ABSTRACT

CHRISTNER, ROBERT K. Static Determination of Synchronization Method for Slipstream Multiprocessors. (Under the direction of Dr. Gregory T. Byrd).

The scalability of a distributed shared memory systems is limited largely by communication overhead, most of which can be attributed to memory latency and synchronization. In systems built with dual-processor single-chip multiprocessors (CMPs) a proposed solution to this scalability limitation is the use of slipstream execution mode for certain applications. Instead of using the second on-chip processor to run a separate slice of the parallel application, slipstream mode utilizes the second processor to run a reduced version of the same task. The reduced version, known as the advanced stream, skips certain high latency events but continues to make accurate forward progress and provides future reference behavior for the unreduced version, known as the redundant stream.

Slipstream mode provides several methods of synchronization between the advanced (A-) and redundant (R-) streams to govern how far the A-stream can advance in front of the R-stream and also to provide a method of keeping the A-stream on the correct control path. A current limitation of slipstream is that the method used for A-R synchronization must be specified by the user at run time and used throughout the entire execution of the program. This is because the method that results in the best performance is application dependant and unknown beforehand.

We investigate alternate procedures to determine the A-R synchronization method by the use of static code analysis in the form of both profile- and compiler-driven techniques. A trace profile algorithm is presented that gives insight into shared memory access patterns that favor certain synchronization methods. We also discuss compiler integration of the synchronization method determination. Techniques similar to those which are used for compiler-driven prefetching and data forwarding are shown that could be used in this context as well.

Static Determination of Synchronization Method for Slipstream Multiprocessors

by

Robert K. Christner

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial satisfaction of the
requirements for the Degree of
Master of Science

Computer Engineering

Raleigh

2004

Approved By:

Dr. Gregory T. Byrd
Chair of Advisory Committee

Dr. Eric Rotenberg

Dr. Vince Freeh

To my wife Elizabeth

Biography

Robert K. Christner was born on 20th June 1978 in Hesston, Kansas, USA. He received a B.A. in Physics from Goshen College in Goshen, Indiana and a B.S. in Electrical Engineering from Case Western Reserve University in Cleveland, OH in May 2002.

In August 2002, he entered the Masters program in Computer Engineering at North Carolina State University in Raleigh, NC. He joined the Slipstream-MP team under the direction of Dr. Gregory T. Byrd in August 2003, and was awarded the M.S. degree in May 2004.

Acknowledgements

First and foremost I would like to thank God, because only with the persistence, strength, and intelligence He has given me was I able to complete this thesis.

I am continually grateful for my wife Elizabeth; for her patience, understanding, and grace. Without her support and motivation I surely would have lost hope, or simply procrastinated for many years.

I must thank my parents, Merle and Evelyn Christner, for their outstanding parental guidance and their financial support through the earlier years of my college career. It is because of them that I was able to continue my education this far.

I would like to thank my advisor, Dr. Gregory T. Byrd, for his wisdom and patience. I am also very grateful for the opportunity to work as a research assistant under him.

I would also like to thank Dr. Eric Rotenberg for coming up with the slipstream idea, which has been the inspiration for this work, and for helping guide this thesis by always being around to answer questions and give input.

I would also like to thank Dr. Vince Freeh for agreeing to be on my committee and providing invaluable insight into this work.

And lastly, I would like to thank the current and past members of the Slipstream-MP team, for without their efforts this thesis would have been impossible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

At high levels of concurrency on distributed shared memory systems, the gain from parallelization is negligible compared to the communication overhead induced by shared memory accesses and synchronization. Thus, there is a limit to the amount of parallelism that will continue to give an overall speedup to the application. It has been shown that for many parallel benchmarks this limit occurs at a concurrency as low as sixteen or even eight [7]. This can lead to unused processors on large multiprocessor systems and a degradation in utilization and potential speedup.

A proposed solution that can utilize the unused processors to speed up the execution of an application at high concurrency is slipstream execution mode [6]. Slipstream achieves this by using the second processor on each dual-processor single-chip multiprocessor (e.g. IBM Power4) to run a reduced version of the same parallel task instead of a separate parallel slice of the overall application. This reduced version, known as the advanced stream (A-stream), runs along side but slightly ahead of the unreduced version, known as the redundant stream (R-stream). Thus, the A-stream has an accurate view of the future and can communicate information to its R-stream that can be used to improve the overall speedup. The simplest example of this communication is the A-stream's ability to prefetch data directly into the shared second-level (L2) cache.

The A-stream is allowed to run ahead of the R-stream by skipping high latency events, such as shared memory stores and certain synchronization events. It has been shown that control flow is determined mostly by local variables, so skipping shared memory stores should not affect the control flow of the A-stream, and thus it will continue to make correct forward progress. Shared memory loads are executed in the A-stream because the loads are essentially prefetching data into the shared L2 cache.

To limit the A-stream from advancing too far ahead of the R-stream, and to ensure that it does not deviate from the correct control path, an A-R synchronization mechanism was introduced [6]. This mechanism uses semaphores at traditional parallel synchronization events to determine if the A-stream should be allowed to advance into the next parallel region, or session. A-R synchronization has two definable options: whether the synchronization depends only on streams from the same processor (local synchronization) or on streams from every processor (global synchronization), and how many sessions ahead the A-stream is allowed to advance.

A-R synchronization is implemented with semaphores in the form of a token bucket. When an A-stream reaches a synchronization event (e.g. traditional barriers), it removes a token from the bucket if there is one available. If there is not a token available the A-stream must stall until a token is inserted into the bucket. An R-stream inserts a token when it either enters or leaves a synchronization event, depending on the mode of A-R synchronization that is used. If local synchronization is used, the R-stream inserts a token when it enters the synchronization event. If global synchronization is used, the R-stream inserts a token when it leaves the synchronization event, which signifies that all R-streams from all processors have reached the event. The amount of tokens the bucket is initialized with determines how many sessions the A-stream can be ahead of its R-stream. A more detailed description of A-R synchronization is given in Section 2.2.

A current limitation of the A-R synchronization mechanism is that the optimal mode, be it local or global, and how many sessions the A-stream is allowed to advance is application dependent. For some applications the wrong synchronization method can lead to a degradation in performance. Also, the synchronization method must be specified by the user at run time. To determine the optimal method, the user must run the application with all the possible modes, then select the best mode for future runs. This puts the burden on the user and makes slipstream mode a less appealing solution.

This work attempts to overcome the limitations of A-R synchronization by first

providing some insight into what factors lead to certain synchronization methods, and second using static code analysis to provide a framework that can be used to determine the best A-R synchronization method automatically with little or no intervention by the user.

1.2 Contributions

The main contributions of this work are:

- We provide an extensible trace profile framework that requires only simple modifications to the application, but allows for a wide array of advancements to further study A-R synchronization and slipstream execution mode.
- We detail a simple read-after-write dependency calculation algorithm, then improve on the algorithm to decrease the execution time of the profiling pass.
- We describe a shared access pattern detection method for determining whether a conservative (e.g. zero-token global) or aggressive (e.g. one-token global) mode will provide the best performance for a particular application.
- We discuss the implications and possibilities of incorporating A-R synchronization determination into a compiler. This would allow the process to be automated and eliminate the burden of choosing the A-R synchronization method from the user.

1.3 Organization of Thesis

Chapter 2 gives a detailed introduction to slipstream execution mode, A-R synchronization and the current limitations thereof. Chapter 3 discusses different methods to overcome the limitations of A-R synchronization by determining the optimal synchronization method using dynamic as well as static approaches. In Chapter 4 we introduce the trace profile static approach and give experimental results for several parallel benchmarks. Chapter 5 provides an in-depth study on using current compiler techniques to automatically determine the best A-R synchronization mode. Related work in trace profiling and compiler techniques is presented in Chapter 6. Chapter 7 concludes and gives possibilities for future work.

Chapter 2

Slipstream Execution Mode

2.1 Slipstream Introduction

In current distributed shared memory (DSM) systems, one of the largest limitations to scalability is communication overhead resulting from memory latency and synchronization. At a certain point the addition of computational resources no longer decreases and possibly increases the execution time of an application. Slipstream execution mode is an idea proposed to reduce the communication overhead, and thus increase the scalability of these systems [6]. Slipstream mode is implemented for a dual-processor single-chip multiprocessor (CMP) DSM system. Instead of using the second processor on a CMP node to execute another slice, or task, of the parallel application, slipstream mode uses the processor to run a reduced version of the same task, which is allowed to run ahead of the unreduced version and provide future execution behavior. This behavior can be exploited in a way that allows both versions to run faster in conjunction.

Figure 2.1 shows the three different execution modes for CMP-based multiprocessors. The double and single modes are the most common execution modes currently, with double mode running two parallel tasks on each CMP, and single mode running only one parallel task on each CMP. Single mode can result in better performance than double mode at the scalability limit by reducing network traffic and contention at the L2 cache [7]. In single and slipstream modes the same number of parallel tasks are executed simultaneously.

However, slipstream mode utilizes the second processor on a CMP, whereas in single mode the second processor is idle.

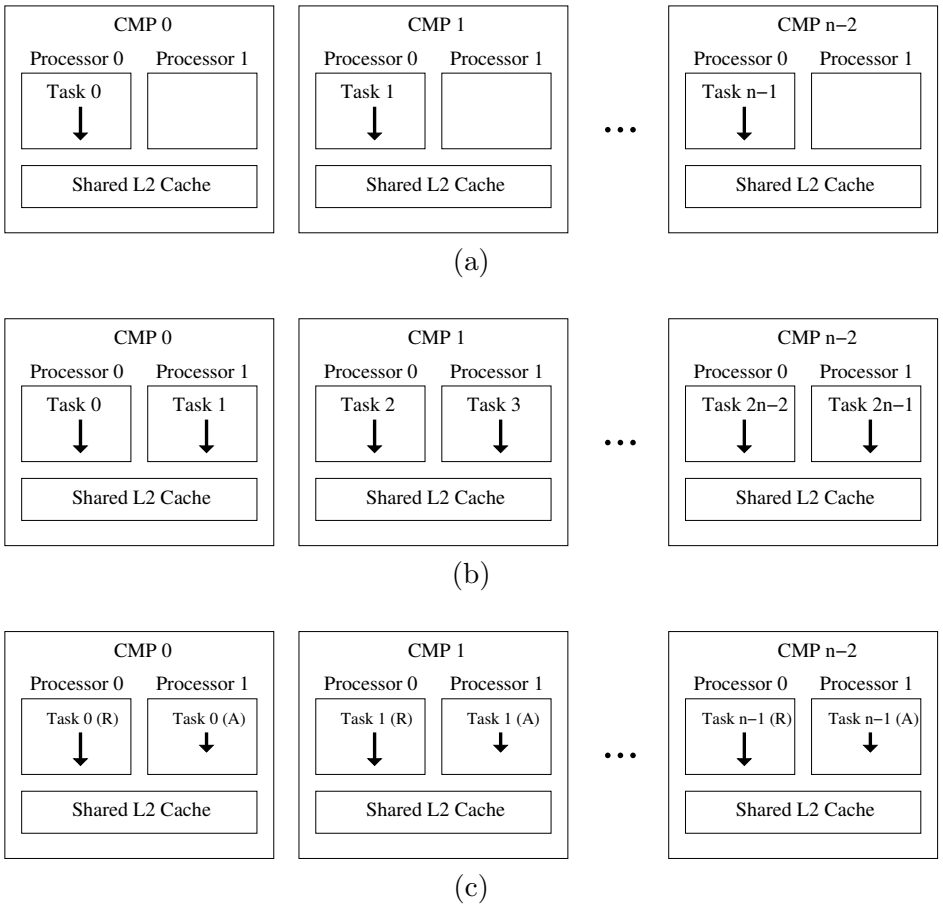


Figure 2.1: Execution modes for CMP-based multiprocessors: (a) single mode execution, (b) double mode execution, (c) slipstream mode execution.

In slipstream mode the reduced version of the program, known as the advanced stream (A-stream), skips high latency events such as shared memory writes, and synchronization events to run ahead of the unreduced version, called the redundant stream (R-stream). The A-stream continues to make accurate forward progress, and thus provides future profile information that can be used by the R-stream in several ways. The current implementation of slipstream mode utilizes a number of optimizations aimed at reducing the communication overhead resulting from coherence events. Since the CMPs utilize a shared L2 cache the natural benefit of slipstream mode is data prefetching as a result of the A-stream bringing future data into the L2 cache as it is progressing forward. Two other

optimizations that have been researched as benefits from slipstream mode are dynamic self-invalidation and barrier speculation [6]. Use of these optimizations, along with prefetching, leads to an increase in efficiency, and consequently a reduced execution time. This execution time is often better at the scalability limit than using the two processors independently for additional parallelism.

To reduce the execution of the A-stream, several key high latency events, such as shared-memory stores and synchronization events, are speculatively removed. By not executing these events the A-stream can advance much more quickly than the R-stream. However, there is a point at which the prefetches provided by the A-stream are too early. Thus, the prefetched data may be invalid or cause premature migration from processor to processor. To prevent the A-stream from advancing too far, a synchronization method has been implemented. The A-R synchronization method, which is detailed in Section 2.2, can be run in several modes. The mode chosen has a large impact on the performance of the application, but currently must be selected by the user at run time [7]. Since it is not always clear which A-R synchronization mode will allow for the best performance, the current implementation requires the user to run the application multiple times to determine the optimal mode. This can be a severe limitation of slipstream execution mode since the applications for these systems can typically take hours for a single execution. Several methods for automatically determining the A-R synchronization mode have been proposed. This paper presents the results of implementing a static analysis technique.

2.2 A-R Synchronization

The importance of A-R synchronization is two-fold. First, as stated in Section 2.1, A-R synchronization prevents the A-stream from advancing too far in front of the R-stream. Second, it provides a mechanism to correct a *rogue* A-stream (one that has deviated from the correct control path) [7]. To implement A-R synchronization the libraries for barriers and event-wait synchronizations are modified further. The libraries had to be modified initially to let the A-stream skip these synchronization events. When an A-stream reaches the end of a session, which is signified by either a barrier or an event-wait synchronization, it either skips the operation or waits for permission to continue from the R-stream. This is implemented using a single counting semaphore for each A-stream and R-stream pair. The

first A-R synchronization parameter that must be specified by the user at run time is the maximum sessions the A-stream can advance ahead of the R-stream. This is implemented in a *token bucket* fashion as in Figure 2.2 [7]. The semaphore is initialized with the total number of sessions provided by the user input, each of which corresponds to a token. When an A-stream enters a synchronization event, it removes a token from the bucket. If there are zero tokens left in the bucket, the A-stream must wait. When an R-stream reaches a synchronization event it inserts a token, which releases the A-stream to continue executing.

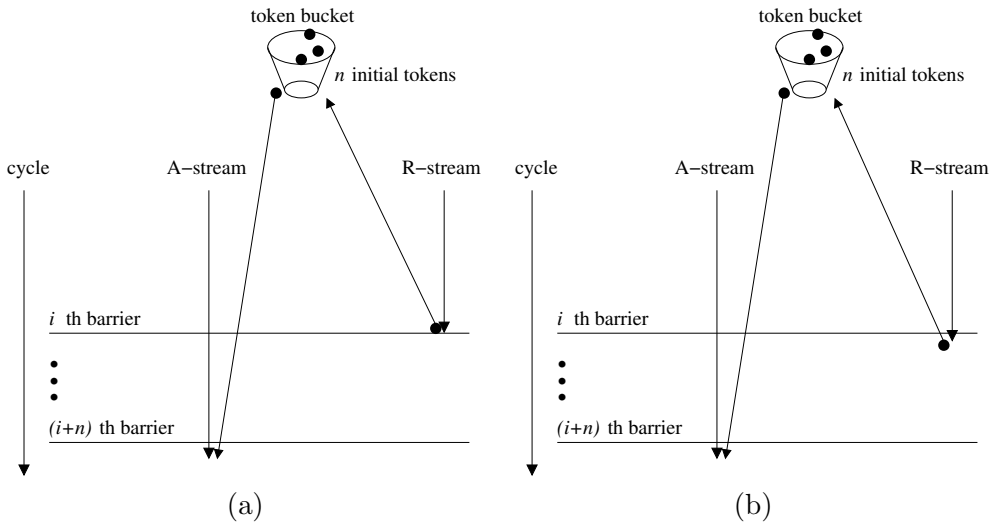


Figure 2.2: A-R synchronization methods: (a) local; a token is inserted by the R-stream before *entering* barrier synchronization, (b) global; a token is inserted by the R-stream after *exiting* barrier synchronization

The second parameter that the user specifies is whether the R-stream issues a token upon entering the synchronization event, or upon exiting. If the R-stream issues a token upon entering, the A-stream is released after only its local R-stream reaches the event, and thus it is termed local synchronization. If the R-stream issues a token upon exiting, the A-stream is released only after all the R-streams participating in the synchronization have reached the event. This is termed global synchronization (refer to Figure 2.2).

A-R synchronization is also responsible for correcting a deviated A-stream, which it does by killing the *rogue* A-stream and spawning a new one. An A-stream is determined to be rogue if its companion R-stream reaches a barrier before it [6]. There are also the problems of I/O, initial memory allocation, and other global operations. To handle this,

the A-stream ignores these operations, but blocks and waits for the R-stream to complete them before moving ahead.

2.3 Benchmarks

The benchmarks used for this study were selected because of the wide range of speedups that they exhibit from different A-R synchronization modes. A listing and a brief summary of the benchmarks used is given in Table 2.1.

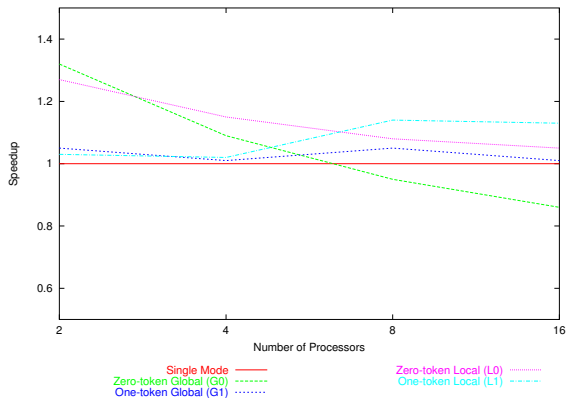
SOR is a red-black successive over-relaxation algorithm for solving partial differential equations. In this implementation, the red matrix and the black matrix are separated in memory to avoid false sharing [1].

LU is a kernel included in the SPLASH-2 parallel benchmark suite, and is used as a differential equation solver by factoring a dense matrix into a product of lower and upper triangular matrices [17].

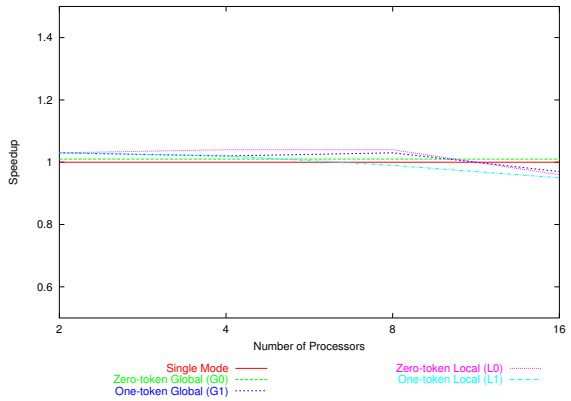
OCEAN is an application that studies ocean movements, and is also a SPLASH-2 benchmark [17]. It uses a red-black Gauss-Seidel multigrid equation solver [2].

Table 2.1: Benchmarks used throughout this study.

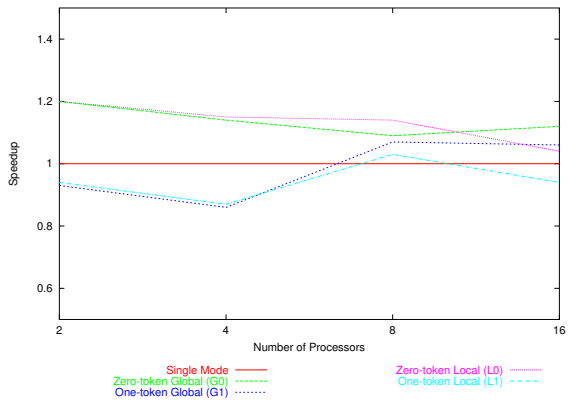
Figure 2.3 presents speedup (compared to single mode) for the assortment of benchmarks when the two A-R synchronization parameters were varied [7]. As can be seen in the figure, the optimal parameters are highly dependent on the application and the number of processors used. For example, in SOR with four processors the synchronization speedup in order of performance benefit is L0, G0, L1, and G1, whereas with eight processors the speedup is now L1, L0, G1, and G0. OCEAN shows a completely different order for eight processors with L0, G0, G1, and L1.



(a) SOR



(b) LU



(c) OCEAN

Figure 2.3: Speedup for slipstream mode over single mode with four different A-R synchronization methods for (a) SOR, (b) LU, and (c) OCEAN (all with default data set sizes).

Chapter 3

Determining A-R Synchronization

The current method of selecting the parameters for A-R synchronization at run time is undesirable for a couple of reasons. First, it places the burden on the user to select the most optimal mode. Second, this method may involve running the application multiple times to determine the best mode.

Several solutions to this problem have been proposed. The first set of solutions are termed *dynamic*, and are used to determine the synchronization method at run-time. Dynamic methods use execution performance as the measure for method determination. The second set of solutions are used to determine the best method *before* run-time, and are termed *static*. Static methods are also differentiated in that they use memory access patterns for determining the method.

3.1 Dynamic Determination

It is beneficial to first look at a dynamic technique that has been proposed to automatically choose the best synchronization mode for the running application. This technique is outside the scope of this research, but provides a background to distinguish this thesis from other work. The goal of the technique is to adaptively switch synchronization modes by analyzing the current execution performance. One method of implementing this approach would be to start with a conservative synchronization method, and at each barrier

examine certain criterion that could indicate the effectiveness of the current method, and whether a more or less aggressive method might give better performance. The criterion could be in the form of prefetch statistics gathered with performance counters, in conjunction with how far ahead the A-stream was able to get in the last session. The difficulty in this method lies in collecting and analyzing the statistics with minimal overhead, so as to not degrade performance. Also, the aggressive methods span multiple sessions, so they may require a more advanced statistical analysis.

To show the effectiveness of dynamically switching throughout the program, previous work uses a sliding window method to switch between synchronization modes [13]. This method reduces the overhead involved with the switching by not allowing aggressive switching at every session. The technique however currently requires a profiling run to generate a trace that gives the execution time for each of the four synchronization methods for each session. Only with this information fed into the running program can the dynamic switching occur. The dynamic methods have the advantage of tailoring the synchronization to the profile of application, so that different synchronization methods can be used for different computational phases.

3.2 Static Determination

The focus of this work is on investigating static analysis techniques to determine the best A-R synchronization, without running multiple versions of application. Static techniques can be broken down into profile-drive and compiler-driven. Profile-driven techniques determine the best method by analyzing a shared memory access input trace for certain patterns that favor specific synchronization methods. These techniques are described in greater detail in Section 3.2.1. Compiler-driven techniques are similar to profile-driven. However, the memory access pattern analysis is conducted on source code (or intermediate representation) instead of on a memory execution trace. Compiler-driven analysis is discussed in Section 3.2.2. These techniques can be expanded to incorporate the changes in access patterns between different computation regions, and accordingly switch methods intermittently throughout the execution of the program. However, we do not explore that aspect.

We make the assumption that slipstream execution mode is desired over single

mode or double mode. We do not incorporate the determination of whether slipstream gives any improvement over tradition execution modes. From previous studies on slipstream execution mode, it can be seen that most applications that are improved by slipstream favor either aggressive (one-token local) or conservative (zero-token global) synchronization methods. Because of this reason, only the two extremes were considered for this simple discussion. Also, only the affect of the mode on prefetching was considered. The A-R synchronization mode has some implications on other optimizations, such as barrier speculation and self-invalidation, but these were not investigated.

A decrease in cache miss rate is the tangible result of reducing communication latency. Aggressive and conservative synchronization modes affect cache misses in several ways. In aggressive mode, the A-stream has the possibility of executing in a different session than its R-stream. For this to happen, the A-stream would need to skip at least one synchronization event. These events were originally put in place to ensure dependencies. Since the A-stream skipped the event, it could read a shared data variable (shared memory load) before the producer R-stream in the previous session finished writing that same shared data variable (shared memory store). In this situation, when the A-stream reads the shared data it would cause the shared memory location to be removed from exclusive ownership in the producer's cache. In the subsequent write to that shared data the producer would have to regain exclusive ownership and invalidate the data in the A-stream's cache. This leads to an extra cache miss in the producer processor, and provides no benefit for the consumer processor, and thus a degradation of performance [7].

In conservative mode, the previous situation can not happen because the A-stream is never allowed to advance past a synchronization event before all the participating R-streams reach the event. Conversely, when the A-stream reads a shared memory location, the synchronization event enforces that the producer R-stream has finished writing to that location. Therefore, if the A-stream loads a line that was in the exclusive state, even if it still results in a cache miss, the R-stream only has to fetch the data from memory, instead of retrieving it from a remote processor. The fetch is a much lower latency than the remote operation. However, since conservative mode does not allow the distance between the A-stream and the R-stream to get very large, the A-stream does not have as many opportunities to prefetch data early enough to successfully hide the latency from the R-stream.

An analysis of the code can show which mode, conservative or aggressive, would

produce better results for a given application. This analysis is similar to prior work that has been done for data forwarding [9, 10, 12]. Data forwarding is a mechanism that aims to determine when, and by which processor, certain data will be used next, and then forwards the data to that processor to reduce coherence cache misses. This mechanism requires knowledge of shared memory access patterns and producer and consumer information, both of which would be useful to determine the optimal A-R synchronization mode for an application. This information can be provided by static code analysis. Two approaches of static code analysis, profiling and compiler analysis, are used for data forwarding, and were initially considered for realizing an automatic selection of the optimal A-R synchronization method.

3.2.1 Profile-Driven Analysis

Profile-driven analysis techniques consist of collecting data on a single execution of the full program, and then using that data to determine the best mode for subsequent executions of the same program [12]. Data could be collected that would give information about how and when shared data is accessed, as well as which processor accessed it. The resulting list of data is usually considered a *trace*.

One use of profiling for data forwarding targets array elements and distinguishes between them by both the static reference number (index) and the base address [12]. In this method, separate reference information can be collected for each array reference in the program. The reference information contains the reference number and base address as well as the consumer processor. This information is then stored in a data structure that contains arrays of bit vectors, which contain the information for each reference. The data structure can then be analyzed for eligible forwarding writes, those in which there is a later read from a different processor. On subsequent executions of the application, this data structure is used to implement a forwarding write when possible.

A similar method of profiling could be used for slipstream mode. However, instead of analyzing the reference information for eligible forwarding writes, the access patterns could be analyzed to determine the best A-R synchronization mode. For example, if the distance (in time, or cycles) between the production and consumption of the shared data was large and accesses to shared data were infrequent, then this application would benefit most from an aggressive synchronization method. Aggressive synchronization methods do

not stipulate that the A-stream must wait for all R-streams to reach the synchronization event before proceeding. By allowing the A-stream to proceed before all the R-streams have reached the event, the A-stream has a better chance of prefetching early enough to hide the latency from the R-stream. Since the profile information indicates that there is a large distance between the production and consumption of data, there is only a small chance that the prefetched data will be invalidated before the R-stream has a chance to use it.

While profiling requires little complexity in the compiler, it does require a profiling run. The original goal of automatically selecting an A-R synchronization method was to minimize the burden on the user and programmer. Profiling reduces the burden somewhat from the original method, and it allows for a more comprehensive analysis of how A-R synchronization affects slipstream behavior. Compiler analysis shows greater promise in reaching the goal of an automatic method. However, an understanding of which shared memory access patterns favor which synchronization methods must be obtained first.

3.2.2 Compiler-Driven Analysis

The two largest advantages of compiler analysis over profiling is that it is transparent to both the programmer and the user and it has the potential to be much faster. Without the need for a memory access trace, a compiler method can simply augment the existing analysis with that which is needed for slipstream. Compiler analysis has been accomplished for data forwarding [9, 10] as well as data prefetching [5, 11]. As mentioned previously, the information necessary to implement data forwarding is very similar to that which is needed for selecting an A-R synchronization method. The information needed for data prefetching is somewhat different, but a few of the same techniques may be helpful for A-R synchronization analysis. A more complete description of compiler-driven analysis is given in Chapter 5.

Chapter 4

Trace Profile Tool

An approach to selecting the best A-R synchronization method via compiler-driven techniques requires an understanding of trends and patterns in the application before it can be implemented. The trace profile tool we describe in this section provides some of this information and illustrates a framework that can be easily expanded for future research.

Trace-driven simulators have been used for many years to aid in the development and evaluation of memory systems and hierarchies [16]. The major issue with this methodology is that memory traces for scientific workloads are extremely large and can generate file sizes in the gigabyte range. In addition, a tool that analyzes these large traces can take an unreasonably long time. Several techniques have been proposed to reduce these factors [16]. However, rather than researching and implementing these methods, we instead reduced the data set sizes for the applications studied to levels that result in tolerable execution times and file sizes. We claim that the general memory access patterns will not be greatly affected by the data set size in these workloads, because they are regular, loop-based, scientific workloads.

4.1 Shared Memory Cycle-Based Trace Profile

The main goal of implementing a profiling tool was to be able to analyze an application to determine what factors lead to a performance benefit or degradation with specific

A-R synchronization methods. Since the environment is a distributed shared memory system and the latency that slipstream tries to hide is due to cache misses as a result of shared memory, we are not concerned with local memory accesses. Also, shared memory misses have a significantly longer miss penalty than local memory misses. It is also assumed that most local variables result in cache hits, because they are used frequently and are relatively few in number in the type of applications that are typically used in these environments. Shared memory access patterns provide the data we need to analyze the application for trends that lead to a specific A-R synchronization method. Therefore, we create a profile model based only on shared accesses.

The trace created for each application is a list of entries for each shared memory access, barrier, or lock. Each entry contains the address being accessed, whether the access was a load, store, or synchronization event, and which processor was responsible. For simplicity a single cycle is assigned to each shared memory load, shared memory store, and synchronization event. Throughout the rest of this thesis *cycle* refers to this shared memory cycle.

4.2 Implementation

The only modification necessary to the application to create the trace is an instrumentation to print out a list entry on every shared memory access or synchronization event. Shared memory variables are usually clearly indicated by a global memory allocation macro such as `G_MALLOC` or `sh_malloc`. Barriers and locks are also clearly indicated, as well as the current processor ID. Some difficulty may arise when trying to instrument code that reassigns private pointer variables to shared addresses. However, scientific workloads do not typically exhibit this behavior. It may be beneficial in the future to automate the instrumentation with a simple pre-processor step in the profile tool.

Once the trace files are created, the profile tool is used to import these entries into a data structure to be used in future analysis. The data structure is a two dimensional dynamically allocated matrix, which is indexed in one dimension by the processor ID, and in the other by the cycle number. Initially, the cycle numbers are independent of other processors. Each barrier and lock only has a cost of a single cycle, because there is no synchronization with the rest of the processors at this point. Therefore, the first pass

through the matrix must line up all the barriers to the worst case cycle time of that session out of all the processors, and modify the cycles of all the other instructions accordingly. Figure 4.1 presents the algorithm used for this pass.

```
for(all processors) {
  for(all instructions starting at the last instruction) {
    if(instruction type is BARRIER) {
      if(this is the last barrier in the instruction trace) {
        barrier_increment = the very last instruction's cycle
                          - this barrier's cycle;
        increment = barrier_increment
                  - this session's maximum length over all processors
                  + this session's length for this processor;
      } else if(this is the first barrier in the instruction trace ) {
        barrier_increment = 0;
      } else { //this is a barrier somewhere in the middle
        barrier_increment = increment;
        increment = increment
                  - this session's maximum length over all processors
                  + this session's length for this processor;
      }
      add barrier_increment to this barrier's cycle;
    } else { //the instruction type is either READ or WRITE
      add increment to the regular instruction's cycle;
} } }
```

Figure 4.1: Algorithm for lining up barriers within a cycle-based trace matrix.

The algorithm assumes that there is a barrier at the beginning and end of the trace. The algorithm starts by changing the cycle number of the last barrier in the instruction trace to the very last cycle across all processors. This maximum cycle is calculated by summing together each session's maximum length across all processors. It then iterates backwards through the instruction stream and increments each regular instruction cycle (non-barriers) by how much longer the session is now than it was before the last barrier was moved. Figure 4.2 gives an example.

If the maximum cycle number is 220, and processor zero (P0) ends on cycle number 215, and processor one (P1) ends on cycle 210, the last barriers of P0 and P1 will have their cycle numbers changed to 220. Then, if the last session of P1 happens to be the longest of all the processors at 10 cycles, but the last session of P0 is only 5 cycles, the next-to-last barriers in both processors will be moved to cycle 210. All the intermediate instructions

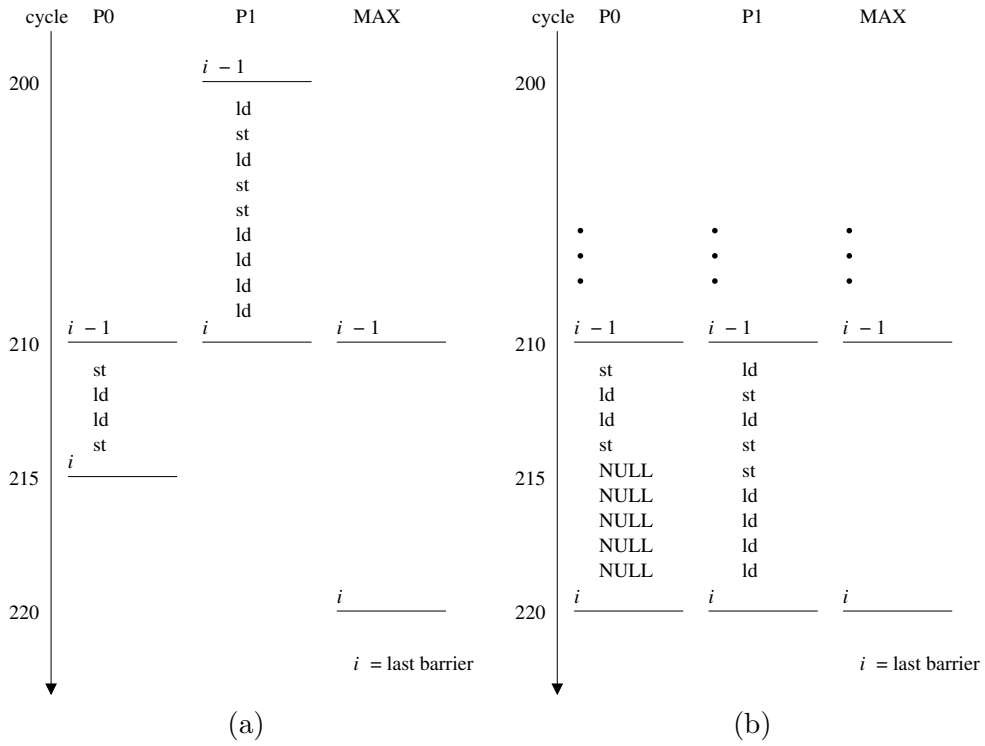


Figure 4.2: Example of lining up barriers for a case where there are more than two processors; (a) original instruction trace, (b) final instruction trace.

on both processors will be moved forward the same amount as the next-to-last barrier (no cycles in the case of P0, 15 in the case of P1). When the pass is complete there are blank instructions in the form of `NULL` pointers to fill the empty space between the last instruction in a session and the barrier to the next session. Therefore, the barrier now incurs the cost of all the added empty instructions. This simulates some notion of barrier stall time when calculating the dependencies.

4.3 RAW Dependency Algorithm

If A-streams are allowed to advance as far forward as they can by skipping all shared memory stores and synchronization events, there will be a point at which prefetching starts to pollute the shared L2 cache. This happens when too many global read-after-write (RAW) dependencies are broken. A global RAW dependency occurs when a processor writes to a shared data variable, and subsequently a different processor reads that same variable.

That read is dependent on the previous write. Synchronization events, such as barriers and locks, are set in place to protect these dependencies. However, since an A-stream is allowed to skip these events on occasion, it is now possible that an A-stream can violate these dependencies by performing a read prior to a dependent write being completed by another processor's R-stream. Only RAW dependencies are a concern because the A-stream does not commit any shared memory writes, so the synchronization events still protect write-after-write (WAW) and write-after-read (WAR) dependencies. Thus, the next pass in the profiling tool is used to determine the RAW dependencies. We present two algorithms to accomplish this. The simple algorithm is correct and was sufficient, but eventually was too inefficient to gain enough data in a reasonable amount of time. Therefore an improved algorithm was developed.

4.3.1 Simple Algorithm

The simple algorithm is shown in Figure 4.3. The main concept for this algorithm is to search the instruction trace from the end backwards, and for every read find the closest previous write of the same address. If a previous write exists to the same address, it is a dependency. These dependencies can then be used to find trends and patterns that can be analyzed to determine the best A-R synchronization method.

4.3.2 Improved Algorithm

Although the performance of the profiling tool was not initially a concern, the increasing complexity of the tool made it a large burden to collect data. Therefore it was necessary to develop an improved algorithm. Figure 4.4 shows this algorithm. In general, shared data is read much more often than it is written. Therefore, the improved algorithm makes a pass through the instruction matrix, creating a list of write instructions before dependencies are found. This list is then used to find dependencies from the store instruction to its dependent loads, instead of the inverse as before. The difference here is that a single store can have multiple dependent loads, whereas a load can only have one dependent store.

For all the results presented in this paper the benchmarks were run with smaller data set sizes than the default. Table 4.1 lists the sizes used. The reduction in data set size

```
for(all processors) {
  for(all instructions in the matrix) {
    if(instruction type is READ) {
      for(all processors) {
        /* only look for writes in other processors */
        if(write processor is different than the read processor) {
          /* set the write cycle to the same cycle as this read */
          /* look backwards from here to find the closest write */
          for(all instructions starting at this cycle looking backwards) {
            if(instruction type is WRITE and
               this instruction's address matches the read address) {
              /* add dependency */
              if(the distance from the read to its preceding barrier is greater
                 than that of the write to its preceding barrier) {
                /* flag the dependency as G1-type */
              } else {
                /* flag the dependency as G0-type */
              }
              /* we found a dependency, quit looking */
              break out of write loops and go to next read;
} } } } } } }
```

Figure 4.3: Simple algorithm for RAW dependence calculation.

was to facilitate a faster turn-around on data collection. The profiling passes are slow in nature, so execution time was a major concern. The improved algorithm gives a significant reduction in execution time of the profiling tool. Figure 4.5 shows the average speedup achieved by the improved algorithm over the simple algorithm.

benchmark	data set size
SOR	128x128 (2 iterations)
LU	64x64
OCEAN	32x32

Table 4.1: Benchmarks and data set sizes used throughout this paper.

```
for(all processors) {
  for(all instructions in the write list for this processor) {
    /* search through instructions from this cycle forward to find */
    /* reads that match this write */
    stop_cycle = last instruction in total instruction matrix;
    for(all processors) {
      for(all instructions from this cycle until stop_cycle) {
        if(instruction type is READ) {
          /* we are only concerned with reads from OTHER processors */
          if(this processor ID and that of the write are different) {
            /* check to see if we have a dependency */
            if(this instructions address matches that of the write) {
              if(a depedancy for this read already exists) {
                /* remove the old dependency */
              }
              /* add the new dependency, but don't quit looking */
              if(the distance from the read to its preceeding barrier is greater
                 than that of the write to its preceeding barrier) {
                /* flag the dependency as G1-type */
              } else {
                /* flag the dependency as G0-type */
              }
            }
          }
        } else if(instruction type is WRITE) {
          /* stop looking for dependencies on the current write? */
          if(this instruction's address matches that from the write_list) {
            stop_cycle = this instruction's cycle;
            break out of the instruction loop and move on to the next processor;
} } } } } }
```

Figure 4.4: Improved algorithm for RAW dependence calculation.

4.4 Zero-token Global verses One-token Global

Initially, the primary goal of the tool was to discover what trends determine whether a conservative or aggressive A-R synchronization method would be more effective. By limiting the study to only these two extremes, a simpler but still insightful analysis can be conducted. From the RAW dependence analysis it can be noted where the dependent loads and stores are located within the session. We consider the most conservative method, zero-token global (G0), and the next most aggressive method, one-token global (G1). Each RAW dependence can then be labeled as either one that favors G0 (G0-type), or G1 (G1-type).

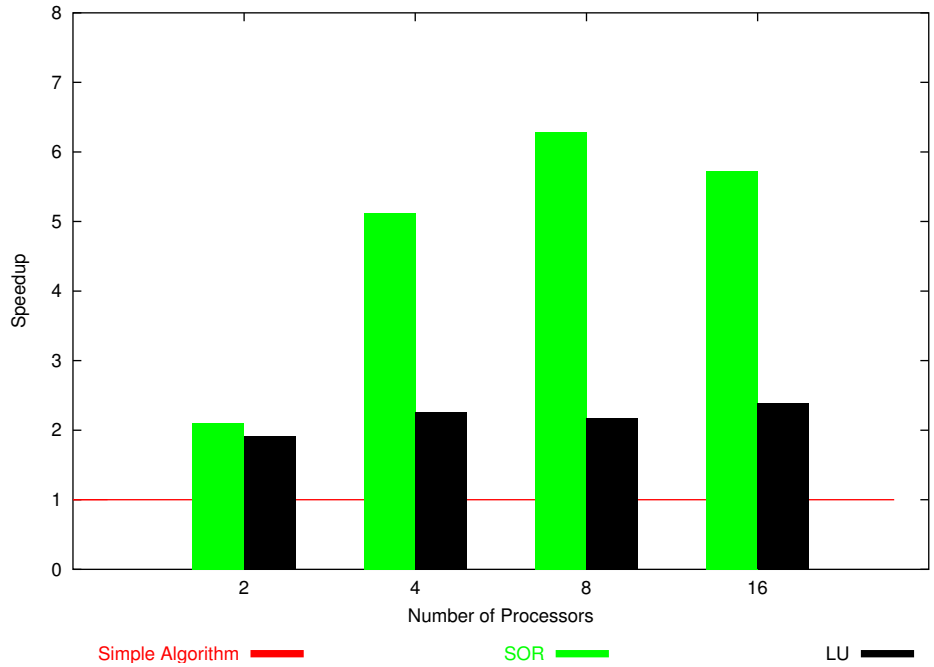


Figure 4.5: Average speedup for LU and SOR given by the improved RAW dependence algorithm over the simple algorithm.

In global-0, the case in which the A-stream and R-stream are furthest apart is when one processor's A-stream is at the end of a session while another processor's R-stream is still back at the beginning of the same session. In this situation no dependencies will be broken, but in some cases the A-stream can not effectively hide the latency, because it never gets a chance to get very far ahead. In the most aggressive case of global-1, a processor's A-stream can be at the end of a session while another processor's R-stream is held back at the beginning of the *previous* session. In this situation some dependencies may have been broken, but the A-stream was possibly able to successfully prefetch more valid data. However, before the A-stream can continue into the next session, it must wait for all R-streams to reach the previous synchronization event.

We claim that after some amount of warm up global-1 will likely end up in this position. What we consider next is how dependencies are handled in the following session. The A-stream is held up at a synchronization event waiting on the slowest processor's R-stream to reach the previous one. When that happens the A-stream is allowed to advance

into the next session at the same time the R-streams from all the processors advance into the current session. At this point the A-stream is exactly one session ahead. The A-stream now has a chance to break a RAW dependency. Doing so would bring an invalid copy of the data into its companion R-stream's cache and could remove the previous owner's exclusive cache state. If the previous owner writes to the data again, it would have to regain exclusive ownership, which would cause a degradation in performance. In order to avoid this, the other processor's R-stream would have to write the dependent data *before* the A-stream reaches the dependent load. To estimate the likelihood of this happening, we look at the distance from the dependent store to its previous synchronization event and compare it to the distance from the dependent load to its previous synchronization event. Figure 4.6 details this situation.

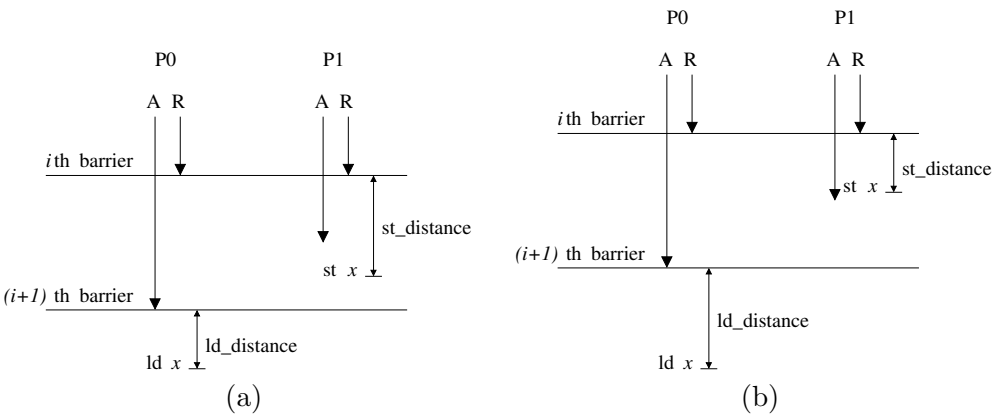


Figure 4.6: Examples of (a) G0-type and (b) G1-type dependencies

We simply say that if the store distance (`st_dist`) is greater than the load distance (`ld_dist`), the store is more likely to happen first, and therefore will likely avoid the dependence violation. This would be considered a G1-type, because the dependency is unlikely to be broken even if the synchronization method were global-1, or another semi-aggressive method like local-0. However if the load distance was greater it would be a G0-type. If global-0 were used the dependency would not be violated, whereas an aggressive method would likely break this dependency. The algorithm for finding the RAW dependency, shown in Figure 4.4, includes this analysis.

To determine if a particular application would be more effective with a global-0 or global-1 method, we sum the G0-types and G1-types. Whichever is greater is considered to be more likely and thus determines the best method. Figure 4.7 shows the ratio of G1-types to G0-types for three benchmarks. A value greater than one indicates the benchmark would favor a global-1 method. A value less than one indicates that a global-0 method would be more beneficial to the application.

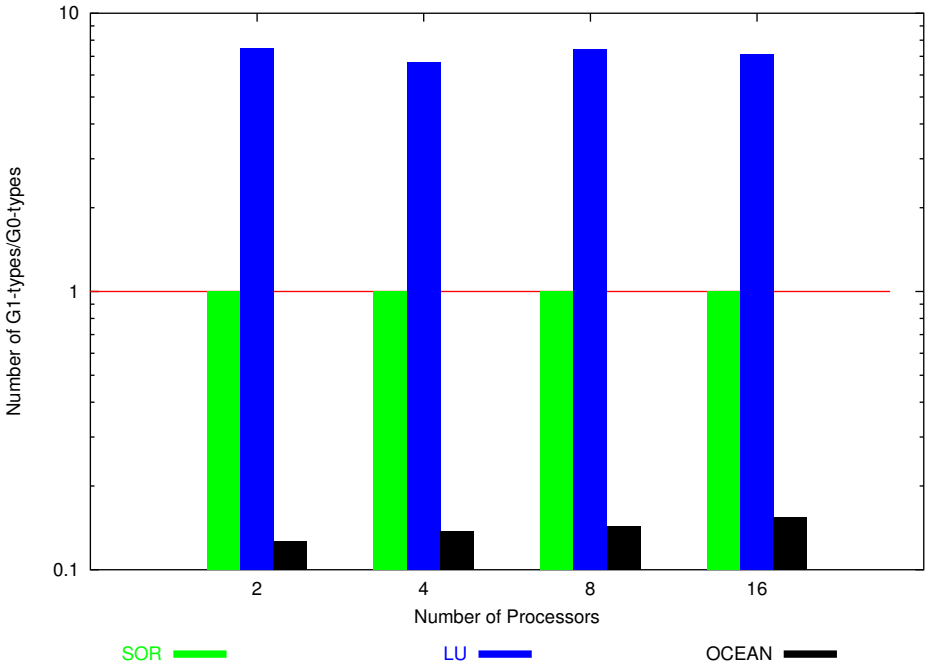


Figure 4.7: Ratio of G1-types to G0-types.

What can be seen from the results in Figure 4.7 is that using only this information LU would benefit from global-1 synchronization, OCEAN from global-0, and SOR is inconclusive. However, when labeling the dependencies as either G0-type or G1-type we do not consider how much different the `ld_dist` is from the `st_dist`. If that load-store difference is small and the load and store are close to the beginning of the session, the estimation may not be accurate. Therefore, in determining the best A-R synchronization method we also take into account this difference. If there are more G1-types than G0-types for an application, and the average load-store difference is small compared to how far the

dependency is from the previous barrier, then we claim that the A-stream has the potential to advance far enough to violate the dependencies anyway. Therefore, in this situation we say the application benefits from global-0 instead. The inverse of this is not very significant, because if the type was already determined as better with global-0, then this claim would mean more dependencies could be broken and that would just reinforce the earlier decision. To represent this information we define a *confidence level* for G1-types and G0-types using Equations 4.1 and 4.2. The results in Figure 4.8 show the confidence level for both G0-types and G1-types for each benchmark.

$$\text{G0-type confidence level} = 1 - \frac{1}{N}\sum_{i=1}^{N}\frac{ld\_dist_i}{st\_dist_i}, \qquad \text{where } N = \text{Number of G0-types} \quad (4.1)$$

$$\text{G1-type confidence level} = 1 - \frac{1}{N}\sum_{i=1}^{N}\frac{st\_dist_i}{ld\_dist_i}, \qquad \text{where } N = \text{Number of G1-types} \quad (4.2)$$

The important information to extract from Figure 4.8 is that the G1 confidence level for LU is fairly low to begin with and drops significantly as the number of processors increase. This would suggest that at higher parallelism the dependencies protected by synchronization events would likely be broken with an A-R synchronization method other than global-1. Therefore, we might change the earlier determination of global-1 to global-0 for larger number processors. However, at this point we have insufficient information to determine at exactly what level this switch should occur.

SOR is an interesting case because the analysis we have shown thus far has not given any indication that one method is better than the other. In fact, it would seem that global-0 and global-1 would perform equally well for SOR. It can be seen from Figure 4.9 that for all but the first session, this result is accurate. For this situation global-0 would not be chosen as the synchronization method, because if they are equally well suited according to the analysis, then choosing a more aggressive would allow for more successful prefetches then global-0. This is especially true in the beginning of the application where cold misses are frequent and the latency could be hidden with a more aggressive method. A validation of this claim is to look at the detailed results for which synchronization method performs the best for each session. Figure 4.9 presents these results for SOR with eight processors.

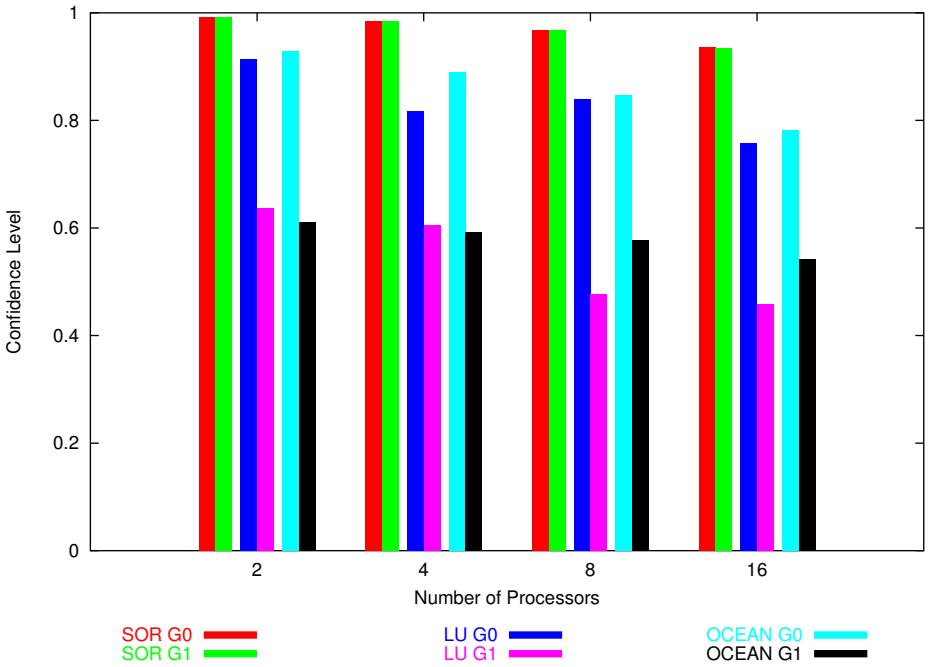


Figure 4.8: G0-type (Equation 4.1) and G1-type (Equation 4.2) confidence levels for SOR (128x128), LU (64x64), and OCEAN (32x32).

We can see that the aggressive methods do significantly better than global-0 only in session one. In the rest of the sessions global-0 and global-1 are almost equal, except the last session in which global-0 performs better than global-1.

SOR is a red-black successive over-relaxation algorithm for solving partial differential equations. Session zero is the first iteration of the red matrix calculation, and session one is the first iteration of the black matrix calculation. Both of these would incur many cold misses, which would benefit more from a global-1 method. The results for session one show this. Because of limitations of the simulation environment, results for session zero could not be obtained, but it is expected that the results would be similar to session one. A summary of the chosen A-R synchronization methods for all the benchmarks is given in Table 4.2. The "=" for SOR indicates that the analysis determined that G0 and G1 performed equally well. The "?" for higher levels of parallelism for LU indicate that the confidence level for those cases made the original determination questionable.

Results that show the speedup of slipstream mode over single mode for all four

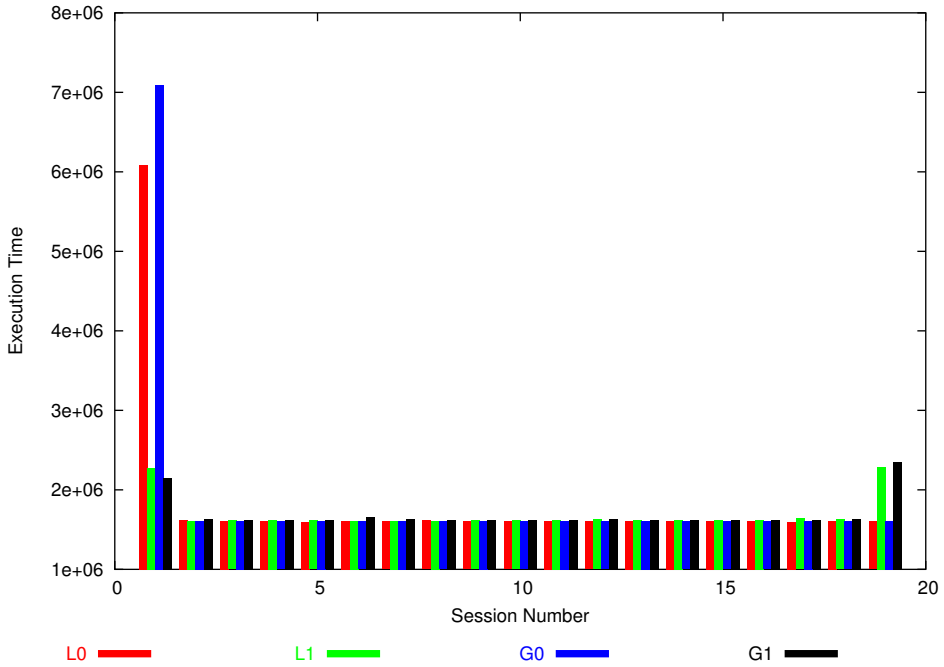


Figure 4.9: Execution times for each sessions of SOR for each A-R synchronization method.

	2 processors	4 processors	8 processors	16 processors
SOR	=	=	=	=
LU	G1	G1	G1?	G1?
OCEAN	G0	G0	G0	G0

Table 4.2: Summary of A-R synchronization methods chosen by profile analysis. An "=" denotes that G0 and G1 were equally well suited for that instance. A "?" denotes that the confidence level made the original determination somewhat questionable.

A-R Synchronization methods for these benchmarks are given in Figures 4.10, 4.11, and 4.12. The estimations for whether global-0 or global-1 would be the most beneficial are mostly correct. The only discrepancies are SOR, which was correct except for initialization, and LU with higher parallelism, which we accounted for with the G1-type confidence level.

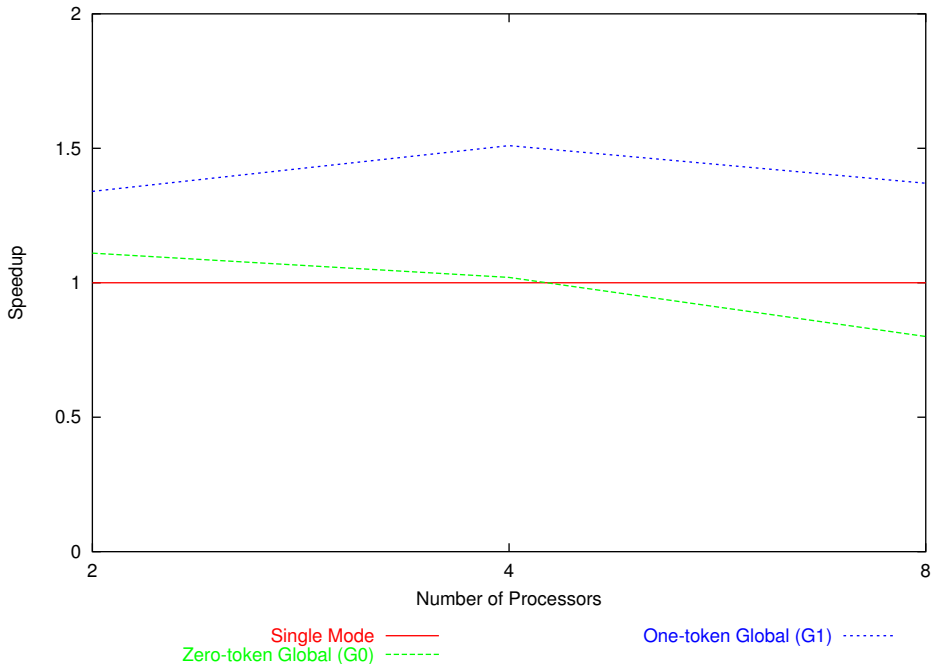


Figure 4.10: Speedup for slipstream mode over single mode with four different A-R synchronization methods for SOR (128x128).

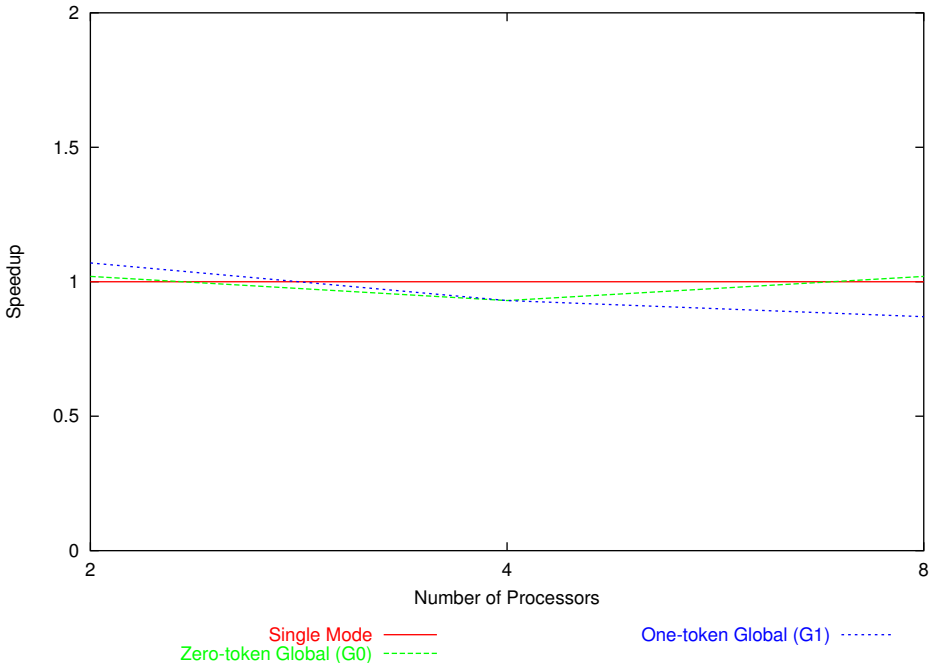


Figure 4.11: Speedup for slipstream mode over single mode with four different A-R synchronization methods for LU (64x64).

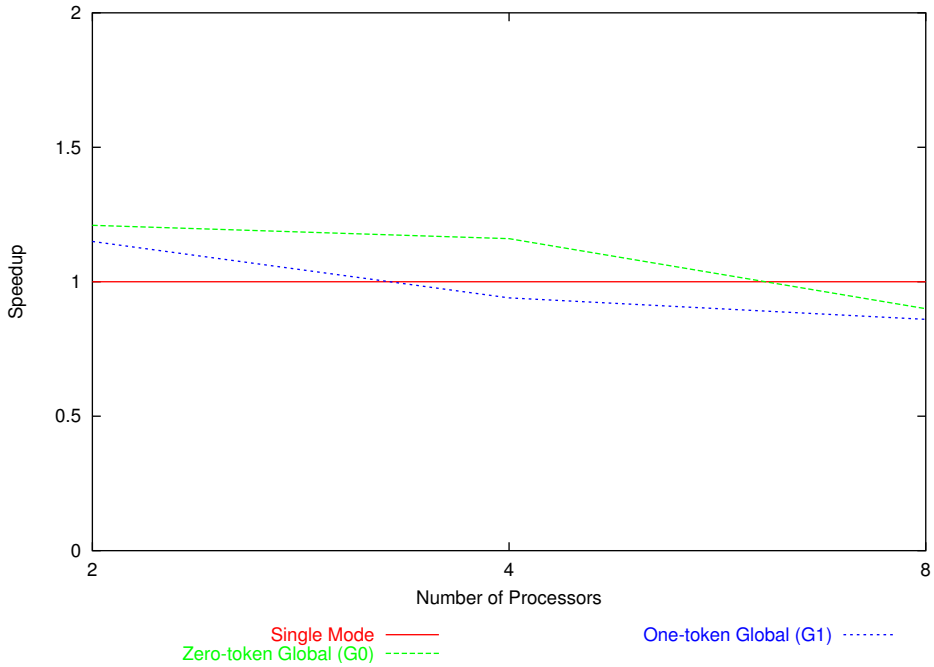


Figure 4.12: Speedup for slipstream mode over single mode with four different A-R synchronization methods for OCEAN (32x32).

Chapter 5

Compiler Integration

In this chapter we describe compiler methods that can be used to determine the A-R synchronization method based on information provided by the profiling tool. One large advantage of compiler analysis over profiling is that it is transparent to both the programmer and the user. Analyzing the application in the compiler also allows for reuse of some of the methods already implemented in modern parallelizing compilers. This, combined with the fact that we can remove the burden of analyzing a trace, can lead to a much more efficient solution.

Compiler analysis has been accomplished for data forwarding [9, 10], as well as data prefetching [5, 11]. As mentioned in Section 3.2, the information necessary to implement data forwarding is very similar to that which is needed for selecting an A-R synchronization method. The information needed for data prefetching is somewhat different, but a few of the same techniques may be helpful for A-R synchronization analysis. Compilers have also been used to improve cache performance by using interprocedural array data-flow analysis to find data access patterns that influence cache coherency [3]. Some of these data access patterns are very similar to what can contribute to a specific A-R synchronization mode.

5.1 Necessary Information

The first step in evaluating compiler analysis for A-R synchronization is to determine what information the compiler needs to provide. This information is similar to what the trace provides in the profile-driven analysis described in Section 4.1. There are a few basic pieces of information that are necessary:

1. Shared memory accesses distinguished from local memory accesses
2. The location of the access (i.e. memory address)
3. The type of shared memory access (i.e. load or store)
4. Which processor accessed the location
5. When the shared memory location is accessed

These pieces of information can be combined and analyzed to provide several key elements relevant to the selection of an A-R synchronization mode.

The first two pieces of information are related. For the compiler to know which addresses are private and which are shared, it can simply look at the scope of the data variable. Local variables are private, whereas global variables are shared. However, this method may become difficult if global pointers are then assigned to local variables. Still, in most cases pre-compiler macros are used to specify which variables are to be allocated as shared. For example, OpenMP is a set of directives that aid in the programming of parallel workloads [8]. OpenMP includes constructs that declare data as either `PRIVATE` or `SHARED`. The information provided by these constructs could be passed to the compiler to aid in the determination of shared memory variables. Once a memory access has been defined as shared and the address is noted, the compiler needs to recognize whether it is a store or a load.

The fourth element leads to the formation of producer-consumer pairs. These pairs make a connection between the processor that writes a shared memory location, and those that subsequently read that location. This is similar to the analysis for definition-use chains that modern compilers use for optimization strategies. For A-R synchronization it is not necessary to determine the exact processors involved in the producer-consumer pair. It is only necessary to ensure that the pair do not consist of memory accesses by the same

processor. The determination of the consumer processor has been accomplished for data forwarding [10]. A similar method may be useful here to ensure the accesses are by different processors.

The last of these elements is analogous to the defined shared memory cycle introduced for the profile tool in Section 4.2. Without the run-time behavior the trace profile-driven analysis can leverage, the compiler has a more difficult time determining *when* the access occurs. What is of importance here is shared memory access patterns. If we limit the study to loop-based scientific codes, which are typical for distributed shared memory systems, the loop iterations can be used to determine the *distance* between the accesses. Previous work on compiler analysis for data prefetching [5, 11, 12], data forwarding [9, 10, 12], and cache coherence [3], has assumed this limitation as well.

If certain access patterns can be identified in the compiler analysis, the reference profile is more structured, which makes it easier to assign an A-R synchronization mode to the application. For example, migratory data is a fairly simple access pattern to recognize. Migratory data is that which is both read and written by a single processor in a sequence of time intervals. In each interval a different processor first reads, and then writes the shared data. Thus, the data is said to migrate from processor to processor [4]. If this migration happens frequently, and each processor writes the data more than once before the next migration, the pattern would benefit the most from a conservative A-R synchronization method because of the many shared memory writes by different processors. An aggressive mode would constantly be removing exclusive access to a data location before it needed to. The processor would have to regain its exclusive state by either invalidating or updating all the shared copies the next time it wrote the data. Access patterns that lead to an aggressive mode would be patterns which involve shared memory write operations that are very distant from subsequent reads. If a processor first writes a memory location then continues to read the same location many times without writing it again, the A-stream of a different processor can effectively prefetch that data for its own R-stream without worrying about it being invalidated in the near future.

5.2 Implementation

After the necessary information has been determined, the next step is to explore methods of collecting and analyzing this information. Data forwarding algorithms provide a basis for this study. The first task of the compiler algorithm for forwarding, which should also work well for A-R synchronization, is to determine the producer-consumer pairs [9, 10].

5.2.1 Producer-Consumer Pairs

Algorithms presented in related work in data forwarding for finding producer-consumer pairs can also be applied here. These pairs can be found in three basic steps [10].

1. Construct modified data flow graph (DFG)
2. Generate the array sections read and written by each node of the DFG
3. Find producer-consumer pairs using classic methods for definition-use chains

The first step is to construct a modified data flow graph (DFG). This data flow graph has nodes that correspond to explicitly serial or parallel sections. The edges in the graph show the flow from one epoch to the next. An example is given in Figure 5.1 that shows a parallel epoch followed by a serial epoch. To incorporate synchronization events into this analysis we start a new parallel epoch for each synchronization event encountered. Therefore, there can be consecutive parallel epochs separated by a synchronization event.

The second step is to examine the ranges of the arrays that are accessed by each node of the DFG. These ranges are further specified by lower bound, upper bound, and stride [10]. Both parallel and serial sections of code are specified by these ranges.

The last step in the algorithm is to use definition-use chains to determine what data is *live* at the entrance to each node in the DFG. This is classified by what data has been defined (written) prior to the node that is used (read) in or after the node. This information can then be combined with the array ranges to determine the producer-consumer pairs [10]. This method only works for dependencies within procedures. For most scientific workloads this analysis is sufficient. However, there have been techniques researched that will expand this algorithm to create DFGs and find definition-use chains using interprocedural array data-flow analysis [3].

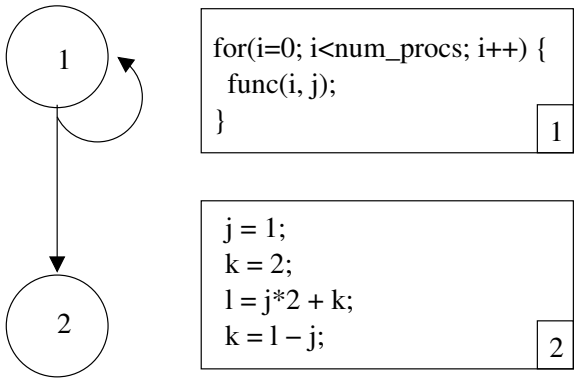


Figure 5.1: An example of a DFG for a code segment

Most of the steps in the algorithm are compiler passes that are already implemented in many modern compilers, but may need modification. For example, similar DFGs and definition-use chains are commonly used for optimizations and scheduling algorithms. The difficult part of compiler analysis for data forwarding is that once the producer-consumer pair is known, the consumer processor has to be specifically identified. This is because the goal of forwarding is to send the shared data to the consumer processor before the processor requests it. For determining the A-R synchronization mode, only the access pattern is of concern, not the particular processor that participates in the access, other than the determination that the participating processors are either the same or different. Therefore, for this analysis the exact identification of the consumer processor is unnecessary.

5.2.2 Access Patterns

After the producer-consumer pairs are determined, the next step in the compiler analysis is to identify access patterns that would lead to a specific synchronization mode. A full study would need to be conducted to determine exactly which access patterns would be best suited by which form of synchronization. That study is beyond the scope of this thesis. However, general suggestions concerning access patterns can be made.

One pattern that has been studied for data forwarding concerns the distance (in

execution cycles) between the producer and the consumer [5]. This distance can also be used for selecting an A-R synchronization method. For example, if the average distance between the producer of data and the consumer of that same data is large, then a more aggressive synchronization mode may be preferred. This is because the A-stream is likely to prefetch the data early enough for the R-stream to benefit, though it may cause the producer processor to lose its exclusive copy. Several other access patterns may lead to a conservative mode (e.g. migratory data).

Another access pattern that may be beneficial for the compiler to detect is one that is analogous to the RAW dependency method described for the profiling tool in Section 4.3. The access pattern is similar to a *stale reference sequence* as described in [3]. This work attempts to identify and invalidate cached data that is stale before there is a cache miss. The reference pattern is simply a write to a shared data element, followed by one or more synchronization events, and a read to the same data element by a different processor. Similar techniques can be employed to find this pattern for A-R synchronization method determination.

An access pattern that stems from our research in the trace-driven profiling tool concerns not only how large the RAW dependency distance is, but also where the references are situated within a parallel session. An example of this pattern is a write at the end of one session, and a read at the beginning of the next. An aggressive synchronization mode in this case would lead to premature migration of that data, and would degrade performance. Whereas, if the write was situated at the beginning of a session, and the dependent read was at the end of the next session, an A-stream could safely skip the synchronization event with confidence that the other processor's R-stream would write the shared data before the A-stream's read would occur.

Chapter 6

Related Work

This thesis work builds on the original slipstream paradigm, a hardware mechanism to improve both fault tolerance and performance on processors with multiple contexts executing simultaneously [15]. Slipstream execution mode for multiprocessors was introduced as a variant of the slipstream paradigm specifically geared towards distributed shared memory systems consisting of dual-processor single-chip multiprocessors [7]. In that work a single A-R synchronization method was specified at run-time and used throughout the execution of the program.

Later research introduced an implementation of a dynamic method for switching the A-R synchronization mode during the execution of a program to account for profile behavior of applications [13]. The goal here was to determine if dynamically switched the mode could provide a performance benefit. The work has the limitation of the need for an initial profiling run to determine the barrier-to-barrier execution time for each A-R synchronization mode in each session. Also, there is overhead involved with the dynamic switching, which was mostly overcome with a sliding window switching method. This adaptation limited the switching frequency to reduce any thrashing between synchronization methods.

In this thesis we also provide a way to determine the A-R synchronization method. However, we differ from previous work in that we discuss a profiling pass and a compiler option to statically determine the method based on shared memory access patterns. In our method the same synchronization mode is used throughout the entire execution of the

application and therefore does not take into account the bahavior of distinct computational phases within a program.

Trace-based profiling is a common approach to determine patterns in memory access. It has been used to develop and model memory systems for many years [14]. Trace-driven simulators are often difficult to use because a full memory trace of a scientific workload can result in file sizes over one gigabyte. Also, these simulators are often very slow because of the amount of data that must be analyzed. More recent research looks into three aspects of trace-driven memory simulation: trace collection, trace reduction, and trace processing [16]. For simplicity we countered the difficulties with traces by reducing the data set sizes of the benchmarks used.

We base our compiler analysis techniques on those done for data forwarding [9, 10, 12] as well as data prefetching [5, 11]. Our work differs from these in that the end result is to determine shared memory access patterns that lead to a specific A-R synchronization mode. Similar techniques have also been researched for cache coherence and performance [3]. In that work interprocedural array data-flow analysis is used to find a *stale reference sequence* consisting of a write, followed by one or more session boundaries, and then a read to the same data. Stale references are very similar to the RAW dependencies we calculate. Our work differs from theirs in how we use this reference pattern after it is detected.

Chapter 7

Conclusions and Future Work

This study showed that static code analysis to select an A-R synchronization mode is a solution to some of the current limitations of A-R synchronization. An introduction to slipstream was given and the problems of a user-selectable synchronization method were discussed. Different techniques for determining the A-R synchronization method were explored. A trace profile framework was presented that requires only simple changes to the application, and is flexible to facilitate further study.

A simple algorithm for finding RAW dependencies was detailed. We improved on this algorithm and gave results that showed a significant speedup over the simple algorithm for two benchmarks. A shared access pattern detection method was introduced that can be used to determine if a conservative A-R synchronization method would be the best choice for the given application. We present results that show that the detection method was correct on two of the three benchmarks. The method for the third benchmark was correct for every session except for the first two where run-time dynamic behavior dramatically affected the result.

We showed that compiler analysis provides the reference information necessary to select an A-R synchronization mode without requiring any further effort by the programmer or the user. However, the access patterns that are necessary to successfully predict the correct synchronization method are left for further work.

The limitations that bounded this study could be expanded to include a separate analysis to determine global verses local A-R synchronization mode. An in-depth study

of the effects of different shared memory access patterns could give better insight as to how the access patterns and the synchronization mode relate. The analysis can also be expanded to include the affect the synchronization mode has on other optimizations possible in slipstream, such as self-invalidation and barrier speculation.

Another area of future work is to implement a fully automatic selection of A-R synchronization. This may include modifying the compilation process for applications to use slipstream execution mode for inclusion of the compiler analysis previously described. This modification could be simply modifying the current compiler algorithms used for optimization to provide the necessary reference data, or instead adding new compiler passes to parallelizing compilers, such as SUIF[1] or POLARIS[2], to provide this data.

The compiler analysis proposed to implement this automatic selection is similar to that which has been done for data forwarding. Therefore, it may be possible to expand the algorithms for A-R synchronization slightly to include all the information necessary to implement data forwarding. Most implementations of data forwarding do not include a large amount of additional hardware support [9, 10, 12]. Consequently, it may be feasible to implement a version of data forwarding as a part of slipstream execution mode.

[1] http://suif.stanford.edu
[2] http://polaris.cs.uiuc.edu/polaris/polaris-old.html

Bibliography

[1] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. M. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* Philadalphia: Society for Industrial and Applied Mathematics. Also available as postscript file on http://www.netlib.orgtemplatesTemplates.html, 1994.

[2] A. Brandt. Multi-level adaptive solutions to boundary-value problems. In *Mathematics of Computation*, 1977.

[3] L. Choi and P.-C. Yew. Compiler analysis of cache coherence: Interprocedural array data-flow analysis and its impact on cache performance. In *IEEE Transactions on Parallel and Distributed Systems*, September 2000.

[4] A. L. Cox and R. J. Fowler. Adaptive cache coherency for detecting migratory shared data. In *International Symposium on Computer Architecture*, 1993.

[5] E. H. Gornish, E. D. Granston, and A. V. Veidenbaum. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *International Conference on Supercomputing*, 1990.

[6] K. Z. Ibrahim. *Slipstream Execution Mode for CMP-Based Shared Memory Systems.* PhD thesis, North Carolina State University, 2003.

[7] K. Z. Ibrahim, G. T. Byrd, and E. Rotenberg. Slipstream execution mode for CMP-based multiprocessors. In *International Symposium on High Performance Computer Architecture*, February 2003.

[8] H. Jin, M. Frumkin, and J. Yan. The OpenMP implementation of NAS parallel benchmarks and its performance. Technical Report NAS-99-011, NASA Ames Research Center, 1999.

[9] D. Koufaty, K. Chen, D. K. Poulsen, and J. Torrellas. Data forwarding in scalable shared-memory multiprocessors. In *IEEE Transactions on Parallel and Distributed Systems*, December 1996.

[10] D. Koufaty and J. Torrellas. Compiler support for data forwarding in scalable shared-memory multiprocessors. In *International Conference on Parallel Processing*, 1999.

[11] R. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.

[12] D. K. Poulsen and P.-C. Yew. Data prefetching and data forwarding in shared memory multiprocessors. In *International Conference on Parallel Processing*, 1994.

[13] S. Sivagnanam. Implementation of dynamic synchronization for slipstream multiprocessors. Master's thesis, North Carolina State University, 2003.

[14] A. J. Smith. Cache memories. In *ACM Computing Surveys*, 1982.

[15] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream processors: Improving both performance and fault tolerance. In *9th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[16] R. A. Uhlig and T. N. Mudge. Trace-driven memory simulation: A survey. In *ACM Computing Surveys 29*, 1997.

[17] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22th International Symposium on Computer Architecture*, 1995.