whether particular references within the node will access data that resides in cache, and if not, whether retrieving the data will require coherence activity. To account for cache size constraints, we enhance the analysis by incorporating an "age" function on array sections that approximates the number of capacity misses since the section was cached. The data-flow machinery updates the ages of each section during the analysis; when a section's age reaches a machine-dependent cutoff point, it is eliminated from the set that contains it.

7.5.2 Array-section analysis

Most scientific Fortran programs spend the bulk of their execution time performing computations on arrays in loops; in order to characterize the memory usage patterns for these programs, the compiler must analyze how arrays are accessed within loops. In our framework, we capture information on array access patterns using *array-section analysis* [12, 15, 45, 67]. When applied to a portion of the program (typically a basic block, loop, or loop nest), array-section analysis produces a summary representation of the region accessed within each array.

Each of the summaries (referred to hereafter as "sections") contains a component that describes the geometric region accessed within the array, in addition to other bookkeeping data and information about the order in which array dimensions are traversed. In order to be useful for our work, the sections must also capture information on how the accesses to a given array region are distributed among the available processors at run-time. For example, in Figure 7.11, the "`do` k_3" loop nest and the "`do` j_2" loop nest both access the region `a(1:100,1:100)`. In the first case, however, each processor accesses a block of columns, whereas in the second case, each processor accesses a block of rows [recall that for each "doall" loop, a single contiguous block of iterations is assigned to each processor].

To distinguish between these cases, we augment each section with a *parallel mapping* component, or PMAP, that contains information on how the elements within the section are accessed by the available processors at run time. The PMAP is a restricted version of the "mapping function descriptor" used in the *Available Section Descriptor* abstraction [40].

We assume an unbounded virtual processor grid with N dimensions (note that since our program model currently permits only a single level of parallelism, N is currently 1). For a given section S_1, the mapping function is of the form $\langle P, F \rangle$, where P and F are vectors of length N. Element i within P (denoted P_i) is the dimension of the array mapped to grid dimension i, and element i within F (denoted F_i) is a mapping function that gives the position(s) along the processor grid that array elements are mapped to. The mapping function is of the form

$$F_i(j) = (c * j + l : c * j + u : s)$$

```
      do k1 = 1, n
C$PAR   parallel shared(a,b,d,k1) local(k2,k3,k4)
C$PAR   pdo
        do k2 = 1, 100
          do k3 = 1, 100
            ... = ... a(k3,k2) +
                b(k3,k2,k1) + d(k3)
          enddo
        enddo
C$PAR   end pdo
C$PAR   pdo
        do k4 = 1, 50
          do k5 = 1, 50
            ... = ... b(k5,k4,k1)
          enddo
        enddo
C$PAR   end pdo
C$PAR   end parallel
      enddo
```

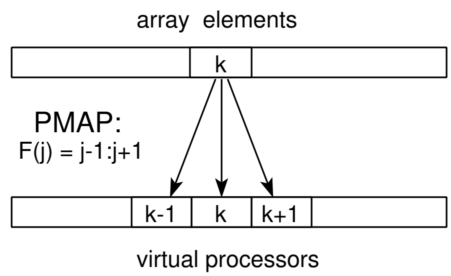
```
        do i1 = 1, 100
          ... = ... c(i1,2)
        enddo

C$PAR   parallel shared(a) local(j1,j2,j3)
C$PAR   pdo
        do j1 = 1, 100
          do j2 = 1, 100
            ... = ... a(j1,j2)
          enddo
          do j3 = 1, 50
            ... = ... a(j3,j1)
          enddo
        enddo
C$PAR   end pdo
C$PAR   end parallel
```

Section	loop	REGION	PMAP
S_1	do k_3	a(1:100,1:100)	$\langle 2, F(j) = j : j : 1 \rangle$
S_2	do k_3	b(1:100,1:100,1:n)	$\langle 2, F(j) = j : j : 1 \rangle$
S_3	do k_3	d(1:100)	$\langle 0, \top \rangle$
S_4	do k_5	b(1:100,1:50,1:n)	$\langle 2, F(j) = j : j : 1 \rangle$
S_5	do i_1	c(1:100,2)	$\langle 0, \bot \rangle$
S_6	do j_2	a(1:100,1:100)	$\langle 1, F(j) = j : j : 1 \rangle$
S_7	do j_3	a(1:50,1:100)	$\langle 2, F(j) = j : j : 1 \rangle$

Figure 7.11 Example loop nests with section information

where l and u are invariants, s (stride) is an integer constant, and c is either 1 or 0.[7] This triplet-style formulation allows one-to-one mappings (when $l = u$), one-to-many mappings (when $u \geq l + s$) and constant mappings (when $c = 0$). The following picture illustrates the effects of an example mapping function:



There are two special-case mapping functions as well. We use the mapping function $\langle 0, \perp \rangle$ to indicate accesses that take place outside parallel loops, and the mapping function $\langle 0, \top \rangle$ to indicate that that the section in question is accessed by all processors.

The table appearing at the bottom of Figure 7.11 shows each of the regions accessed within the loop nests, along with their parallel mapping functions.

7.5.3 Predicting coherence misses

Figure 7.12 shows a program fragment containing a parallel loop, along with its corresponding control-flow graph (CFG). We would like to predict whether some or all of the array accesses in the loop will result in coherence misses. This will depend on the access patterns within and prior to the loop, *i.e.*, node X and its predecessors. Suppose that a loop prior to node X writes to a section S_X in the array "`b`", and that the `doall j` loop accesses a section S_j in array "`b`". When the `doall j` loop executes, there will be coherence activity if regions of the two sections overlap, but the common elements are cached on different processors (we will formalize this notion shortly, with the definition of *conformability* of array sections). In order to predict coherence activity, therefore, the compiler must symbolically compare the regions of the arrays accessed within a given loop with the sections that are likely to be in cache on entry to the loop.

[7] In the mapping function that appears in Gupta and Schonberg's *ASD*, c can take on a rational value, provided that the function evaluates to a range over integers. This is more general than is needed for our work.

Cache size is an important factor. Even though there may have been accesses to the array "b" in a prior portion of the program's execution, subsequent cache activity may have flushed these elements from the caches of the processors. If none of the array elements accessed within the `doall j` loop are resident in any processor's cache on entry to the loop, then there will be no coherence activity within the loop. For simplicity, we initially ignore cache size constraints in our analysis. Section 7.5.6 describes the mechanism that we use to factor in a specific cache size.

```
       ...X...
C$PAR  parallel shared(a,b,n)
C$PAR&       local(i,j)
C$PAR  pdo
       do j = 1, n
          do i = 1, n
             a(i,j) = ... b(i,j) ...
             ...
          enddo
       enddo
C$PAR  end pdo
C$PAR  end parallel
       ...Y...
```

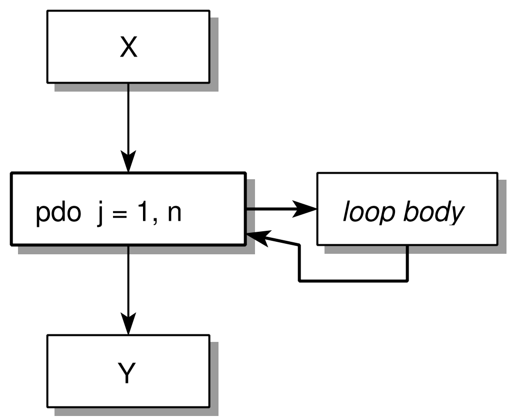


Figure 7.12 Parallel loop with corresponding CFG

Conformability of sections

We say that section S_1 *conforms* to section S_2 if and only if all of the following conditions are met:

1. $\textsc{region}(S_1)$ contains $\textsc{region}(S_2)$, or $\textsc{region}(S_2)$ contains $\textsc{region}(S_1)$
2. $\textsc{pmap}(S_1) = \textsc{pmap}(S_2)$
3. if $\textsc{pmap}(S_1) = \langle N, F\rangle$ where $N > 0$, then the bounds on dimension N in $\textsc{region}(S_1)$ are identical to the bounds on dimension N in $\textsc{region}(S_2)$

Intuitively, one section conforms to another if the same data is accessed on the same set of processors. For example, in Figure 7.11, section S_1 conforms to S_7, since these two sections satisfy all of the conditions above. However S_1 does not conform to S_6, since $\text{PMAP}(S_1) \neq \text{PMAP}(S_6)$. Similarly, S_2 does not conform to S_4, since the third condition above is not met.

For the purposes of this analysis, we relax the definition of containment to include situations where there isn't strict geometric containment, but the two regions substantially overlap, *i.e.*, the boundaries match except for a small constant, or one region is a slightly shifted copy of another region. For example, we would treat a section S_1 = `a(1:1000,2:999)` as containing the section S_2 = `a(1:1000,1:998)`, and vice versa.

We use the following notation in the remainder of the chapter:

notation	interpretation
X $\approx$ Y	X conforms to Y
X $\not\approx$ Y	X does not conform to Y
X $\sqsupseteq$ Y	X contains Y

Read outcomes

Figure 7.13 shows the possible outcomes for a load (read) of shared data on a DSM multiprocessor, from the perspective of the processor issuing the load. There are three possible scenarios; the particular outcome depends on the state of the local cache and the caches of the other processors. For a given scenario, the latency incurred by the read is given in terms of M, the main memory latency, and N, the network latency (time required to send a message over the interconnect). For example, in scenario "B", the processor issuing the read is accessing a location that is currently not cached by any processor. In this scenario, the issuing processor sends a request message to the processor whose memory contains the location in question, which then sends a message back with the data. The scenarios shown in Figure 7.13 are worst-case latencies (we assume that the data being accessed is owned by a remote processor, not the local processor).

Figure 7.14 gives a set of conditions sufficient to predict each of the three scenarios for a read operation. We construct these conditions in terms of array sections, using the notion of section conformability. In the figure, X refers to the CFG node containing the read we want to analyze. Note that for the remainder of this chapter,

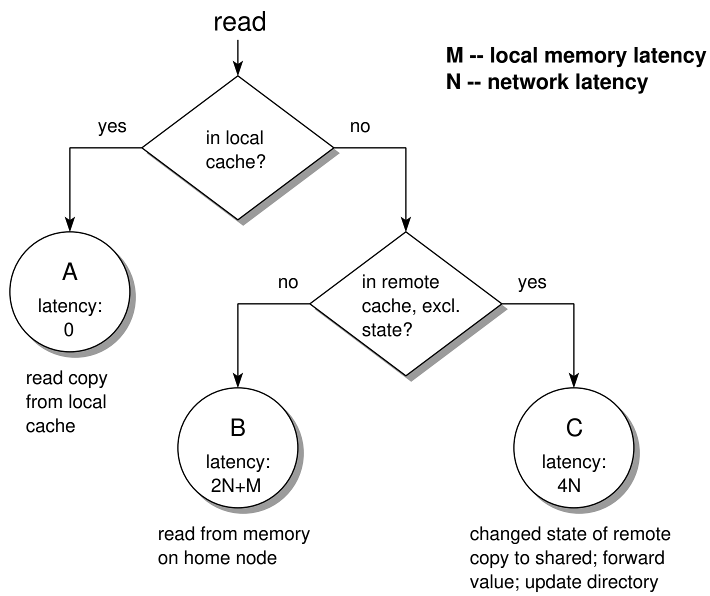


Figure 7.13 Possible outcomes for a cache-coherent read (worst-case)

Scenario	Conditions	
A	1	a section S_Y^{rw} reaches node X from some predecessor Y, and
	2	$S_Y^{rw} \sqsupseteq S_X^r$ and $S_Y^{rw} \approx S_X^r$
B	1	No S_Y^w (s.t. $S_Y^w \sqsupseteq S_X^r$) reaches X for any Y
C	1	a section S_Y^w reaches X from some predecessor Y, and
	2	$S_Y^w \sqsupseteq S_X^r$ and $S_Y^w \not\approx S_X^r$

Figure 7.14 Conditions for predicting read outcomes

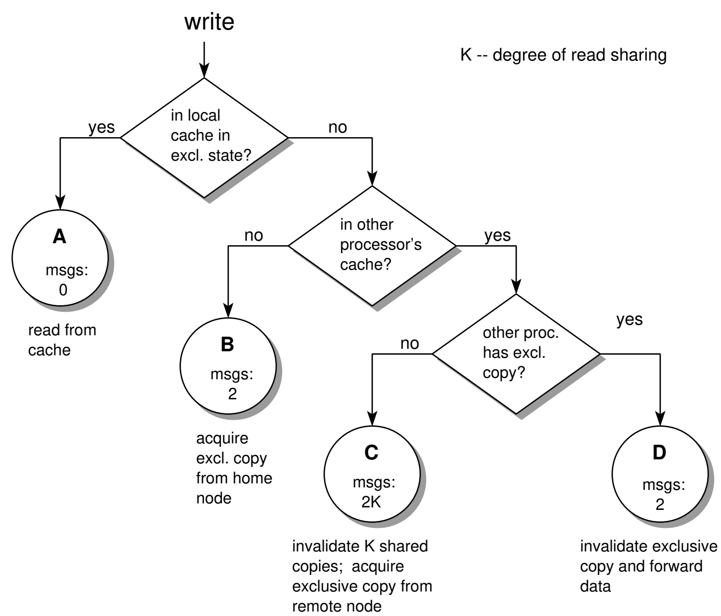


Figure 7.15 Possible outcomes of cache-coherent write (worst-case)

Scenario	Conditions	
A	1	a section S^w_Y reaches X from a predecessor Y, where $S^w_Y \approx S^w_X$ and $S^w_Y \sqsupseteq S^w_X$, and
	2	there is no intervening section S^r_Y such that $S^r_Y \sqsupseteq S^w_X$, and
	3	there is no intervening section S^w_Z such that $S^w_Z \not\approx S^w_X$ but $(S^w_Z \cap S^w_X) \neq \emptyset$
B	1	No section S^{rw}_Y reaches X
C	1	a section S^r_Y reaches X, where $S^r_Y \sqsupseteq S^w_X$, and
	2	there is no intervening section S^w_Z such that $(S^w_Z \cap S^r_Y) \neq \emptyset$
D	1	a section S^w_Y reaches X, where $S^w_Y \sqsupseteq S^w_X$ but $S^w_Y \not\approx S^w_X$, and
	2	there is no intervening section S^r_Z such that $(S^r_Z \cap S^w_Y) \neq \emptyset$

Figure 7.16 Conditions for predicting write outcomes

the term "node" will be used to refer to a CFG node, not to a processor within the DSM machine. The superscript notation for array sections specifies the type of access: S^w corresponds to a write, S^r corresponds to a read, and S^{rw} is either a read or a write. The subscripts for sections indicate the CFG node where the access takes place; S^r_X is the read operation at node X whose outcome we want to predict. In the figure, the term "a section S reaches X from Y" should be interpreted as follows: a region S of an array is accessed in node Y, and this section is still in the cache(s) of the accessing processor(s) when node X is subsequently executed. In the cases where there are multiple conditions for a given outcome, they should be interpreted as being "and'ed" together to form the condition for the outcome.

Write outcomes

Figure 7.15 shows the possible outcomes for a write (store) to shared data; the section being written is S^w_X. For writes, the cost in terms of network traffic of some of the outcomes is no longer fixed: the number of messages needed depends on the degree of sharing prior to the execution of the loop. In particular, if a processor tries to write a cache line that is currently read-shared by K processors, then K messages will need to be sent to invalidate all of the outstanding copies. The actual latency incurred will depend on the type of write buffer used in the processor, and on the degree to which the K invalidation and acknowledgement messages can be processed in parallel. Figure 7.16 gives a set of conditions sufficient to predict each of the scenarios for a write operation. As before, all of the numbered conditions for a given outcome must be satisfied.

7.5.4 Interval analysis

In this section we describe how we formalize the conditions presented in Figures 7.14 and 7.16 through the use of data-flow analysis. The data-flow framework we employ for this task is based on *interval* analysis. We refer the reader to previous works for a complete description of the terminology and mechanics of this form of data-flow analysis [4, 38, 39, 43, 53]. Interval analysis proceeds in two steps: an "interval contraction" phase, followed by an "interval expansion" phase. In the contraction phase, intervals are processed from innermost to outermost; an interval is only processed after all the intervals it contains are completed. We solve a set of equations for the nodes in the current interval, and then the nodes in the interval are summarized and collapsed into a single node. The contraction phase proceeds until all intervals have been contracted; the final graph (a DAG) is then analyzed as if it were an interval. In the expansion phase, the process is reversed: summary nodes are expanded into their original intervals and then re-analyzed.

Our interval analysis framework was inspired by that of Granston and Veidenbaum [38], which in turn was based on the framework of Gross and Steenkiste [39]. It differs from the work of Granston and Veidenbaum in several important respects. Their work was geared towards a multiprocessor without hardware support for cache coherence, whereas we specifically target machines with coherent caches in our data-flow framework. Since their techniques were designed for software-controlled local memories, they did not develop any mechanism for taking into account cache replacement effects.

The data-flow analysis we use is also related to the compiler techniques for optimizing communication placement on distributed-memory multiprocessors developed by Gupta, Schonberg, and Srinivasan [41]. Communications placement is a fairly different problem, however and analyzes the flow of *values* within the program, whereas our analysis focuses on the use of *locations*. Their framework also does not handle resource constraints, such as cache size, or architectural features that support cache coherence.

As with the work of Gross and Steenkiste [39] and of Granston and Veidenbaum [38], we present two sets of equations, the first for computing information within an interval, and the second for collapsing the nodes in an interval into a single summary node.

7.5.5 Data-flow equations

Figure 7.17 gives the names and definitions for the sets of sections used in our flow analysis. For a given CFG node n corresponding to a particular basic block, the compiler computes initial values of "UREF(n)" and "CREF(n)" by simply inspecting n. These initial sets are used as the inputs to the framework. The remainder of the variables are computed during the flow analysis. In the equations, we parameterize set names according to access type ("r" and "w" superscript notation indicates read and written sections, respectively; an "rw" superscript indicates that the section is either read or written). During the analysis we maintain the invariant that for every

Set name	Definition	Remarks
UREF(n)	the sections accessed within node n that unconditionally downwardly reach the end of node n.	computed for each basic block as part of the initial information.
UIN(n)	the sections accessed by some predecessor of node n that unconditionally downwardly reach the start of node n	computed during data-flow propagation
UOUT(n)	the sections accessed within node n or node n's predecessors that unconditionally downwardly reach the end of node n	computed during data-flow propagation
CREF(n)	the sections accessed in node n that conditionally downwardly reach the end of n, *i.e.*, sections that *may* be accessed in n	computed for each basic block as part of the initial information.
CIN(n)	the sections accessed by some predecessor of node n that conditionally downwardly reach the start of node n	computed during data-flow propagation
COUT(n)	the sections accessed within node n or node n's predecessors that conditionally downwardly reach the end of node n	computed during data-flow propagation

Figure 7.17 Data-flow sets

node n, $\mathrm{UREF}^w(n) \cap \mathrm{UREF}^r(n) = \emptyset$; if a region of an array is both read and written within a node, then we place the region in $\mathrm{UREF}^w(n)$ and not in $\mathrm{UREF}^r(n)$.

Figure 7.18 and 7.19 show the data-flow equations that we solve to obtain information about coherence activity. The equations in Figure 7.18 are computed for each of the nodes within an interval. When processing the interval, we visit the nodes in the interval in reverse postorder; for each node, we compute the "*IN*" sets and then the "*OUT*" sets. All sets are initially empty.

$$\mathrm{UIN}^r(\mathrm{n}) = \bigcap_{p \in PRED(n)} (\ \mathrm{UOUT}^r(\mathrm{p})\ ) \tag{7.1}$$

$$\mathrm{UIN}^w(\mathrm{n}) = \bigcap_{p \in PRED(n)} (\ \mathrm{UOUT}^w(\mathrm{p})\ ) \tag{7.2}$$

$$\mathrm{UOUT}^r(\mathrm{n}) = (\ \mathrm{UIN}^r(\mathrm{n}) \cup (\ \mathrm{UREF}^r(\mathrm{n}) -_c \mathrm{UIN}^w(\mathrm{n})\ )\ ) - \mathrm{CREF}^w(\mathrm{n}) \tag{7.3}$$

$$\mathrm{UOUT}^w(\mathrm{n}) = (\ \mathrm{UIN}^w(\mathrm{n}) \cup \mathrm{UREF}^w(\mathrm{n})\ ) -_{nc} \mathrm{CREF}^{rw}(\mathrm{n}) \tag{7.4}$$

$$\mathrm{CIN}^r(\mathrm{n}) = \bigcup_{p \in PRED(n)} (\ \mathrm{COUT}^r(\mathrm{p})\ ) \tag{7.5}$$

$$\mathrm{CIN}^w(\mathrm{n}) = \bigcup_{p \in PRED(n)} (\ \mathrm{COUT}^w(\mathrm{p})\ ) \tag{7.6}$$

$$\mathrm{COUT}^r(\mathrm{n}) = (\ \mathrm{CIN}^r(\mathrm{n}) \cup (\ \mathrm{CREF}^r(\mathrm{n}) - \mathrm{UIN}^w(\mathrm{n})\ )\ ) - \mathrm{UREF}^w(\mathrm{n}) \tag{7.7}$$

$$\mathrm{COUT}^w(\mathrm{n}) = (\ \mathrm{CIN}^w(\mathrm{n}) \cup \mathrm{CREF}^w(\mathrm{n})\ ) -_{nc} \mathrm{UREF}^r(\mathrm{n}) \tag{7.8}$$

Figure 7.18 Data-flow equations computed within an interval

Note the use of the "$-_{nc}$" operator in equations 7.4 and 7.8. Intuitively, $X -_{nc} Y$ subtracts sections in Y from X, but removes only those sections that do *not* conform to the sections in X. In other words, if there is some section $S_1 \in X$ and $S_2 \in Y$ such that $(S_1 \cap S_2) \neq \emptyset$, but $S_1 \approx S_2$, then S_1 will not be removed from X when forming $X -_{nc} Y$. Similarly, the "$-_c$" operator removes only the sections that do conform.

$$\mathrm{UREF}^r(\mathrm{S}) = \biguplus_{loop} (\ \mathrm{UOUT}^r(\mathrm{E})\ ) - \biguplus_{loop} (\ \mathrm{COUT}^w(\mathrm{E})\ ) \tag{7.9}$$

$$\mathrm{UREF}^w(\mathrm{S}) = \biguplus_{loop} (\ \mathrm{UOUT}^w(\mathrm{E})\ ) -_{nc} \biguplus_{loop} (\ \mathrm{COUT}^{rw}(\mathrm{E})\ ) \tag{7.10}$$

$$\mathrm{CREF}^r(\mathrm{S}) = \biguplus_{loop} (\ \mathrm{COUT}^r(\mathrm{E})\ ) - \biguplus_{loop} (\ \mathrm{UOUT}^w(\mathrm{E})\ ) \tag{7.11}$$

$$\mathrm{CREF}^w(\mathrm{S}) = \biguplus_{loop} (\ \mathrm{COUT}^w(\mathrm{E})\ ) -_{nc} \biguplus_{loop} (\ \mathrm{UOUT}^{rw}(\mathrm{E})\ ) \tag{7.12}$$

Figure 7.19 Data-flow equations for interval summarization

If a section $S \in \mathrm{UIN}^r(n)$, then this means that the cache lines containing S will be in a shared state (see Figure 7.2) on entry to node n, whereas if $S \in \mathrm{UIN}^w(n)$, the lines containing S will be in a modified state.

The "$\mathrm{UREF}^r(\mathrm{n}) - \mathrm{UIN}^w(\mathrm{n})$" term in equation 7.3 seems counterintuitive, but it is necessary due to the way the coherence protocol works. If a previously written line is already in the cache in a modified state when a read takes place, the line will remain in the modified state (thus a section S already in $\mathrm{UIN}^w(\mathrm{n})$ should not be added to $\mathrm{UOUT}^r(\mathrm{n})$, even if $S \in \mathrm{UREF}^r(\mathrm{n})$).

Figure 7.19 shows the data-flow equations used to summarize an interval into a single representative node. On the left hand side of these equations, the node S refers to the summary node being created; on the right hand side of the equations, node E refers to the exit node of the interval being summarized. The $\uplus$ operator is defined as "loop translation"; when applied to a section within a given loop, it substitutes the bounds of the loop for the loop induction variable [12]. For example, in Figure 7.11, the section accessed on a given iteration of the "`doall` j_1" loop is `a(`j_1`,1:100)`. Applying the operation $\uplus_{j_1}$ to this section will produce the section `a(1:100,1:100)`.

Figures 7.20 and 7.21 show how the results of the flow analysis correspond to the scenarios in Figures 7.13 and 7.15; they are analogous to Figures 7.14 and 7.16, except that the conditions have been rewritten in terms of the sets generated by the data-flow framework.

Scenario	Conditions
A	$\exists\ S_Y^{rw}$ in UIN^w(X) or UIN^r(X) such that $S_Y^{rw} \sqsupseteq S_X^r$ and $S_Y^{rw} \approx S_X^r$
B	No S_Y^{rw} in UIN^r(X) or in UIN^w(X) such that $S_Y^{rw} \sqsupseteq S_X^r$
C	$\exists\ S_Y^w$ in UIN^w(X) such that $S_Y^w \sqsupseteq S_X^r$ but $S_Y^w \not\approx S_X^r$

Figure 7.20 Read outcomes based on data-flow sets

Scenario	Conditions
A	$\exists\ S_Y^w$ in UIN^w(X) such that $S_Y^w \approx S_X^w$ and $S_Y^w \sqsupseteq S_X^w$
B	No S_Y^{rw} in UIN^r(X) or in UIN^w(X) such that $S_Y^{rw} \sqsupseteq S_X^w$
C	$\exists\ S_Y^r$ in UIN^r(X) such that $S_Y^r \approx S_X^w$ and $S_Y^r \sqsupseteq S_X^r$
D	$\exists\ S_Y^w$ in UIN^w(X) such that $S_Y^w \sqsupseteq S_X^w$ but $S_Y^w \not\approx S_X^w$

Figure 7.21 Write outcomes based on data-flow sets

7.5.6 Incorporating cache size constraints

The data-flow framework we have described thus far does not take processor cache size into account; in order for the analysis to generate useful information, it must model capacity effects. For example, if the set UREF$^r(n)$ for a given node n contains a section S_X^r, this tells us that the section in S was accessed some time in the past, but not whether the section is still resident in cache on entry to node n.

We use the enhancements developed in Chapter 6 to model cache capacity effects, including the notion of array section age and the modifications to the two data-flow operators (see Section 6.2.6).

Data-flow framework modifications

The equations in 7.22 compute the same information as those in Figure 7.18, but for a cache of a particular size. The changes are as follows. First, when computing UIN$^r(n)$ and CIN$^r(n)$, we use the $\vee$ and $\wedge$ operators in place of $\cup$ and $\cap$. $\vee$ and $\wedge$ still form the intersection and union of their arguments, however when a given section S

$$\mathrm{UIN}^r(\mathrm{n}) = \bigwedge_{p \in PRED(n)} (\ \mathrm{UOUT}^r(\mathrm{p})\ ) \tag{7.13}$$

$$\mathrm{UIN}^w(\mathrm{n}) = \bigwedge_{p \in PRED(n)} (\ \mathrm{UOUT}^w(\mathrm{p})\ ) \tag{7.14}$$

$$\mathrm{UIN}^r_\Delta(\mathrm{n}) = \mathrm{UIN}^r(\mathrm{n}) \odot (\ \mathrm{UREF}^r(\mathrm{n}) \cup \mathrm{UREF}^w(\mathrm{n})\ ) \tag{7.15}$$

$$\mathrm{UIN}^w_\Delta(\mathrm{n}) = \mathrm{UIN}^w(\mathrm{n}) \odot (\ \mathrm{UREF}^r(\mathrm{n}) \cup \mathrm{UREF}^w(\mathrm{n})\ ) \tag{7.16}$$

$$\mathrm{UOUT}^r(\mathrm{n}) = (\ \mathrm{UIN}^r_\Delta(\mathrm{n}) \cup (\ \mathrm{UREF}^r(\mathrm{n}) - \mathrm{UIN}^w(\mathrm{n})\ )\ ) - \mathrm{CREF}^w(\mathrm{n}) \tag{7.17}$$

$$\mathrm{UOUT}^w(\mathrm{n}) = (\ \mathrm{UIN}^w_\Delta(\mathrm{n}) \cup \mathrm{UREF}^w(\mathrm{n})\ ) -_{nc} \mathrm{CREF}^{rw}(\mathrm{n}) \tag{7.18}$$

$$\mathrm{CIN}^r(\mathrm{n}) = \bigvee_{p \in PRED(n)} (\ \mathrm{COUT}^r(\mathrm{p})\ ) \tag{7.19}$$

$$\mathrm{CIN}^w(\mathrm{n}) = \bigvee_{p \in PRED(n)} (\ \mathrm{COUT}^w(\mathrm{p})\ ) \tag{7.20}$$

$$\mathrm{CIN}^r_\Delta(\mathrm{n}) = \mathrm{CIN}^r(\mathrm{n}) \odot (\ \mathrm{CREF}^r(\mathrm{n}) \cup \mathrm{CREF}^w(\mathrm{n})\ ) \tag{7.21}$$

$$\mathrm{CIN}^w_\Delta(\mathrm{n}) = \mathrm{CIN}^w(\mathrm{n}) \odot (\ \mathrm{CREF}^r(\mathrm{n}) \cup \mathrm{CREF}^w(\mathrm{n})\ ) \tag{7.22}$$

$$\mathrm{COUT}^r(\mathrm{n}) = (\ \mathrm{CIN}^r_\Delta(\mathrm{n}) \cup \mathrm{CREF}^r(\mathrm{n})\ ) - \mathrm{UREF}^w(\mathrm{n}) \tag{7.23}$$

$$\mathrm{COUT}^w(\mathrm{n}) = (\ \mathrm{CIN}^w_\Delta(\mathrm{n}) \cup \mathrm{CREF}^w(\mathrm{n})\ ) -_{nc} \mathrm{UREF}^{rw}(\mathrm{n}) \tag{7.24}$$

Figure 7.22 Data-flow equations computed within an interval (with cache constraints) appears in both arguments, the age value of S in the result set will be the maximum of the ages of S in the arguments. Second, we introduce a new operator $\odot$, which models the "aging" effects of passing through a given block. Let X be the set of sections that reaches the start of node n, and let Y be the set of sections accessed within n. To form the set $X \odot Y$, we compute the number of capacity misses that will be caused by executing n (by computing the volume of the set $Y - (X \cap Y)$), then "age" each of the sections in X by the resulting miss count, and finally remove any sections whose age values indicate that they have been displaced from the cache. The $\odot$ operator is substantially similar to the *FiniteCache*-$\vee$ operator described in Figure 6.5. The OUT sets are then computed using the aged ("Δ") versions of the various IN sets, *i.e.*, $\mathrm{UIN}^r_\Delta(n)$, $\mathrm{UIN}^w_\Delta(n)$, $\mathrm{CIN}^r_\Delta(n)$, and $\mathrm{CIN}^w_\Delta(n)$. The equations for interval summarization (shown in Figure 7.23) are essentially the same as those in Figure 7.19, except that the $\uplus$ operator must take section volume effects into account.

$$\mathrm{UREF}^r(\mathrm{S}) = \biguplus_{loop} (\ \mathrm{UOUT}^r(\mathrm{E})\ ) - \biguplus_{loop} (\ \mathrm{COUT}^w(\mathrm{E})\ ) \tag{7.25}$$

$$\mathrm{UREF}^w(\mathrm{S}) = \biguplus_{loop} (\ \mathrm{UOUT}^w(\mathrm{E})\ ) -_{nc} \biguplus_{loop} (\ \mathrm{COUT}^{rw}(\mathrm{E})\ ) \tag{7.26}$$

$$\mathrm{CREF}^r(\mathrm{S}) = \biguplus_{loop} (\ \mathrm{COUT}^r(\mathrm{E})\ ) - \biguplus_{loop} (\ \mathrm{UOUT}^w(\mathrm{E})\ ) \tag{7.27}$$

$$\mathrm{CREF}^w(\mathrm{S}) = \biguplus_{loop} (\ \mathrm{COUT}^w(\mathrm{E})\ ) -_{nc} \biguplus_{loop} (\ \mathrm{UOUT}^{rw}(\mathrm{E})\ ) \tag{7.28}$$

Figure 7.23 Interval summarization equations (with cache constraints)

7.6 Optimizations

In this Section, we show how a compiler can use the data-flow information provided by our framework to improve prefetching for parallel programs. Section 7.6.1 outlines a strategy for prefetching of references that incur coherence misses. Section 7.6.2 discusses additional optimization opportunities related to widely shared data, exclusive-mode prefetching, and false sharing.

```
C$PAR   parallel shared(a,b,n,...) local(i,j,k)
C$PAR   pdo
        do j = 2, 99
           do i = 1, 100
              a(i,j) = ...
              b(i,j) = ...
           enddo
        enddo
C$PAR   end pdo
C$PAR   barrier
C$PAR   pdo
        do k = 2, 99
           do i = 1, 100
              ... = a(i,k) + b(k,i) ...
              ... = a(i,k+1) + a(i,k-1)
           enddo
        enddo
C$PAR   end pdo
C$PAR   end parallel
```

Figure 7.24 Example with coherence misses

```
C$PAR    parallel shared(x,...) local(i,j,k)
C$PAR    pdo
         do j = 1, 100
            do i = 1, 100
               ... = x(i,1)
            enddo
         enddo
C$PAR    end pdo
C$PAR    pdo
         do k = 1, 100
            do i = 1, 100
               x(i,k) = ...
            enddo
         enddo
C$PAR    end pdo
C$PAR    end parallel
```

Figure 7.25 Example with many-processor read-sharing

```
C$PAR    parallel shared(a,...) local(i,j)
C$PAR    pdo
         do j = 2, 99
            do i = 1, 100
               a(i,j) = ...
            enddo
         enddo
C$PAR    pdo
         do i = 1, 100
            a(i,1) = ...
            a(i,100) = ...
         enddo
C$PAR    end pdo
C$PAR    end parallel
```

Figure 7.26 Example with false sharing

7.6.1 Exploiting coherence miss information

Our framework provides information about the sets of references likely to cause coherence misses, but in order to derive an effective prefetching strategy, we need to combine this knowledge with information about the iteration space of the loop nest being optimized.

Consider the second loop nest in Figure 7.24. The data-flow framework will predict coherence activity for 3 of the 4 references in this nest: `a(i,k+1)`, `a(i,k-1)`, and `b(k,i)`. Our compiler then further classifies the references into two sets: those that incur coherence misses on every loop iteration (*unconditional* coherence misses) and those that incur coherence misses only on some small subset of iterations (*conditional* coherence misses).

Unconditional coherence misses

Our main mechanism for handling the long latencies of coherence misses is to issue prefetches for the references in question farther in advance than for other data. We do this either by increasing the *prefetching distance* for the references or by applying outer loop pipelining, as described in Section 3.4.5. In some situations, only a subset of the references will be identified as causing long-latency cache misses. In these cases we use different prefetching distances for the various references, fetching the long-latency references farther in advance than the rest of the data.

Conditional coherence misses

Consider the references `a(i,k+1)` and `a(i,k-1)` in Figure 7.24. In the `doall k` loop, if processor P is assigned iterations K_p through $K_p + b$, then it will read the set of elements a(1:100,$K_p - 1$:$K_p + b + 1$). Of these elements, a(1:100,K_p:$K_p + b$) will have been written previously on the same processor (and thus will not cause coherence misses), whereas the elements a(1:100,$K_p - 1$) and a(1:100,$K_p + b + 1$) will have been written on a neighboring processor and will probably result in coherence misses. This type of "nearest neighbor" communication occurs quite often in matrix-based scientific programs. In such situations, we don't want to treat all the references as long-latency misses, since coherence activity only takes place on a subset of the iterations (and since prefetches that arrive too early may displace useful data). Instead, the compiler can apply loop peeling to isolate the coherence-causing loop iterations.

We identify peeling opportunities as follows. Given a section S^r corresponding to read reference r in CFG node N, peeling is applicable if there is a section $S^w_X \in$ UIN^w(n) such that $S^w_X \not\approx S^r$ where

1. PMAP(S^r) = PMAP(S^w_X), and
2. PMAP$(S^r) = \langle N, F \rangle$ where $N > 0$, and
3. the bounds on dimension N in REGION(S_r) are equal to the the bounds on dimension N in REGION(S^w_X) shifted by a small constant c.

If any references meet these criteria, then the compiler can peel the first and/or last c iterations of the parallel loop, effectively isolating the coherence misses within the peel loop.

7.6.2 Additional optimizations

Many-processor read sharing

As previously discussed, a write miss to a line that is being widely read-shared can require a series of K invalidation and acknowledgement messages where K is proportional to the degree of sharing. Figure 7.25 shows an example.

Consider the `doall k` loop in this example. Suppose that processor 1 is assigned iteration 1 of the loop. Each time it writes a cache line in the section `x(1:100,1)`, the hardware must invalidate all K copies of the line. Depending on the circumstances, an advanced write buffer might be able to hide the latency for such a write. However due to the nature of the invalidation protocol, there would still be considerable additional network traffic. One solution would be to determine the parallel nest where the line is multiply shared and apply a local invalidation operation following the last read of the data. This would cut the total network traffic in half (provided that the invalidations did not require acknowledgements) and would spread the traffic out over a longer period of time. The data-flow analysis required to drive such an optimization is very similar to that presented in this paper, but requires backward instead of forward propagation; this form of analysis is currently beyond the scope of our framework.

Exclusive-mode prefetching

As discussed in Section 7.2, when a processor issues a prefetch for a line that is about to be read and then immediately written, it makes sense to bring the target line into the cache in an exclusive/unmodified state, as opposed to fetching it initially in a shared state, and then subsequently acquiring an exclusive copy of the line. To accomplish this, the compiler must consider access patterns within the loop when selecting the type of prefetch for each read reference.

```
C$PAR   parallel shared(a,b,c,q)
C$PAR&      local(i1,i2,j,x)
C$PAR   pdo
        do j = 2, 99
          do i1 = 1, 100
            x = c(i1,j) + b(i1,j)
            if (a(i1,j) .eq. x) then
              b(i1,j) = 1
            endif
            c(i1,j) = 0
          enddo
          do i2 = 1, 100
            a(i2,j) = q + b(i2,j)
          enddo
        enddo
C$PAR   end pdo
C$PAR   end parallel
```

```
C$PAR   parallel shared(d,e,f,q)
C$PAR&      local(i,j)
C$PAR   pdo
        do j = 2, 99
          do i = 1, 100
            f(i,j) = d(i,j)
            if (e(i,j) .lt.  0) then
              d(i,j) = 0
            else
              d(i,j) = d(i,j) + 1
            endif
          enddo
        enddo
C$PAR   end pdo
C$PAR   end parallel
```

Figure 7.27 Exclusive-mode prefetching opportunities

For example, consider the first loop nest in Figure 7.27. The initial reference to the array "`c(i1,j)`" is a read, hence it might seem logical to fetch the cache line containing it in a shared state. Shortly after `c(i1,j)` is read, however, it is then written. If the line is prefetched in a shared state, the subsequent write will be forced to acquire an exclusive copy of the line, causing additional network traffic. It is more efficient, therefore, to use an exclusive-mode prefetch for "`c`" from the start, even though the first reference to "`c`" is a read.

Mowry *et al.* developed a compiler algorithm for deciding where to issue exclusive-mode prefetches [72, 74]. In this algorithm, references are grouped into equivalence classes based on group reuse. If any of the references within an equivalence class is a write, then an exclusive-mode prefetch is used for the reference.

While this strategy works well in many cases, there are some situations where it is not optimal. Consider again the example on the left hand side of Figure 7.27. Mowry's compiler would not place the two references to the array "`a`" in the same equivalence class, since they are not identically nested, and hence exclusive-mode prefetching would not be employed, resulting in additional unnecessary network traffic.

The algorithm used by Mowry *et al.* also does not consider control flow when deciding whether to use an exclusive-mode prefetch: in the case of the references to "`b`", the write to `b(i1,j)` is not always executed, so the exclusive-mode prefetch might not be necessary at all, depending on the outcome of the test. It may be that in such situations it is best to use an exclusive-mode prefetch only when the location *must* be written, as opposed to situations where the location *may* be written. The array "`d`" on the right hand side of Figure 7.27 is an example of such a situation: in spite of the control flow, we know that `d(i,j)` is always written after being read.

The information provided by our analysis framework can be used to make more intelligent decisions in the situations above. A simple method is to test each reference r to see if $S_X \sqsupseteq S_r$ for some $S_X \in \mathrm{UREF}^w(\mathrm{L})$ for the loop in question; if so, then exclusive-mode prefetching is profitable. More sophisticated tests are possible as well.

False sharing

The parallel loops in our model are synchronization-free, thus we can be guaranteed that if one processor writes an array element within a parallel loop, no other processor will read that element. With non-unit cache line sizes, however, there may be false sharing. For references to false-shared data, prefetching may actually make matters worse, not better. Figure 7.26 illustrates this situation.

In the first parallel loop ("`doall j`") each processor writes a block of columns of the array. In the second parallel loop, however, each processor is assigned a chunk of the first and last columns in the array. This will result in contention for the cache lines that are on the boundary between processors– if a line is prefetched too far in advance, it may very well be invalidated before it can be used, since another processor may be trying to write it.

Our framework provides a simple way recognize such situations: we examine all of the sections $S \in \mathrm{UREF}^w(\mathrm{N})$ (where N is a node contained in an inner loop), and suppress prefetching for references whose sections have mapping functions of the form

PMAP(S) = $\langle 1, \ldots \rangle$. While this will not eliminate the false sharing, it does eliminate the additional network traffic and thrashing caused by the prefetching.

7.7 Experiments

This section contains the results of a set of preliminary experiments, in which we apply the techniques described in 7.5 by hand to a set of subroutines taken from our benchmark suite.

7.7.1 Subroutines

We apply our techniques to the three subroutines shown in Figure 7.28. The subroutines were chosen as representatives of the program in question in terms of late prefetch behavior (fraction of late dirty remote prefetches vs. late clean remote prefetches).

Subroutine	Taken from program:	Loop nests	Remarks
`eflux`	`flo52`	5	Most late prefetches are dirty remote
`filter`	`hydro2d`	14	Most late prefetches are clean remote
`jacz`	`appsp`	4	Mix of late clean and late dirty remote prefetches

Figure 7.28 Selected subroutines

7.7.2 Optimizations

We measure the execution time of each subroutine with three compilation strategies, shown in Figure 7.29. The `Default` compilation strategy uses prefetching without any additional optimizations. In the `Long` strategy, we double the prefetching distance for all loops. In the `DF-Long`, `DF-Outer`, and `DF-Twice` strategies, we apply optimizations selectively, only where our data-flow framework predicts that they will be profitable. `DF-Long` uses a larger prefetching distance, `DF-Outer` uses outer loop pipelining, and `DF-Twice` uses a form of double prefetching, in which reference predicted to cause coherence misses are prefetched twice: the first prefetch is issued an outer loop iteration in advance, and then the second prefetch is issued using the default inner loop pipeline.

Strategy name	Description
`Default`	prefetching with no additional optimizations
`Long`	prefetching distance is doubled for all loops
`DF-Long`	prefetching distance is doubled for loops in which coherence activity is detected
`DF-Outer`	outer loop prefetching is applied to loops in which coherence activity is detected
`DF-Twice`	references predicted to cause coherence activity are prefetched twice, at outer loop level and at inner loop level

Figure 7.29 Optimization strategies

Figure 7.30 shows the performance of each compilation strategy, relative to the default version of prefetching. For "`eflux`", doubling the prefetching distance for all loops provides about a 3% reduction in execution time. In the `DF-Long` case, the compiler selects 2 of the 5 loop nests for long-distance prefetching, and achieves essentially the same improvement. In the `DF-Outer` and `DF-Twice` strategies, the more aggressive transformations provide better execution time reductions of 9.1% and 6.9% respectively. For "`filter`", doubling the prefetching distance for all loops causes a slight degradation in performance, and for the `DF-Long` strategy, there is little change in performance, since the compiler targets only one minor loop nest out of the 14 in the subroutine. The `DF-Outer` and `DF-Twice` strategies provide no change in performance, since the compiler cannot find any applicable loop nests. In the case of "`jacz`", the compiler targets 2 of the 4 loop nests in the procedure, but none of the optimizations appear to have any beneficial effects; in the case of `DF-Outer`, there is even a slight performance degradation.

Figure 7.31 shows the effects of each compilation strategy in terms of its effects on late prefetch amelioration and on overshoot. For each compilation strategy, the "late" column shows the percent reduction in the average number of cycles that the

	Compilation strategy			
	`Long`	`DF-Long`	`DF-Outer`	`DF-Twice`
`eflux`	2.9%	2.8%	9.1%	6.9%
`filter`	-0.7%	-0.1%	0%	0%
`jacz`	-0.1%	-0.1%	-2.1%	0.1%

Figure 7.30 Relative performance of different optimization strategies

	Compilation strategy							
	`Long`		`DF-Long`		`DF-Outer`		`DF-Twice`	
	late	over	late	over	late	over	late	over
`eflux`	8.1%	48.7%	1.7%	42.6%	14.5%	-33.8%	47.3%	1.8%
`filter`	42.8%	194.8%	43.4%	206.2%	--	--	--	--
`jacz`	3.2%	8.5%	3.5%	99.1%	24.7%	130.8%	49.5%	0%

Figure 7.31 Percent reduction in late prefetch penalty, increase in overshoot for each optimization strategy

processor has to wait for a prefetch instruction. The "over" column indicates the increase in overshoot caused by the strategy.

We interpret the data as follows. Nearly all the strategies result in an increase in overshoot, which is not surprising, given that this is typically the price of prefetching farther in advance. Most of the strategies also reduce the penalties due to late prefetches, as well. In the case of "`eflux`", the more aggressive strategies produce the best payoff. The `DF-Twice` strategy reduces the late prefetch penalty by almost 50%, with very little cost in terms of additional overshoot. The `DF-Outer` strategy produces less of a reduction in the late prefetch penalty, but actually does better overall than `DF-Twice`: this is due to the fact that the `jacz` subroutine has fairly short inner loops, and with outer loop pipelining a much higher percentage of prefetches are issued during the steady state stage of the pipeline. In the case of "`filter`", we see why the increased prefetching distance provides few benefits: the level of overshoot rises dramatically, cancelling out any scheduling benefits. A similar situation exists for "`jacz`".

In general, these results show that the data-flow analysis is able to recognize situations where scheduling optimizations are profitable (as in the case of "`eflux`") and when they are not profitable (as in the case of "`filter`"). For the one subroutine that exhibits a large fraction of late dirty remote prefetches, the optimizations selected by the compiler reduce the late prefetch penalty by 40 to 50%, with a decrease in execution time of 6 to 9%. For the subroutines with fewer late dirty remote prefetches, we see relatively little change in performance.

7.8 Summary

Applying software prefetching on DSM multiprocessors is more difficult than on uniprocessor machines. Cache miss latencies vary tremendously, and artifacts of the cache coherence protocol can result in miss latencies that are orders of magnitude larger than those encountered on uniprocessor systems. This variability can cause problems for existing prefetching techniques, which were originally developed for machines with uniform miss latencies.

In this chapter we provide a detailed experimental evaluation of software prefetching on DSM machines, using execution-driven simulation of a set of compiler-parallelized Fortran programs. Our experimental results confirm that late prefetches are much more of a problem on DSM machines than on uniprocessors. To attack the problem of prefetch scheduling for parallel programs, we describe a data-flow framework designed to detect situations where late prefetches are likely to arise. It operates by analyzing the program's sharing patterns to predict the locations where coherence misses are likely. Unlike previously developed methods, our framework incorporate knowledge of both the program's memory access patterns and the important characteristics of the cache subsystem on the target machine. We demonstrate methods for using the resulting data-flow information to improve the effectiveness of software prefetching, including better prefetching for coherence misses, and handling of false-shared and heavily shared data. Preliminary experimental results indicate that the framework is successful in detecting situations where scheduling optimizations are profitable, and that optimizations guided by the framework can significantly reduce late prefetch penalties in situations where coherence activity is plentiful.

Chapter 8

Conclusions

This dissertation advances the state of the art with respect to compiler support for software prefetching. Our contributions are in two areas: the detailed experimental evaluation of software prefetching for various architectures, and the development of new compiler algorithms and techniques to improve prefetching performance.

In our study of prefetch prefetching on uniprocessor architectures, we find that the chief impediments to optimal prefetching performance are cache conflicts, which reduce the level of prefetch coverage, and useless prefetches, which increase the total instruction overhead for prefetching. We develop a set of new reuse analysis strategies that are progressively more powerful than those used in previous studies, and we gauge their effectiveness in eliminating useless prefetches. We find that prefetching overshoot (prefetches of data that is subsequently displaced before being used) is largely due to cache conflicts, and not due to compiler-related factors. For uniprocessors, we find that prefetch scheduling is not a major source of poor performance, and that there are relatively few penalties due to poor compiler scheduling of prefetches. We demonstrate that scheduling optimizations can reduce the percentage of late prefetches for a given program, but we also find that for our simulated uniprocessor, there is little to be gained in terms of performance by these techniques.

As a precursor to our work on parallel prefetching, we introduce the notion of cross-loop reuse, and demonstrate a new form of data-flow analysis designed to detect it. We show how this analysis can be used to help reduce prefetching overhead. Additionally, we demonstrate how a compiler can use it to predict the profitability of transformations such as loop reversal and loop fusion, and we provide experimental results on the applicability of these techniques in a set of benchmark programs. Our cross-loop reuse framework forms the foundation for the tools we use to improve parallel prefetching.

Our multiprocessor prefetching study is conducted using a set of parallelized scientific programs in which the parallel structure of the programs is exposed to the compiler, as opposed to so-called explicitly parallel programs, in which the compiler often has little specific information on the program's parallelism. We study the performance of these programs running on a simulated DSM multiprocessor. Our results show that prefetching provides better performance improvements than in the uniprocessor case, but we also find that late prefetches are a much more serious problem, due to the more highly variable cache miss latencies on these architectures. We then develop a novel data-flow framework that is designed to predict one particular class of late prefetches. This framework builds on the tools we have developed for detecting cross-loop reuse. It combines information on the program's sharing patterns with an understanding of the target machine's coherence protocol and cache configuration to detect locations where coherence activity is likely. We show how the framework can be used to drive a variety of scheduling optimizations, and we provide experimental results that demonstrate its effectiveness.

Bibliography

[1] Sarita V. Adve and Mark D. Hill. Weak Ordering - A New Definition. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 2–14, Seattle, WA, May 1990.

[2] A. Agarwal, R. Bianchini, D. Chiaken, K. Johnson, D. Kratz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife machine: Architecture and performance. In *Proceedings of the 22th International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, June 1995.

[3] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz. APRIL: A processor architecture for multiprocessing. In *Proceedings of the 17th International Symposium on Computer Architecture*, Seattle, WA, May 1990.

[4] A. V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, second edition, 1986.

[5] J. R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[6] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, 1990.

[7] Gurindar S. Sohi amd Manoj Franklin. High-bandwidth data memory systems for superscalar processors. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 53–62, Santa Clara, CA, April 1991.

[8] B. Appelbe and B. Lakshmanan. Program transformations for locality using affinity regions. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[9] Jean-Loup Baer and Tien-Fu Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of Supercomputing '91*, pages 176–186, July 1991.

[10] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. *International Journal of Supercomputing Applications*, 5(3):63–73, Fall 1991.

[11] V. Balasundaram. *Interactive Parallelization of Numerical Scientific Programs*. PhD thesis, Dept. of Computer Science, Rice University, May 1989.

[12] V. Balasundaram. A mechanism for keeping useful internal information in parallel programming tools: The data access descriptor. *Journal of Parallel and Distributed Computing*, 9(2):154–170, June 1990.

[13] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Boston, MA, 1988.

[14] David Bernstein, Dohon Cohen, Ari Freund, and Dror E. Maydan. Compiler techniques for data prefetching on the powerPC. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Limassol, Cyprus, June 1995.

[15] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986.

[16] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, NY, June 1990.

[17] D. Callahan, J. Cocke, and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. *Journal of Parallel and Distributed Computing*, 5(5):517–550, October 1988.

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

[19] S. Carr, K. S. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, San Jose, CA, October 1994.

[20] Tien-Fu Chen. *Data Prefetching for High-Performance Processors*. PhD thesis, University of Washington, Seattle, WA, July 1993.

[21] Tien-Fu Chen and Jean-Loup Baer. A performance study of software and hardware data prefetching schemes. In *Proceedings of the 21th International Symposium on Computer Architecture*, April 1994.

[22] William Y. Chen, Scott. A. Mahlke, Pohua P. Chang, and Wen mei W. Hwu. Data access microarchitecturess for superscalar processors with compiler-assisted data prefetching. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, 1991.

[23] R. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. Technical Report SMLI 93-12; UWCSE 93-06-06, Sun Microsystems Laboratories, Inc. and University of Washington, 1993.

[24] CONVEX Computer Corporation. *Exemplar Architecture*. CONVEX Press, Richardson, Texas, first edition, 1993.

[25] G. Cybenko, L. Kipp, L. Pointer, and D. Kuck. Supercomputer performance evaluation and the Perfect benchmarks. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[26] Fredrik Dahlgren and Per Stenström. Evaluation of hardware-based stride and sequential hardware prefetching in shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(4):385–398, April 1996.

[27] S. Dwarkadas, J. R. Jump, and J. B. Sinclair. Execution-driven simulation of multiprocessors: Address and timing analysis. In *Journal of Transactions on Modeling and Computer Simulation*, October 1994.

[28] Keith I. Farkas, Norman P. Jouppi, and Paul Chow. How useful are non-blocking loads, stream buffers and speculative execution in multiple issue processors? In

Proceedings of the 1st International Symposium on High Performance Computer Architecture, pages 78–89, Raleigh, North Carolina, January 1995.

[29] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[30] J. Ferrante, V. Sarkar, and W. Thrash. On estimating and enhancing cache effectiveness. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing, Fourth International Workshop*, Santa Clara, CA, August 1991. Springer-Verlag.

[31] J. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[32] Kourosh Gharachorloo et al. Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors. In *Proc. of ASPLOS4*, pages 245–257, Santa Clara, CA, April 1991.

[33] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Hiding memory latency using dynamic scheduling in shared-memory multiprocessors. In *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[34] P.B. Gibbons and S.S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, 1986.

[35] G. Goff, K. Kennedy, and C.-W. Tseng. Practical dependence testing. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.

[36] E. Gornish, E. Granston, and A. Veidenbaum. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[37] 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, 1995.

[38] E. Granston and A. Veidenbaum. Detecting redundant accesses to array data. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[39] T. Gross and P. Steenkiste. Structured dataflow analysis for arrays and its use in an optimizing compiler. *Software—Practice and Experience*, 20(2):133–155, February 1990.

[40] M. Gupta and E. Schonberg. A framework for exploiting data availability to optimize communication. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[41] M. Gupta, E. Schonberg, and H. Srinivasan. A unified data-flow framework for optimizing communication. In *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, Ithaca, NY, August 1994.

[42] E. Hagersten. *Towards scalable cache only memory architectures.* PhD thesis, Swedish Institute of Computer Science, October 1992. SICS Dessertation Series 08.

[43] R. v. Hanxleden. *Compiler Support for Machine-Independent Parallelization of Irregular Problems.* PhD thesis, Dept. of Computer Science, Rice University, December 1994.

[44] R. v. Hanxleden and K. Kennedy. Give-N-Take — A balanced code placement framework. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, Orlando, FL, June 1994.

[45] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[46] Paul Havlak. *Interprocedural Symbolic Analysis.* PhD thesis, Dept. of Computer Science, Rice University, May 1994. Also available as CRPC-TR94451 from the Center for Research on Parallel Computation and CS-TR94-228 from the Rice Department of Computer Science.

[47] J. Hennessy and D. Patterson. *Computer Architecture A Quantitative Approach.* Morgan Kaufmann Publishers, San Mateo, CA, 1990.

[48] High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, 2(1-2):1–170, 1993.

[49] Mike Johnson. *Superscalar Microprocessor Design*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[50] N. Jouppi. Improving direct-mapped cache performance by addition of a small fully associative cache and prefetch buffers. In *Proceedings of the 17th International Symposium on Computer Architecture*, Seattle, WA, May 1990.

[51] R. M. Keller. Look-ahead processors. *ACM Computing Surveys*, 7(4):177–195, 1995.

[52] K. Kennedy and K. S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[53] Ken Kennedy. A survey of data flow analysis techniques. In S. Muchnick and N. Jones, editors, *Program Flow Analysis*, pages 5–54. Prentice-Hall, 1981.

[54] Daniel R. Kerns and Susan J. Eggers. Balanced scheduling: Instruction scheduling when memory latency is uncertain. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 278–289, Albuquerque, NM, June 1993.

[55] A. C. Klaiber and H. M. Levy. Architecture for software-controlled data prefetching. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 43–63, May 1991.

[56] J. S. Kowalik, editor. *Parallel MIMD Computation: HEP Supercomputer and its applications*. The MIT Press, Cambridge, MA, 1985. .

[57] Sanjay Krishnamurthy. A brief survey of papers on scheduling for pipelined processors. *ACM SIGPLAN Notices*, 25(7):97–106, 1990.

[58] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the Eighth International Symposium on Computer Architecture*, pages 81–87, May 1981.

[59] Kuck & Associates, Inc. *KAP User's Guide*. Champaign, IL 61820, 1988.

[60] Kiyoshi Kurihara, David Chaiken, and Anant Agarwal. Latency tolerance through multithreading in large-scale multiprocessors. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, pages 91–101, April 1991.

[61] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simon, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH multiprocessor. In *Proceedings of the 21th International Symposium on Computer Architecture*, April 1994.

[62] M. Lam. Software pipelining: An effective scheduing technique for VLIW machines. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, Atlanta, GA, June 1988.

[63] James Laudon and Daniel Lenoski. The SGI Origin: A ccNUMA Highly Scalable server. In *Proceedings of the 24th International Symposium on Computer Architecture*, Denver, Colorado, July 1997.

[64] J. Lee and A. J. Smith. Branch prediction strategies and branch target buffer design. *IEEE Computer*, 17(1):6–22, January 1984.

[65] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th International Symposium on Computer Architecture*, May 1990.

[66] W. Li and K. Pingali. Access normalization: Loop restructuring for NUMA compilers. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, Boston, MA, October 1992.

[67] Z. Li and P. Yew. Efficient interprocedural analysis for program restructuring for parallel programs. In *Proceedings of the ACM SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages, and Systems (PPEALS)*, New Haven, CT, July 1988.

[68] Tom Lovett and Russell Clapp. STiNG: A CC-NUMA computer system for the commercial marketplace. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 308–317, Philadelphia, Pennsylvania, May 22–24, 1996. ACM SIGARCH and IEEE Computer Society TCCA.

[69] E. Markatos and T. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):379–400, April 1994.

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

[71] F. McMahon. The Livermore Fortran Kernels: A computer test of the numerical performance range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, 1986.

[72] T. Mowry. *Tolerating Latency Through Software Controlled Data Prefetching*. PhD thesis, Dept. of Computer Science, Stanford University, March 1994.

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

[74] T. Mowry, M. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 62–73, Boston, MA, October 1992.

[75] Todd Mowry. Personal communication. Telephone conversation, September 1996.

[76] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. The Impact of Instruction-Level Parallelism on Multiprocessor Performance and Simulation Methodolgy. In *Proceedings of the 3rd International Symposium on High Performance Computer Architecture*, pages 72–83, February 1997.

[77] Subbarao Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of the 21th International Symposium on Computer Architecture*, April 1994.

[78] A. Porterfield. *Software Methods for Improvement of Cache Performance*. PhD thesis, Dept. of Computer Science, Rice University, May 1989.

[79] David K. Poulsen. *Memory latency reduction via data prefetching and data forwarding in shared memory multiprocessors*. PhD thesis, University of Illinois at Urbana-Champaign, 1994.

[80] David K. Poulsen and Pen-Chung Yew. Data prefetching and data forwarding in shared memory multiprocessors. In *Proceedings of the 1987 International Conference on Parallel Processing*, volume II, pages 276–280, 1994.

[81] S. A. Przybylski. *Cache and Memory Hierarchy Design: A Performance-Directed Approach*. Morgan Kaufmann, San Mateo, CA, 1990.

[82] W. Pugh. The definition of dependence distance. Technical Report CS-TR-2292, Dept. of Computer Science, Univ. of Maryland, College Park, November 1992.

[83] William Pugh. Counting solutions to presburger formulas: How and why. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, Orlando, FL, June 1994.

[84] 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 International Symposium on Computer Architecture*, Denver, Colorado, July 1997.

[85] Vatsa Santhanam, Edward H. Gornish, and Wei-Chung Hsu. Data prefetching on the HP PA-8000. In *Proceedings of the 24th International Symposium on Computer Architecture*, June 1997.

[86] J. Singh, W. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[87] A. J. Smith. Sequential program prefetching in memory hierarchies. *Computer*, 11(12):7–21, December 1978.

[88] A. J. Smith. Cache memories. *ACM Computing Surveys*, 13(3):473–530, September 1982.

[89] R. E. Tarjan. Testing flow graph reducibility. *Journal of Computer and System Sciences*, 9:355–365, 1974.

[90] O. Temam, E. Granston, and W. Jalby. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *Proceedings of Supercomputing '93*, Portland, OR, November 1993.

[91] S. Tjiang, M. E. Wolf, M. Lam, K. Pieper, and J. Hennessy. Integrating scalar optimization and parallelization. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing, Fourth International Workshop*, Santa Clara, CA, August 1991. Springer-Verlag.

[92] Marc Tremblay and J. Michael O'Connor. UltraSparc I: A four-issue processor supporting mulitimedia — combining on-chip multimedia instructions with a high-performance, four-issue architecture. *IEEE Micro*, 16(2):42–50, April 1996.

[93] H. Tsalapatas. Interprocedural array side effect analysis. Master's thesis, Dept. of Computer Science, Rice University, February 1994.

[94] D. Tullsen and S. Eggers. Limitations of cache prefetching on a bus-based multiprocessor. In *Proceedings of the 20th International Symposium on Computer Architecture*, San Diego, CA, May 1993.

[95] Dean M. Tullsen and Susan J. Eggers. Effective cache prefetching on bus-based multiprocessors. *ACM Transactions on Computer Systems*, 15(1):57–89, February 1995.

[96] J. Uniejewski. SPEC Benchmark Suite: Designed for today's advanced systems. SPEC Newsletter Volume 1, Issue 1, SPEC, Fall 1989.

[97] Wolf-Dietrich Weber and Anoop Gupta. Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: Preliminary results. In *Proceedings of the 16th International Symposium on Computer Architecture*, pages 273–280, June 1989.

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

[99] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. The MIT Press, Cambridge, MA, 1989.

Appendix A

Interprocedural analysis to support shadow regions

The analysis and transformations needed to add shadow regions to a Fortran programs are performed in three phases: a local analysis phase, an interprocedural analysis phase, and a transformation phase.

In the local phase, we record for each procedure P the set of arrays used within the procedure. For each array, we store the name of the array (using name/offset/length representation to deal with equivalencing) and the storage class of the array (local, common, or formal). We also examine each call site within the procedure and determine the actual positions where arrays are being passed, and for each such actual, the offset within the passed array (or a marker that indicates that the offset is unknown).

In the interprocedural phase, we make a top-down sweep over the call graph. The goal of the analysis is to compute the function `ShadowForFormal`, which holds a value for each array formal of each procedure. In the case that a formal f for a procedure p is always passed the same array a, and a appears in a common block, then the value of `ShadowForFormal`(p, f) will be a. In this case, a declaration for the block containing a can be simply added to the p, and the compiler can refer directly to it. In the case that different arrays are passed to f from different call sites, `ShadowForFormal`(p, f) will have the value $\langle$explicit$\rangle$, which indicates that the caller of f must pass the shadow region for a at the call along with a itself. Figure A.1 shows the algorithm used in the interprocedural phase.

Finally, in the transformation phase, declarations for shadow regions are inserted into each procedure, essentially by adding dummy variables of the appropriate size. Call sites and lists of formals are patched to include shadow region variables (where indicated by `ShadowForFormal`), and calls to the simulator runtime routines are inserted to tell the simulator at runtime the starting address and length of each array and corresponding shadow region.

procedure *ComputeShadowsForFormal*
input: call graph $G = (E, N)$
output: function `ShadowForFormal`: (proc, formal) $\rightarrow$ array

```
for each n ∈ N in reverse post-order:
    for each formal f of n:
        p ← ⟨nil⟩
        ShadowForFormal(n,f) ← false
        for each edge e = (x → n):
            let a be the array in x passed to n's formal f
            if (p = ⟨nil⟩) then
                ShadowForFormal(n,f) ← a
            endif
            if ((non-constant offset within a passed at e) or
                ((p ≠ ⟨nil⟩) and (a ≠ p)) or
                ((a is a formal within x) and (ShadowForFormal(x,a) = ⟨explicit⟩))) then
                ShadowForFormal(n,f) ← ⟨explicit⟩
            endif
            p = a
        endfor
    endfor
endfor
```

Figure A.1 Algorithm for computing `ShadowForFormal`

Appendix B

Cache volume estimation for nested loops

This section describes in more detail the mechanics of estimating the cache volume of a given loop nest as part of applying loop peeling. Our implementation uses the `Data Access Descriptor` abstraction for summarizing accesses to a particular region within an array; we refer to the reader to the Balasundaram's thesis for definitions of the various components used in `Data Access Descriptors` (`ReferenceTemplate`), as well as the relevant algorithms [11, 12].

The main entry point for the cache volume estimation algorithm is the function ComputeLoopVolume, which estimates the number of lines brought into the cache during the execution of a given loop nest. This function starts by computing `DAD`s for each of the references contained in the loop; each `DAD`s starts out as a single point, with all entries in the `ReferenceTemplate` for the `DAD` as invariants. The function then calls itself recursively to collect sections for each sub-loop within the target loop. The next step is to invoke the function LoopTranslate on the collected set of `DAD`s, effectively taking into account the loop bounds at level N. Next it calls the function CoalesceAndCombine, which eliminates redundant sections (it is this step that effectively captures the reuse taking place in the loop nest). Once the resulting summary has been coalesced, we compute the number of cache lines in it by summing the volumes of all of the `DAD`s. Unknown symbolics can introduce great inaccuracies in this process; when the compiler can't determine a particular section bound at compile time, it must conservatively assume that the entire extent in a particular dimension is accessed, resulting in frequent over-estimates.

Some extensions were required to the base `Data Access Descriptor` algorithms and data structures for this work. First, we implemented a DAD containment test, which is simply a trivial pairwise comparison of all of the boundary pairs in the `Data Access Descriptor`, and a "substantial containment" test, which checks for sections that are substantially overlapping (`a(1:n,1:n)` and `a(2:n+1,1:n)`, for example) without strict containment in either direction. Second, we implemented a scheme for estimating the number of cache lines accessed by a given `DAD`. Our algo-

```
Function   ComputeLoopVolume
inputs:    loop L at nesting level N
outputs:   tuple ⟨X, V⟩, where:
             • X is a reuse summary set summarizing the accesses within L, and
             • V is the cache line volume of the loop with respect to level N

{
  A = ∅
  For each reference R directly enclosed in L {
    Compute DAD D for R at loop level N
    Add D to A
  }
  for each loop L_j directly enclosed by L {
    ⟨X, V⟩ = ComputeLoopVolume(L_j, N-1)
    Add X to B
  }
  C = LoopTranslate(A, N)
  D = CoalesceAndCombine(C)
  V = sum of volumes of DADs in D
  return ⟨D, V⟩
}
```

```
Function   LoopTranslate(S, L, N)
inputs:    loop L at nesting level N
           reuse summary set S (sections invariant with respect to L)
outputs:   reuse summary set R that takes into account bounds of L

{
  for each DAD x ∈ S {
    y = x.TranslateToLoop(N)
    compute volume of y with respect to level N
    add y to R
  }
  return R
}
```

Function CoalesceAndCombine
inputs: reuse summary set S
outputs: coalesced version of S

```
{
    Q = ∅
    for each variable V in S {
        Let W be the set of DADs summarizing the access to V {
        for each DAD x in W {
            if (there is no DAD y in the V component of Q s.t. y contains x) {
                add x to Q
            }
        }
    return Q
}
```

rithm starts by using the method developed by Tsalapatas to determine the set of redundant boundaries in the DAD [93]. Once the redundant boundaries have been determined, we apply pattern matching to handle common cases (regions with entirely rectangular boundaries, etc.), making conservative assumptions in situations where more complex shapes crop up. We also take into account the cache line size and the stride of the access in question. More general methods exist for computing the volumes of shapes bounded by linear inequalities [83]. Nevertheless, in our experience pattern matching was simple to implement and worked well in most cases.