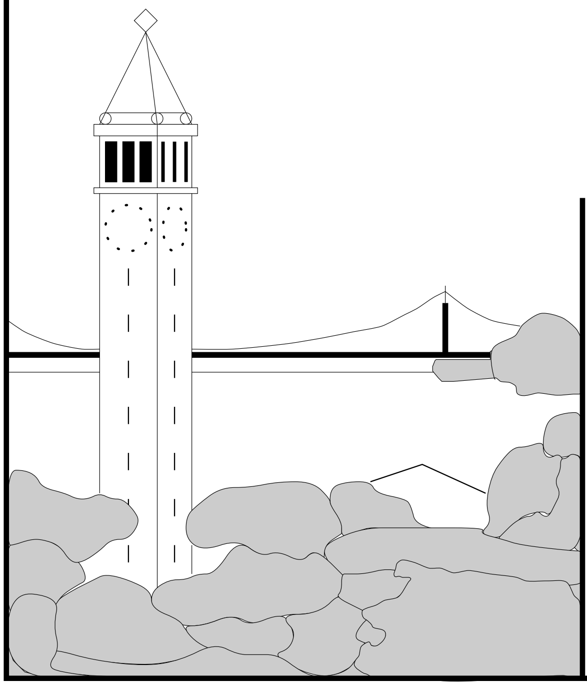
Optimizing Explicitly Parallel Programs

Arvind Krishnamurthy



Report No. UCB/CSD-94-835

September 1994

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Optimizing Explicitly Parallel Programs

Arvind Krishnamurthy

Abstract

We present compiler optimization techniques for explicitly parallel programs that communicate through a shared address space. The source programs are written in a single program multiple data (SPMD) style, and our machine target is a multiprocessor with physically distributed memory and hardware or software support for a single address space. The source language involves normal read and write operations on the address space, which correspond either to local memory operations or to communication over an interconnect network.

The remote operations result in high latencies, but much of the latency can be overlapped with local computation or initiation of further remote operations. Non-blocking memory operations allow this overlap to be expressed directly. However, overlap is difficult for programmers to do by hand; it can lead to subtle program errors, since the order in which operations complete is no longer obvious. Programmers writing explicitly parallel code expect reads and writes from a single thread to take effect in program order, a property called sequential consistency. The use of non-blocking memory operations might yield executions that violate sequential consistency.

We provide a new algorithm for static parallel program analysis to detect memory operations that can safely be made non-blocking. The analysis requires dependency information across and within threads, and builds on earlier work by Shasha and Snir. We improve their results by providing a more efficient algorithm for SPMD programs, and by improving the accuracy of the analysis through the use of synchronization information. Using the results of this analysis, we show how to optimize parallel programs by changing blocking operations into non-blocking ones, performing code motion to increase the time for communication overlap, and caching remote values to eliminate some read accesses entirely.

We show the potential payoff from each of our optimizations on real applications, using hand-transformed programs. The experiments are done on a CM-5 multiprocessor using the Split-C runtime system, which provides a software implementation of a global address space and both blocking and non-blocking memory operations.

Contents

1 Introduction

Traditionally, there have been two different approaches towards programming parallel machines: the parallelizing compiler approach and the explicitly parallel program approach.

In the parallelizing compiler approach, the program is written in a sequential language (or an implicitly parallel language), and the parallelism in the program is identified by the compiler. Parallelizing compilers have been built for imperative languages like Fortran [1] and data-flow languages like Id [2]. This approach requires little effort on behalf of the programmer and allows reuse of existing code (popularly known as parallelizing dusty-decks). However, the parallelizing compiler approach rarely discovers coarse grained parallelism, and is also constrained by the need for good dependence analysis.

The other approach is to write explicitly parallel programs using a language with constructs for creating and synchronizing parallel threads. The programmer does not rely on the compiler to detect the parallelism. Instead, the programmer makes all the important decisions regarding scheduling, data layout, and synchronization, and the compiler deals only with the lower level details of code generation. In this approach, sequential programs cannot be reused, and there is a startup cost for the programmer to learn the parallel semantics of the language. However, the programmer can use algorithms designed specifically for parallel systems and can make use of information about an application that is not available to the compiler. This usually leads to programs that obtain better performance on parallel machines. Therefore, in spite of the additional programming effort, programmers typically choose explicitly parallel languages for writing parallel applications.

Our research focuses on developing optimizing compilers for explicitly parallel programs. In particular, we are interested in optimizing parallel programs that have been written in a Single Program Multiple Data (SPMD) style as opposed to data-parallel programs. In both programming styles, there are multiple threads running on different processors. However, in an SPMD program, the threads can execute asynchronously and need not compute in a lock-step fashion as is the case with data parallel programs. Asynchronous execution enlarges the set of algorithms that can be executed efficiently on these parallel machines. An SPMD model provides maximum control to the programmer in the parallelization effort. However, the programming task required for writing SPMD programs is exacting. We relieve the programmer from having to specify certain low-level aspects of a program's execution by providing compiler optimizations for scheduling certain instructions that are specific to distributed memory machines.

We address the problem of implementing a global address space abstraction on a distributed memory machine. Why do we care about global address space abstraction on distributed memory machines? Parallel computing on distributed memory multiprocessors is popular due to scalability considerations. Most of the high performance machines that are currently being marketed are distributed memory machines, the principal argument being that to obtain teraflop performance we would need a machine that has at least a 1000 processors, and a machine that size is attainable only if the memory is distributed across the machine. Each node in such a machine typically consists of a tightly coupled processor-memory pair.

Given that distributed memory machines are necessary to obtain scalable performance, the question is which programming model allows the user to obtain good performance with a reasonable amount of programming effort. A global address space abstraction allows the programmer to extend standard uniprocessor data structures into distributed data structures [19] [5] that live on multiple processors. The ability to name objects that live on other processors' memory is of significant utility to the programmer. When an object belonging to the global address space is accessed, it is either fetched from local memory or is fetched from remote memory in a manner that is transparent to the programmer. This greatly increases the programming ease.

The principal drawback of using distributed memory machines is the loose coupling of processors. Only a small portion of the global address space is physically close to any given processor. Accesses to the other memory modules (which will henceforth be called remote accesses) are implemented using messages through an interconnect network and are thus high latency operations. The cost of these operations vary from 100 cycles on the T3D[16] to 400 cycles on the CM5[18], with the requesting processor remaining idle for a significant portion of the time.

The high latencies associated with remote memory operations can be tolerated by using certain well-known techniques. *Message pipelining* and *prefetching* are techniques that can potentially hide the associated cost. Pipelining allows a given message latency to be masked by other message latencies. The processor treats the network as a resource that can be pipelined, and allows multiple outstanding messages. Prefetching increases the number of machine instructions between the point at which the read is initiated and the point at which the value is actually used. The intervening instructions could be executed by the processor, and can therefore mask the latency of the global access. *Multithreading* could be considered as a variant of prefetching, where the latency of a thread's remote access is overlapped with the computation of another thread. Another popular way to handle the cost of remote accesses is to cache values. *Caching* eliminates multiple accesses to the same memory location, thus reducing the number of remote accesses and improving the performance of the application code.

There are no automatic tools for incorporating these optimizations in parallel applications that are written in a general model, such as the SPMD model. The programmer either has to code these optimizations explicitly or has to write programs that conform to certain models like proper labeling for release consistency machines[8]. The first alternative requires enormous programming effort. In fact, the designers of the ScaLapack library made a design choice not to pipeline messages in their highly optimized linear algebra kernels [6] due to the programming effort required. The second approach requires the programmer to stick to certain conventions and to classify the memory accesses into different categories; this can be equally taxing to the programmer. These difficulties motivate the need for techniques that can incorporate optimizations that hide remote memory latencies.

Performance benefits in the order of 20-50% can be obtained by introducing the latency masking optimizations [5]. This usually corresponds to a considerable increase in processor utilization. These optimizations have significantly higher benefits than that can be obtained from standard back-end optimizations such as register allocation and instruction scheduling.

In this report, we present methods for automating optimizations such as message pipelining and caching. In section 2, we describe the source language, and in section 3, we describe the target language. We present some basic terminology in section 4, and in section 5, we present an overview of the compilation process. In section 6, we describe the theory developed by Sasha and Snir [17] that helps the compiler in deciding valid usage of these optimizations. We extend their algorithm to handle SPMD programs in section 7, and show that our algorithm is more efficient. In section 8, we discuss the issues involved in code generation. In section 9, we discuss new algorithms that incorporate synchronization analysis techniques to further enlarge the scope of these techniques. In section 10, we extend these techniques to allow compiler controlled caching of values. Section 11 quantifies the potential payoffs by analyzing some application kernels. Related work is surveyed in section 12, and general conclusions drawn in section 13.

Our primary contributions in this research are:

1. A fast polynomial time algorithm for deciding when accesses can be reordered or pipelined in an SPMD program.

2. Enlarged applicability of the optimizations using extra program information obtained from synchronization analysis.

3. Techniques for code generation where the target abstract machine is the *Split-C* programming language[5].

4. Extensions of the theory to other transformations such as caching and common subexpression elimination.

2 Programming Model

We analyze explicitly parallel programs written in an SPMD style. In this style, the same program executes on every processor, but the execution paths through the code image may vary and processors may execute the same statement at different times. The processors access a global address space, which is either supported in hardware or is provided through software mechanisms.

The programmer can declare globally accessible regions of memory using the keyword `shared`. The shared regions could be separate C objects or could be an array of C objects. The different threads of control can access these regions using normal read and write operations.

The threads can synchronize using three different mechanisms. The `barrier` provides a global synchronization for all the threads. The `lock` and `unlock` operations allow the threads to execute mutually exclusive code. The `post` and `wait` calls provide event-based synchronization that could be used for a producer-consumer style of programming.

A sample program is shown in Figure 1. The variables $flag$ and $result$ belong to the global address space, and hence are accessible to all the processors. The variables i and sum are standard C *local* variables that are accessible only to the local processor. The entry point is $main$, and the processors start executing code from this common entry point. However, they could execute different code (as is the case in the example where processor 0 executes code that is never executed by other processors). In this example, there is an event-based synchronization point using the variable $flag$. Upon completion of its task, processor 0 sets the flag by doing a `post` operation. The other processors wait for the flag to be set by executing a `wait` operation. On completion of the wait, they use $result$ to compute local values of sum. The `barrier` synchronization at the end ensures that all processors have reached the same state in their execution.

```
shared event flag;
shared int result[10];

main() {
   int sum = 0;
   int i;

   if (MYPROC == 0) {
      for (i=0; i<10; i++)
         result[i] = i;
      post(flag);
   }
   else {
      wait(flag);
      for (i=0; i<10; i++)
         sum += result[i];
   }
   barrier();
}
```

Figure 1: Shared Memory SPMD Program

3 Target Language

Our target language is Split-C [5], a language for programming distributed memory machines. Split-C provides a global address space abstraction by extending C pointers and C arrays. Two kinds of pointers are provided, reflecting the cost difference between local and global accesses. *Global pointers* reference the entire address space, while standard pointers reference only the portion owned by the accessing processor. Global pointers are declared using a new qualifier `global`. A global pointer can be dereferenced in the same manner as a standard C pointer. For a global access, a local/remote check is involved, and if the object is remote, a dereference incurs the additional cost of communication.

Split-C also provides a simple extension to the C array declaration to specify *spread arrays*, which are spread across the entire machine. The declaration specifies the layout of the array by specifying the block size of the array on each processor. The blocks are always cyclically mapped starting with the block numbered 0 on processor 0.

To compile *shared* variable declarations of our source language, we arbitrarily choose the processor that is going to own the object, and handle variable accesses by using global pointer dereferences. The *shared array* constructs are directly mapped onto the *spread array* declarations of Split-C.

The most important feature of the Split-C language is that it allows efficient access to the underlying machine by providing *split-phase* operations on the global address space. This addresses the concern that the issuing processor is idle during most of the time required to satisfy a remote request. Instead, the processor is allowed to continue with computation until the completion of the access is essential. This technique may be used for both read and write operations. For a read operation, the processor initiates the request, but does not wait for the value's arrival; it may then execute statements that do not depend on the value. For a write operation, the processor initiates the operation either by pushing the value to be written into the network or by initiating a DMA operation and resumes its local computation; it does not wait for the acknowledgement that the remote memory module has received the value.

The primary idea is that any remote access has two phases. The first phase initiates the operation. The second phase synchronizes or joins the thread of control with the remote access by waiting for its completion. These operations are hence called split-phase operations. The asynchronous read operation is called a *get* operation, while its write counterpart is called a *put* operation.

We now describe the primitive operators of Split-C. A remote access has five attributes, the first of which specifies the type of the operation: get or put. To completely specify a read operation, the physical address of the remote object, the local address for storing the fetched value, the type of the object being fetched (which includes the object's size), and the counter (or flag) that needs to be set on completion of the fetch are required. A write operation can be initiated by specifying the value to be stored, the target remote address, the type of the object, and the local counter that is to be updated on receipt of the acknowledgement.

Let us look at two examples of these split-phase operations defined by the Split-C runtime system.

```
void d_get_ctr (double *dest, double *global src, Counter *ctr);

void d_put_ctr (double *global dest, double *src, Counter *ctr);
```

The type and the nature of the operation (get or put) is explicit in the name of the function. The arguments to the functions include the address of the globally accessible object (represented as a global pointer), the local data (represented as a local pointer), and the counter that will be used for synchronization. The program can wait for all the accesses that have been initiated using a particular counter by doing a `sync_ctr` operation on the counter. This is convenient when a set of accesses are related and their completion is required at the same time. The following piece of code computes the sum of two non-local values efficiently by overlapping the latencies of the remote accesses.

```
d_get_ctr (&dest1, src1, ctr);
d_get_ctr (&dest2, src2, ctr);
/* Unrelated computation */
sync_ctr (ctr);
foo = dest1 + dest2;
```

The get-put operations are library functions that are provided by the Split-C system. Constrained versions of these functions (versions that do not allow the user to specify the synchronization counter) are available through syntactic extensions to the C language in the form of new assignment operators. Sample use of the new assignment operator is shown below. The assignment operator := is overloaded for both *get* and *put* operations, and `sync` waits for the completion of all accesses that have been initiated using the assignment operator. The above program can be written as:

```
*dest1 := *src1;
*dest2 := *src2;
/* Unrelated computation */
sync();
foo = *dest1 + *dest2;
```

The flexibility provided by the library functions provided by Split-C is the ability to wait only on a select set of outstanding requests. This is achieved by using different counters for different sets of accesses. This flexibility is useful when the processor works on two different tasks with the compute phase of one task overlapping the communication phase of the other.

Split-C also provides a `store` operation that is a variant of the `put` operation. A `store` operation generates a write to a remote memory location, but does not acknowledge when the write operation completes. It exposes the efficiency of one-way communication in those cases where the communication pattern is well understood. The language provides a new assignment operator $:-$ for specifying `store` operations. The synchronization operation for `store`s is the `all_store_sync` operation that waits for all `store`s on alll processors to complete. `all_store_sync` is therefore a global synchronization operation. A common optimization for Split-C programs is to transform a set of `put` operations followed by a `barrier` synchronization into `store` operations followed by a `all_store_sync` operation.

On the CM5, these operations are provided through software emulation of a shared memory system. On the CRAY T3D, there is greater hardware support for memory-to-memory transfers. On some Intel machines, the software handles just the initiation protocol with the rest of the work borne by the DMA controllers. Split-C runs on top of Active Messages on the CM5, and there are prototype implementations for the Paragon, SP-1, and a workstation network [13]. It defines a portability layer with fast, non-blocking remote accesses that, unlike large message passing systems, can be implemented without message buffering on both ends [18]. It blurs the distinction between machines with a hardware global address space and those without, making it a good choice for an abstract machine language.

4 Basic Terminology

The intermediate representation that we use in the compilation process is a modified control flow graph. Our analysis techniques are presented as transformations on graphs. Therefore, our presentation includes a number of graph theoretic concepts. In this section, we introduce these terms and also describe how explicitly parallel programs are analyzed to obtain the kind of graphs required by our compiler tools.

Let G be a directed graph on the set of vertices V connected by the set of edges E. We denote such a graph by the tuple (V, E). We use the notation $[u, v]$ to denote a directed edge from u to v, and (u, v) to denote an undirected edge. For a graph (V, E), the transitive closure is a new graph (V, E') where $[u, v] \in E'$ if there is a simple path from u to v in the original graph. A partial order is the transitive closure of a directed acyclic graph or DAG.

We analyze explicitly parallel programs specified by a fixed number of program segments, each of which is executed on a single processor. Each program segment defines a total order P_i on the accesses issued by the i^{th} processor. The union of these P_i's is called the program order, P. P is a partial order since it is the union of several disjoint total orders. In any execution, there is a total order E_v on the accesses issued to variable v. The union of these E_v's is the execution order, E, which is also a partial order. Figure 2 illustrates these concepts., with solid arrows for P edges and dashed arrows for E edges.

We have defined P and E for a particular execution of the program. However, we would like

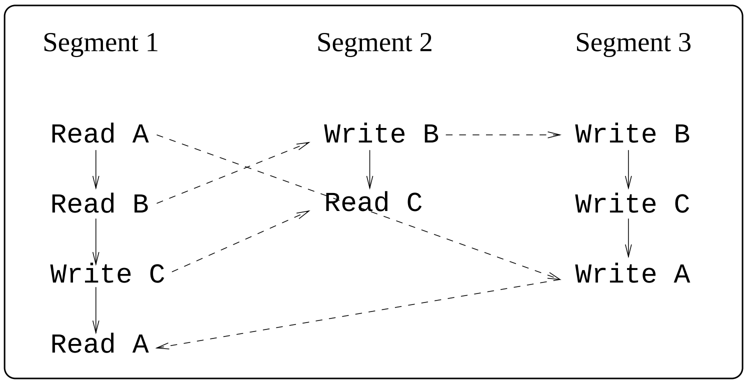


Figure 2: P and E for a particular execution

```
                    shared int flag;
                    shared int data;
       Processor A                          Processor B

for (i=0; i<10; i++)                  while (1)
    local += local/10;                    if (flag == 1)
data = local;                                 break;
flag = 1;                             for (i=0; i<10; i++)
                                          local = (local + data)/10;
```

Figure 3: Sample program

our analysis to work with the compile-time representation of programs. This means that we need to approximate the P and E orders, and use these approximations for generating optimized code.

We can approximate P by the control flow graph (CFG) of the program segment. Since there are multiple execution paths in the control flow graph, P is no longer a total order on accesses. Also, the notion of an access is replaced by that of an *access instruction* that could initiate multiple accesses to a particular memory location during the execution of the program. At compile time, the runtime ordering of accesses to a variable also is not known. Hence, we approximate E by the undirected version of the E edges, which are called the conflict edges, C. To ensure correctness of our program analyses, we require E to be contained in C. This condition is trivially satisfied by making C contain bidirectional versions of the E edges. In section 9, we discuss improvements to our analysis that make a better approximation of E by studying the synchronization constructs in the program.

To illustrate these terms, Figure 3 shows a sample parallel program made up of two processes accessing a common set of variables. The corresponding program graph is given in Figure 4. The conflict edges are represented using dashed lines and are bidirectional. The other edges specify the processor dependencies. Note that we ignore all the accesses to local data in the graph, and if an expression or statement involves multiple global accesses, the evaluation rules of the language (left to right) specifies the ordering of the accesses in the graph.

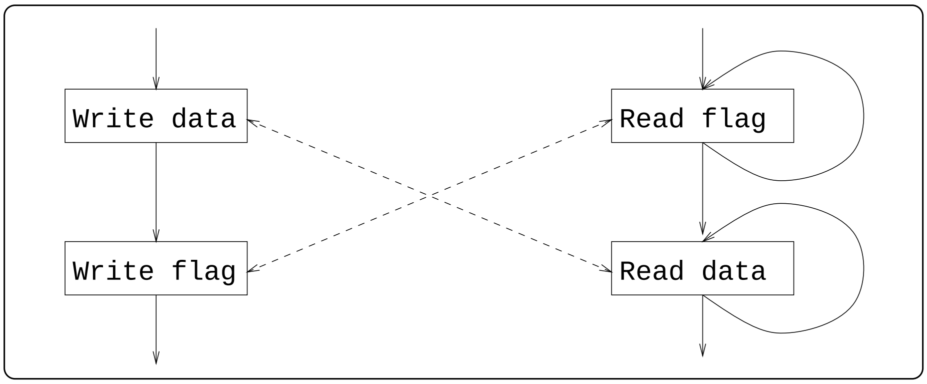


Figure 4: Conflict edges and control flow graph for sample program

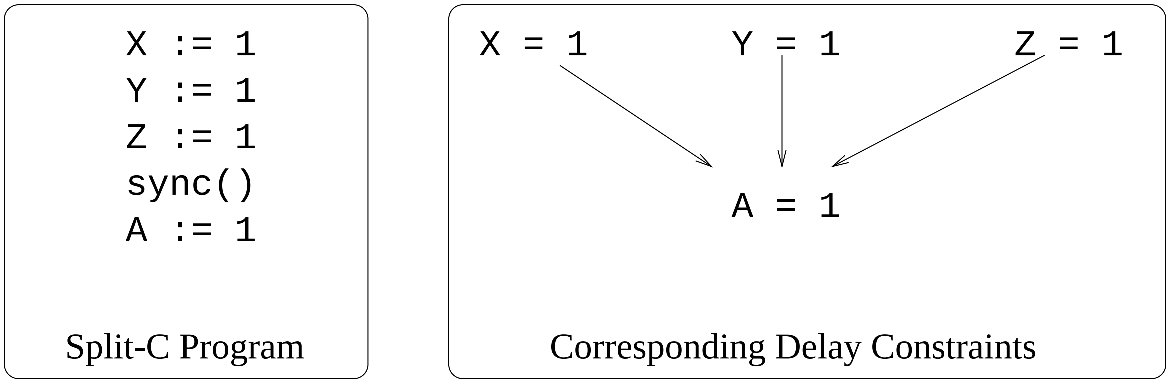


Figure 5: Delays imposed by Split-C code

5 Overview of the Compilation Process

The compiler is a source level transformer. The input to the compiler is a shared memory C program. The compiler generates Split-C code as output. The compile-time analysis is to figure out the schedule for completion of remote accesses. The intermediate representation built by the compiler is the *delay set*, D. This set specifies the restrictions on the completion of remote accesses. The compiler has to ensure that the delay set imposes sufficient restrictions for correct execution[1] of the program. The compiler generates Split-C code based on the computed delay set. The remote accesses are transformed into split-phase operations, and the `sync_ctr` operations are placed at appropriate program points to enforce the delay constraints. Some sample delay constraints and the corresponding Split-C code is shown in Figure 5.

The compiler expects the Split-C system to stick to the delay restrictions imposed by the `sync_ctr` operations. The `put_ctr`, `get_ctr`, and `sync_ctr` operations are provided by the Split-C environment. Other machines or software systems may provide different mechanisms. For example, the DASH multiprocessor[12] provides a *fence* instruction that is similar to Split-C's `sync` operation, but does not allow the flexibility of separate counters. More generally, the system is provided with a set of delays, and the program's execution should be consistent with the delay constraints. In

[1] We will describe our notion of correctness in the next section

other words, the system provides the following contract:

System Contract: $D \cup E$ is acyclic.

A cycle in $D \cup E$ would imply that the system did not adhere to some delay constraint. In the Split-C system, it would imply that some `sync_ctr` operation in the program was faulty and did not wait for the corresponding access to complete.

In summary, the compiler's task is to compute a delay set that would ensure correct execution of the program. The Split-C system's task is to enforce the delay restrictions specified by the compiler using `sync_ctr` operations.

6 Shasha and Snir's Algorithm

In this section, we analyze a correctness criteria for execution of parallel programs called *sequential consistency*. We will also study how this criteria could be violated by indiscriminate use of pipelining optimizations and how the program order on accesses can expose inconsistencies in the execution of the program. Equipped with a clear understanding of how the correctness criteria might be violated, we can decide the extent to which the constraints on the completion of remote accesses can be relaxed.

6.1 Sequential Consistency

Lamport defines sequential consistency as follows[11]:

> A system is *sequentially consistent* if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

A trivial way to satisfy sequential consistency would be for each processor to initiate an instruction only after the preceding operation has taken effect. In the case of remote accesses, this means waiting for a round trip for both read and write operations. Less strict restrictions on the completion of memory operations would lead to greater processor utilization.

To reiterate, sequential consistency is violated if the observed behavior of the program implies that certain accesses were completed in an order that is inconsistent with the program order: an access issued by a processor takes effect before an access issued earlier by the same processor. For the observer, sequential consistency is violated if the observed behavior cannot be reconciled with the program order.

In the program shown in Figure 3, sequential consistency could be violated if the write to $flag$ happened before the write to $data$. The other processor might obtain the old value of $data$ if the read accesses to $flag$ and $data$ take place before $data$ is updated. This would result in incorrect program executions. Sequential consistency is the correctness criterion implicitly assumed when writing parallel programs, and it must be respected when introducing code motion, common subexpression elimination, or pipelining optimizations.

We now elaborate on the relationship between program structure and validity of certain program transformations. The key idea is that accesses which do not take effect in program order do not necessarily lead to observable violation of program order; such a violation can be detected only for certain access sequences of the program segments. In this section, we develop a characterization for the structure of program segments that exposes program order violations.

In this section, we will study what makes a particular execution of a parallel program violate sequentially consistency. We will characterize the program structure that leads to sequentially inconsistent behavior at runtime, and develop an algorithm based on this characterization.

6.2 Violations of Sequential Consistency

We now discuss ways of detecting violations of the sequential consistency model for a particular execution of the parallel program. This will motivate the compiler analysis, which will allow any program transformation that does not permit such executions.

The observer can detect violations of sequential consistency either through local behavior of a program segment or by not being able to reconcile observed behavior of different program segments with the expected program order. Let us first examine how the local behavior could be used to infer such violations.

The simplest violation occurs when accesses to the same variable by a single program segment get reordered. If a later write operation overtakes an earlier read or write operation or if a later read operation overtakes an earlier write operation, the program might enter an inconsistent state. Designers of superscalar pipelined processors have to handle this problem of out-of-order execution of memory operations to the same memory location. In these processors, the hardware detects of these race conditions and delays certain operations to ensure proper execution. However, if the global address space is provided by a software layer, the onus is on the compiler to detect these instances (or make conservative estimates in the presence of aliasing or array references) and introduce the necessary delays. This is a task that is required for doing code motion for sequential languages, and reasonable solutions exist.

Inferring violations of sequential consistency from the behavior of multiple program segments requires constructing a sequence of accesses that exposes a violation of program order. An *access sequence* is an ordering of accesses executed by two or more processors, where the given ordering is necessary to explain the program's behavior. In Figure 3, if the read operation on *data* provides the old value of *data*, this would indicate the presence of the access sequence: write $flag$, read $flag$, read *data*, write *data*. This access sequence would indicate that the program order was violated in the ordering of writes to *data* and $flag$.

To construct an access sequence involving two accesses, the observer can employ two kinds of basic information. The first type is the ordering of instructions executed by a given processor. The second type of information is the ordering of operations of different program segments that take place on a particular variable. While it is unclear how the observer can obtain an ordering of accesses to a particular variable, we note that the program order P introduced in an earlier section can be used to order accesses from a given processor.

Let us now study how accesses to the same variable might be ordered. The observed behavior of accesses implicitly specifies an ordering on global accesses on a given variable. Let a_1 and a_2 be accesses to the same shared variable, and let one of the operations, say a_1, be a write operation. If a_2 is a read operation, the value read might indicate whether the write has taken place earlier if the read operation sees the new value. If a_2 is a write operation, a subsequent read operation

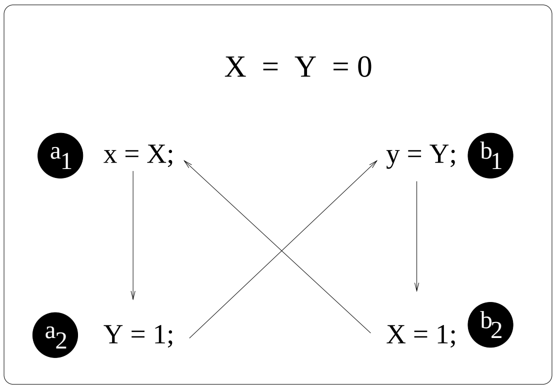


Figure 6: Cycle in $P \cup E$

would indicate which one of the writes was performed earlier. However, if both a_1 and a_2 are read operations, we cannot order the accesses in the absence of competing write accesses to the same location. If there is such a write operation a_3 that takes effect in the time interval between the two read accesses, we can determine the relative order of the read accesses.

To formally state the properties of these access sequences, we will employ the function $time$. $time(a)$ is the time at which the memory access a was handled by the memory module that contains the memory location being accessed. An access sequence is therefore a sequence of events $a_1, b_1, b_2, ..., b_k, a_2$ such that $time(a_1) < time(b_1) < time(b_2) < ... < time(b_k) < time(a_2)$, where each one of the inequalities are implied either by the program order or by the execution order of accesses.

A specific case is shown in Figure 6. The accesses a_1 and b_1 are reads to two variables X and Y. a_2 and b_2 are writes to the same variables. If a_1 is made non-blocking, the operation a_2 can complete before a_1. It is thus possible that $time(a_2) < time(b_1)$ and $time(b_2) < time(a_1)$.

The observer would reason about such a behavior in the following manner. X and Y are initially set to 0. If the reads on X and Y return the value 1, it implies that the access b_2 preceded the access a_1 and the access a_2 was satisfied before the access b_1. This implies that $time(a_2) < time(b_1)$ and $time(b_2) < time(a_1)$. This together with the implied program order of b_1 and b_2, which is $time(b_1) < time(b_2)$, implies that the program order was violated for the accesses a_1 and a_2. This discussion is summarized by the following observation by Shasha and Snir[17].

Observation: Sequential consistency is violated if and only if $P \cup E$ contains a cycle.

In the example we considered, if both a_1 and b_1 were made blocking accesses, sequential consistency would not be violated. This, however, raises the question as to whether accesses could be non-blocking in the presence of conflicting accesses and still ensure the conformance to sequential consistency.

If the accesses to the variables X and Y take place in the same order in both program segments (as in Figure 7), we cannot observe any violation of sequential consistency even if all the operations are made non-blocking. In general, violation of the program order is observed only when there exist enough observations on the execution order of conflicting accesses which when combined with expected behavior (conformance to the program order) of the other program segments implies that accesses are completed out of order. The observations on the orderings of accesses on the same variable are simply the E edges (execution order edges) as defined in the previous section.

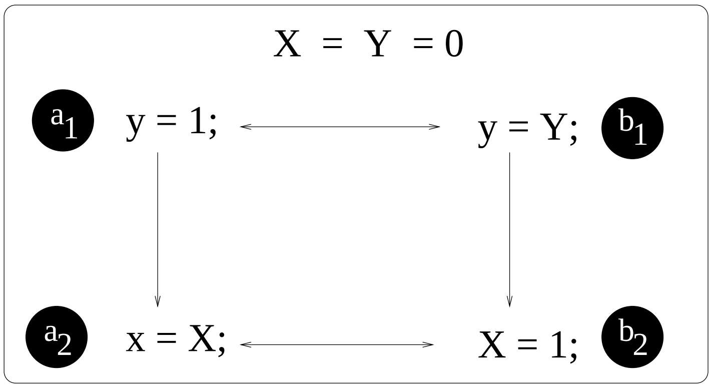


Figure 7: Graph without cycles

Therefore, the access sequence that the programmer uses in proving the violation of sequential consistency corresponds to a path comprising of E and P edges (of other program segments) that links the two distinguished accesses a_1 and a_2. We call these access sequences *back-paths* since they go back up the program order. The back-path when combined with the program order edge $[a_1, a_2]$ corresponds to a cycle in $P \cup E$.

6.3 Compiler Approach

One of our objectives is to develop compile-time techniques for relaxing blocking read-write operations into weaker non-blocking operations without violating sequential consistency. If an access is made non-blocking, the compiler has to decide when the issuing program segment is going to wait for the access to complete.

The compiler should detect at compile-time whether cycles can develop in $P \cup E$ if an access is made non-blocking. The compiler, however, does not have access to the directionality of the E edges (which is known only at runtime). In other words, the compiler can detect the C edges, but cannot determine their orientation. Also, at compile-time, we do not know the exact sequence of accesses the program would initiate. Instead, we have the *Control Flow Graph* that represents all possible execution paths. Also, we have to replace the notion of an *access* by the notion of an *access statement*. The back-paths that we are going to compute will be a string of access statements that does not make any assumptions regarding the ordering of conflicting accesses from different processors.

Therefore, for compile-time analysis, we make the following conservative approximations:

1. We are going to approximate E by C, which is just the undirected version of E.

2. At compile-time, it might not be possible to detect whether two access statements would access the same variable especially if the language allows the use of pointers. The conservative nature of dependency analysis might result in additional C edges, which is still correct since E would be a subset of C.

3. P is approximated by the control flow graph.

So, a back-path corresponds to a path of directed P edges and undirected C edges. Observe that such a path (of P and C edges) indicates only a potential ability to observe a runtime violation of sequential consistency. To illustrate this point, observe that if the access b_1 took place before the access a_2 in the program shown in Figure 6, the observer cannot prove a violation of program order even if a_2 is executed ahead of a_1. Only for a particular orientation of C edges is sequential consistency violated. But the compiler has to be conservative and ensure that a_2 does not complete before a_1 since there is a possibility that if the accesses get reordered, the program order violation might be observed. This does not mean that the earlier access has to be blocking and synchronous. The latency of the earlier request can actually be overlapped with the execution of the instructions that follow the earlier access and precede a_2. These instructions could be purely local operations or could also be remote accesses that would not reveal the non-blocking nature of the previous access. The only constraint is that a_2 should not complete ahead of the earlier access.

The aim of this analysis is to discover all such constraints. We call this set of constraints the delay set, D. The delay set is a set of pairs of access statements that appear in the control flow graph. Since an access statement could correspond to multiple run-time accesses, for every pair of access statements (a_1, a_2) that appears in D, we have to ensure that all previous a_1 accesses are complete before the completion of a_2 access that is initiated later. Efficient ways of handling these situations are discussed in a later section.

Shasha and Snir proved that there exists a minimum delay set, D. We restate their theorem in a different form. Before stating the theorem, we redefine the notion of a back-path.

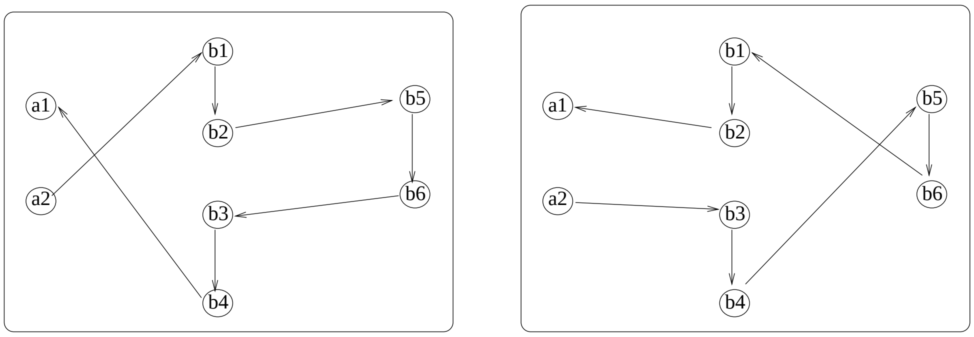
Definition: $a_1, b_1, b_2, ..., b_k, a_2$ is a back-path if:

1. $(a_1, b_1) \in C$ and $(b_k, a_2) \in C$.
2. For all $i \in [1..k-1]$, $[b_i, b_{i+1}] \in P \cup C$.
3. Let $b_m, b_{m+1}, ..., b_{m+n}$ be access statements corresponding to program segment P_l such that $(b_{m-1}, b_m) \in C$ and $(b_{m+n}, b_{m+n+1}) \in C$. Then, for all i not in [m..m+n], b_i does not belong to P_l.

Theorem: $[a_1, a_2]$ belongs to D if and only if there exists a back-path from a_2 to a_1.

The significance of this theorem is that it is sufficient to consider back-paths that visit each program segment at most once. To understand the intuition behind the constraint, let us examine the two invalid back-paths shown in Figure 8.

In case 1, there exists another back-path $(a_2, b_1, b_2, b_3, b_4, a_1)$ from a_2 to a_1 that does not visit the second program segment twice. We can remove the accesses that appear between the two visits to the second program segment. In case 2, there is no valid back-path from a_2 to a_1. There are back-paths from b_4 to b_1 and from b_6 to b_5. This means that the delay set will contain $[b_1, b_4]$ and $[b_5, b_6]$. With this delay set, E cannot contain $[b_4, b_5]$ and $[b_6, b_1]$ since that would make $E \cup D$ cyclic and violate the system contract.



Case 1 Case 2

Figure 8: Invalid Back-paths

```
                Proc: 1                          Proc: 2

Lock:     requestA = 1                     requestB = 1
          turn = 2                         turn = 1
          while   (requestB                while   (requestA
                && turn != 1);                   && turn != 2);
          numTrans++;                      numTrans++;
          fund += giftAmt;                 fund -= giftAmt;

Unlock:   requestA = 0                     requestB = 0
```

Figure 9: Peterson's Locks

The import of Shasha and Snir's theorem is that we need to consider only minimal back-paths (back-paths that do not contain other valid back-paths) in constructing the delay set. We extend this characterization to an algorithm that computes the minimal delay set in section 7.

To summarize, in this section we characterized the type of executions that can violate sequential consistency. We then developed a technique for detecting at compile-time whether such violations are possible. We then presented a technique for constructing the delay set, which encapsulates all the restrictions on the completion of remote accesses.

6.4 Example

In this section, we examine Peterson's locking algorithm (Figure 9) and determine the delay requirements imposed on accesses. There are two processes accessing the following shared variables: $requestA$, $requestB$, $turn$, $numTrans$, and $fund$. The sequence of reads and writes to $requestA$, $requestB$ and $turn$ ensures that only one of the processors is updating the variables $numTrans$ and $fund$. The program orders and the conflict relations are shown in Figure 10. The solid lines represent the program order and are unidirectional. The dashed lines represent the conflict relations and are bidirectional.

We can compute the delay set by determining for every pair of accesses whether there is a back-path. This leads to the set of constraints that is shown as solid lines in Figure 11. We observe

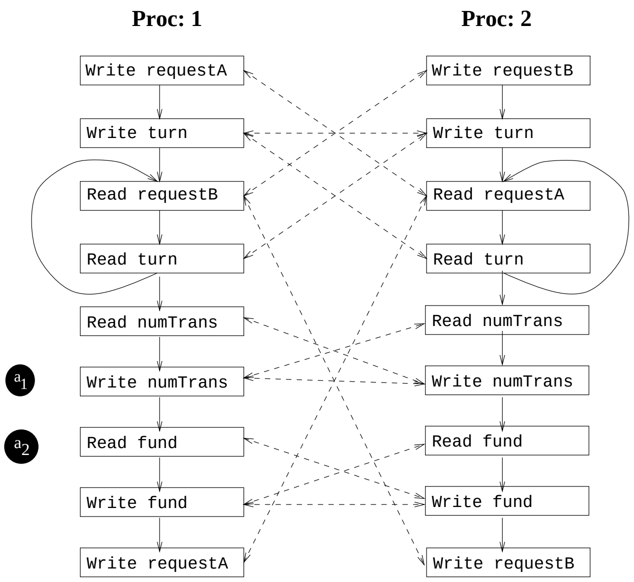


Figure 10: Conflict edges

that all the accesses to the variables $requestA$, $requestB$, and $turn$ have to be blocking accesses, which is not surprising considering that they are a compact set of accesses designed to ensure mutual exclusion. On the other hand, there are weaker constraints on the accesses to $numTrans$ and $fund$. One should note that even if the update operation does not appear in a region that is guarded by the lock, we would not be able to discover a back-path from access a_2 to access a_1. The variables $numTrans$ and $fund$ might not reflect the actual number of transactions executed due to race conditions. However, the program's execution would be sequentially consistent. The distinction is that a program's execution could be sequentially consistent even if it contains race conditions.

6.5 Evaluating Shasha and Snir's Algorithm

One of the problems with Shasha and Snir's algorithm is its dependence on the sequence of accesses, which could sometimes be misleading. If the variables $numTrans$ and $fund$ are accessed in reverse order by the two segments, there would be a back-path from a_2 to a_1 and the accesses would be construed as fragments of a synchronization operation. The algorithm would then require the accesses to be executed synchronously. We will try to alleviate this problem by exploiting global precedence/synchronization information. This is described in section 9.

Another problem with the approach is the rapid growth of the delay set as we compose more program segments (Figure 12). Suppose we had our example generalized to three different agents updating the shared variable. The resulting conflict relations would introduce a new back-path from a_2 to a_1. Here also, an analysis of synchronization operations would invalidate such a back-path and reduce the size of the delay set.

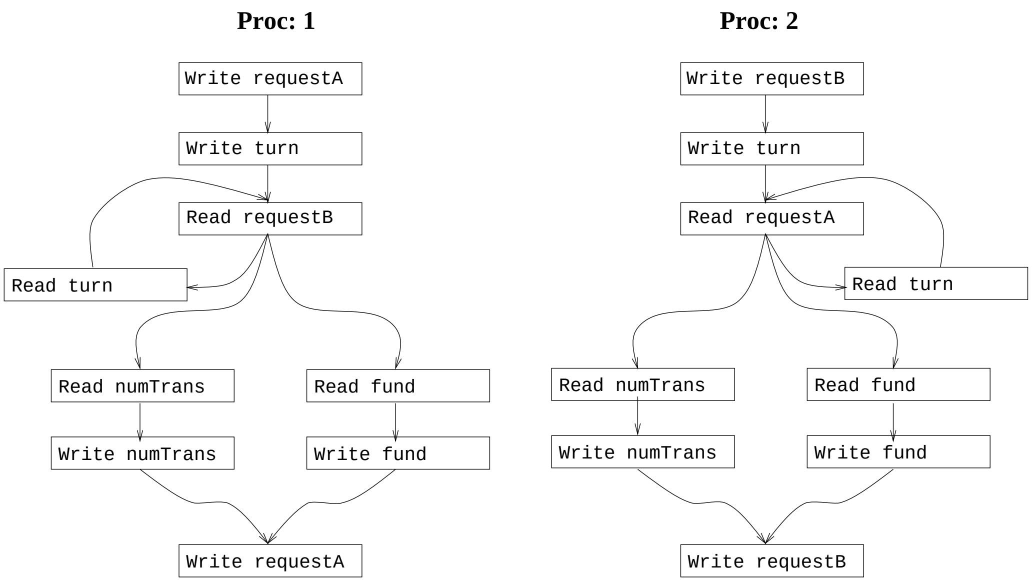


Figure 11: Delay Constraints

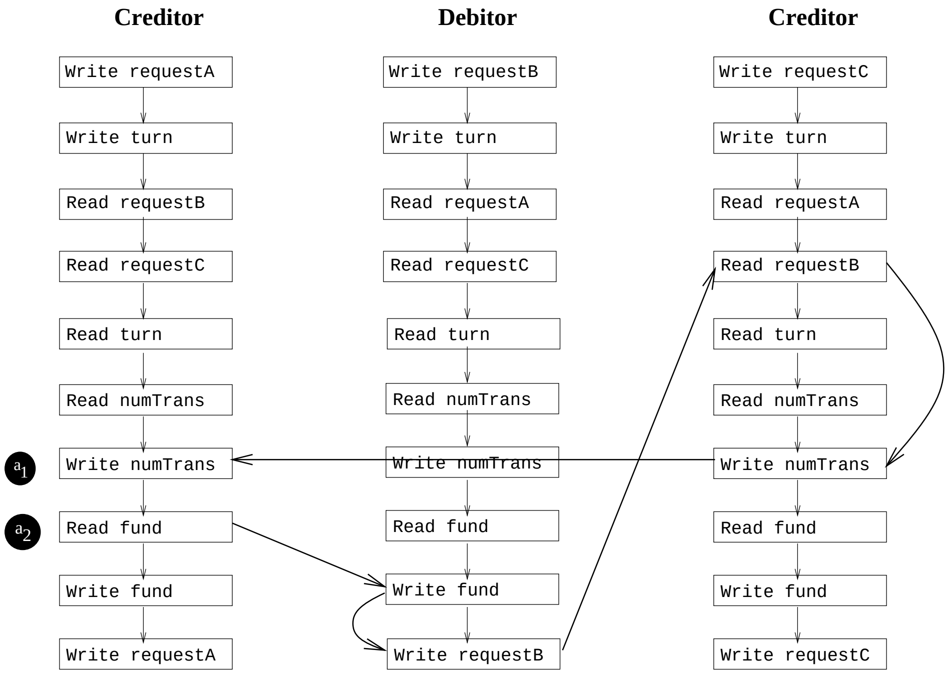


Figure 12: Composing three copies

7 Cycle Detection Algorithm

To compute the delay set, if the number of program segments is fixed, there exists an algorithm whose complexity is polynomial in the number of memory access statements in the program. However, the running time of this algorithm grows exponentially with the number of program segments. In fact, we can reduce the Hamiltonian Path problem on a graph of n nodes to computing the delay set on a program with n program segments; thus, we can show that the problem is *NP-Complete.*

In this section, we first prove that the problem of constructing back-paths is *NP-Complete.* For the sake of completeness, we also present an exponential time algorithm for computing the delay set for a general program. We then restrict the analysis to SPMD programs, and exploit the code uniformity of the program segments to devise a polynomial-time algorithm.

7.1 Path Recognition is NP Hard

We can prove that the problem of finding a back-path from an access a_1 to an access a_2 that does not visit a thread more than once is NP Hard[7]. We show that the Hamiltonian Path recognition problem[7] can be reduced to a particular instance of the back-path recognition problem (BPR). Given a general graph (V, E), we will construct a parallel program with program order P and conflict relation C, such that a Hamiltonian path exists in the original graph if and only if there exists a particular back-path involving P and C edges.

Let the vertices in V be $v_1, \ldots, v_n$. We want to check whether there exists a simple path of length $n-1$ from v_1 to v_n. We will construct a parallel program with $n-1$ threads that access a set of shared variables of the form v_j^i. For every vertex in V, we are going to construct a program thread that initiates a set of accesses in a particular order. The structure of this parallel program is defined by the accesses initiated by each thread and the order in which the accesses are initiated.

To specify the code for the threads, we will define a function $CodeSeq$. The code for thread j is based on the list of neighbors of vertex v_j in G. Let $w_1, \ldots, w_d$ be the neighbors of vertex v_j. Then the function $CodeSeq(j, i)$ expands to the following code:

CodeSeq(j,i):
  Write v_j^i*;*
  Read w_1^{i+1}*;*
  Read w_2^{i+1}*;*
  ...
  Read w_d^{i+1}*;*

We use $CodeSeq$ to define the code for threads P_1 through P_{n-1}.

P_j *is defined as:*
  CodeSeq(j, n-1);
  CodeSeq(j, n-2);
  ...
  CodeSeq(j, 1);

The structure of thread P_n is different from the other threads. It initiates only two accesses, and we will show that there is a Hamiltonian path from v_1 to v_n if and only if there exists a back-path

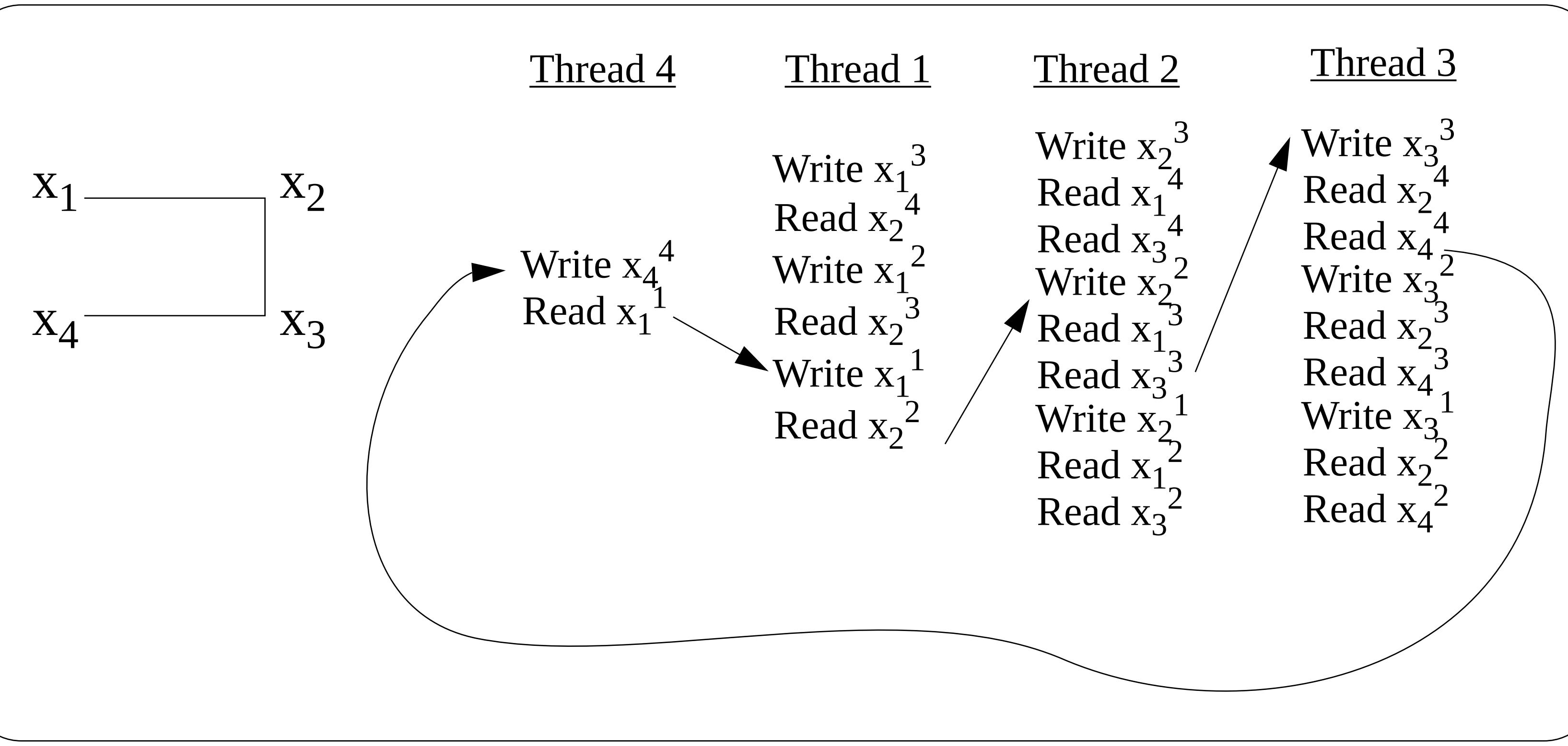


Figure 13: Constructing a parallel program for a given graph.

between the two accesses initiated by thread P_n.

P_n is defined as: Write v_n^n; Read v_1^1

Figure 13 illustrates this construction for a simple graph consisting of four vertices. There exists a Hamiltonian path from vertex x_1 to vertex x_4 in the graph and a back-path between the two accesses initiated by P_4.

Theorem: There exists a Hamiltonian path from v_1 to v_n in the graph G if and only if there exists a back-path from *Read* v_1^1 to *Write* v_n^n, which are accesses initiated by thread P_n.

Proof: We will first show that if there is a Hamiltonian path from v_1 to v_n in G, there exists a back-path between the accesses initiated by P_n.

Let $u_1, \ldots, u_n$ be the Hamiltonian path from v_1 to v_n where u_1 is v_1 and u_n is v_n. Consider the access sequence *Read* u_1^1, *Write* u_1^1, *Read* u_2^2, *Write* u_2^2, *Read* u_3^3, *Write* u_3^3, $\ldots$, *Read* u_n^n, *Write* u_n^n. By construction, (*Read* u_i^i, *Write* u_i^i) $\in C$ and (*Write* u_i^i, *Read* u_{i+1}^{i+1}) $\in P_{u_i}$. Also, this access sequence visits each thread exactly once. Therefore, this is a valid back-path from access *Read* v_1^1 to *Write* v_n^n.

We can also show that if there is a back-path between the two accesses initiated by P_n, there is a Hamiltonian path in G that originates from vertex v_1 and ends in vertex v_n. To prove this, we make some important observations about such a back-path. In particular, the back-path would have the following properties:

1. All conflict edges that appear in the back-path are of the form (*Read* x, *Write* x).
2. All program edges that appear in the back-path are of the form (*Write* v_j^k, *Read* v_l^{k+1}) where

vertex v_l is adjacent to the vertex v_j in G.

3. The back-path visits exactly $n-1$ threads (excluding the thread P_n).

These properties follow from the structure of the parallel program and the definition of back-path. The accesses initiated by a thread after a *Write* v_j^k access are either `reads` to variables of the form v_l^m with $m \leq k+1$ or `writes` to variables of the form v_l^m with $m \leq k$. The accesses initiated after a *Read* v_j^k access are either `reads` to variables of the form v_l^m with $m \leq k$ or `writes` to variables of the form v_l^m with $m < k$. Therefore, if there is a back-path from an access *Op1* v_q^r to an access *Op2* v_s^t, then the back-path must contain exactly $t-r$ P edges of the form *(Write* v_j^k*, Read* v_l^{k+1}*)*. Since $t=n$ and $r=1$, the back-path has the above mentioned properties. These properties also imply the existence of a Hamiltonian path from v_1 to v_n due to the constraint that a back-path does not visit a thread more than once.

Therefore, to solve the hamiltonian path recognition problem on a graph, we can use the reduction specified in this section and formulate an equivalent back-path recognition problem. This implies that BPR is NP Hard.

7.2 Exponential Time Back-Path Recognition Algorithm

We present a simple algorithm for detecting the back-paths in a parallel program. To prevent a back-path from revisiting a thread, we impose an ordering on the threads and orient the conflict edges from lowered numbered to higher numbered threads.

Let T_s be the thread that contains the processor edge $[u,v]$. The goal is to determine whether there exists a path from v to u that is comprised of P and C edges. Let $T_1, T_2, ..., T_k$ be the other threads that make up the program. The algorithm for detecting a back-path from v to u is as follows:

1. Pick a permutation $p_1, p_2, ..., p_k$ of the set $[1..k]$.

2. Orient the conflict edges (*i.e.,* assign directionalities) such that no conflict edge is directed from T_{p_j} to T_{p_i} where $i<j$. Also, remove all conflict edges from the thread T_s to the other threads except for those edges that involve the accesses u and v.

3. Determine whether the vertex u is reachable from vertex v in this modified graph.

4. If a back-path is discovered, add the edge $[u,v]$ to the delay set. Otherwise, repeat the process for a different permutation.

The intuition behind the algorithm is to guess an ordering of the threads that can be used in the proof of violation of sequential consistency. Given a particular ordering of threads, the process of orienting the conflict edges ensures that a path would visit a thread at most once.

This algorithm is exponential in the number of program segments. It is, however, polynomial in the number of access statements in the program. That is, given a fixed number of threads, the complexity of the algorithm grows polynomially in terms of the number of access statements.

7.3 Cycle Detection for SPMD Programs

The programming model used by Shasha and Snir allows each program segment to be different. Under the popular SPMD model, the threads share a common code image and a common entry point, but they might follow different paths in an asynchronous manner. This means that the same instruction might be executed by an arbitrary number of processors, and each processor is capable of imitating the functionality of any other processor.

In the previous section, we examined the algorithm for determining the execution constraints on a program composed of threads executing different code segments. This algorithm is polynomial when the number of threads is fixed. However, the complexity of the algorithm grows exponentially with the number of threads. For the SPMD model, a simple cycle finding algorithm on all the code segments that might exist at runtime is not feasible. First, the number of processors executing (NUMPROCS) might not be known at compile-time. Second, even if NUMPROCS is known at compile-time, replicating the control flow graph of the code image NUMPROCS times and running the cycle finding algorithm is computationally infeasible for large number of threads. In this section, we provide an efficient algorithm for finding the minimum delay set for an SPMD program. We first describe a transformation of the given control flow graph. We then present an algorithm for detecting back-paths in the resulting graph, and we close the section by proving that the back-paths in the two graphs are equivalent.

7.3.1 The Transformed Graph

In this subsection, we show how we prepare the control flow graph for our back-path detection algorithm. Let the control flow graph G be (V, P), where V and P are respectively the access statements and the control flow edges (also referred to as the processor edges) of the graph. The graph does not contain nodes for local accesses or local computation (see Figures 3 and 4). The first step is to determine the conflict edges of the graph. Let (u, v) be a conflict edge if u and v are accesses to the same object in the shared address space and at least one of the accesses is a write operation. Let C be the set of these edges.

We generate a new graph, G', with nodes V' and edges E'. V' is two copies of the accesses in G, which we label with L and R for left and right.

$$\begin{aligned}
V' &= \{< v, L >,\ < v, R > \ \mid v \in V\} \\
T_1 &= \{(< u, L >, < v, R >),\ (< v, L >, < u, R >) \mid (u, v) \in C\} \\
T_2 &= \{(< u, R >, < v, R >) \mid (u, v) \in C\} \\
T_3 &= \{(< u, R >, < v, R >) \mid [u, v] \in P\} \\
E' &= T_1 \cup T_2 \cup T_3
\end{aligned}$$

The transformation is based on the following observation. We have two copies of every access u in the SPMD program. The copy $< u, L >$ appears as endpoints of the back-paths, while the copy $< u, R >$ will always appear as an interior node in these paths. The T_1 edges connect the left and right nodes. The T_2 edges are conflict edges, but they exist only between a pair of right nodes. The T_3 edges are processor order edges and also link the right nodes. The left nodes have

```
while (turn != MYPROC);
numTrans++;
fund += giftAmt;
turn++;
```

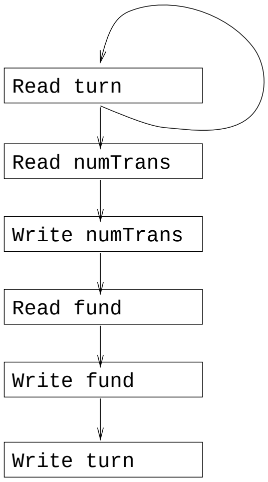


Figure 14: SPMD Program and Modified Control Flow Graph

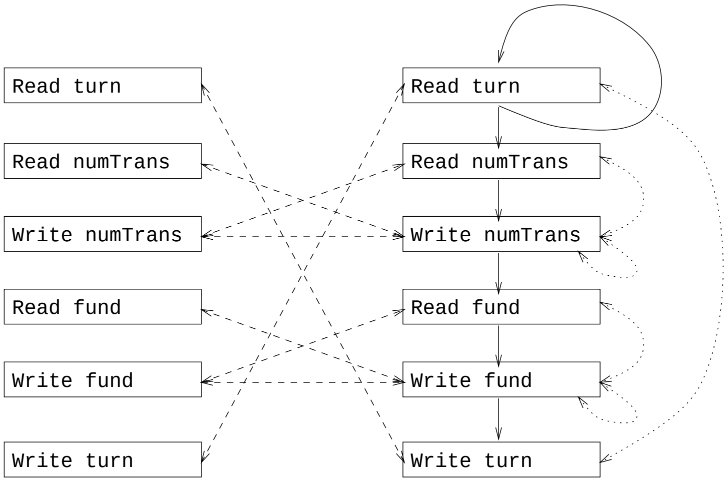


Figure 15: Transformed Graph

no internal edges. Therefore, a path from $< v, L >$ to $< u, L >$ is composed of a T_1 edge followed by a series of T_2 and T_3 edges and terminates with a T_1 edge.

For every edge $[< u, L >, < v, L >] \in P$, we check whether there exists a path from $< v, L >$ to $< u, L >$ in the graph G'. We construct the set D that consists of all edges $[u, v]$ that has a path from $< v, L >$ to $< u, L >$. If we introduce delay constraints to ensure that for every edge $[u, v] \in D$ that the access u is completed before v, there will not be any noticeable violation of sequential consistency.

To illustrate the use of the transformed graph, examine the program that appears in Figure 14. It is similar to the examples studied in the previous section, but ensures mutual exclusion by specifying a predetermined order on the updates made by the processors[2]. The transformed graph is shown in Figure 15. The T_2 edges are shown as dotted lines, the T_1 edges as dashed lines, and the T_3 edges as solid lines. Note the absence of control flow edges linking the "left nodes" and

[2]Note, however, that there is no parallelism in this example!

the presence of conflict edges that link a pair of "right nodes". This is different from the former representation in which the conflict edges always link a pair of nodes that occur in different program segments.

7.3.2 The Back-Path Algorithm

We can piece together our observations on the structure of the transformed graph to derive an algorithm for determining back-paths. The first two steps are used for transforming the control flow graph.

1. Compute the processor order, P. This is the transitive closure of the control flow graph.
2. Compute the conflict edges by identifying all accesses to a given variable. This is done conservatively using dependence analysis techniques.
3. Construct the transformed graph, and compute the reachability matrix (the set of vertices reachable) for all vertices $< u, L >$ using a standard graph algorithm.
4. If $[u, v] \in P$ and if the vertex $< u, L >$ is reachable from vertex $< v, L >$, $[u, v]$ belongs to the delay set.

All the different stages of the algorithm can be completed in polynomial time. To be more precise, if n is the number of accesses in the program, the delay set can be computed in $O(n^3)$ time.

Theorem: For SPMD programs, the Back-Path algorithm computes the same delay set as the standard Shasha and Snir's algorithm.

Proof: To prove the validity of algorithm, we show the equivalence between the paths in the program graph composed of all the threads' code images and the paths in the transformed graph. We can thus show that the delay set computed by our algorithm is correct.

We will first show that a path from $< v, L >$ to $< u, L >$ in G' corresponds to a valid back-path in G. Let S be a back-path from access v to access u in the graph G'. The path is comprised of T_1, T_2, and T_3 edges. Let S be $< u_1, R >, < u_2, R >, < u_3, R >, ..., < u_k, R >$. Consider the accesses $u_1, u_2, u_3, ..., u_k$ in the original control flow graph. This set of accesses can be used to prove violations of sequential consistency in the event of the access v completing before the access u. This can be proved by constructing an equivalent back-path that involves these accesses and chains together different threads of an SPMD program.

Since the different threads of an SPMD program execute the same code, we can utilize an arbitrary number of threads in our construction and attach the different accesses to different threads. If $< u_i, R >$ is connected to $< u_{i+1}, R >$ by a T_2 edge (which is a conflict edge), we assign the access u_{i+1} to a new thread that has not been assigned any accesses earlier in the path. If $< u_i, R >$ is connected to $< u_{i+1}, R >$ by a T_3 edge (which is a processor order edge), we assign the access to the same thread as the access u_i. We repeat the process and might use up to k threads. The result is a sequence of accesses on different threads that constitute a back-path from v to u assuming that the number of threads we used in constructing the cycle is less than the total number of threads that would exist at runtime.

On the other hand, we also have to prove that any back-path that might exist in an SPMD program would be spotted by our algorithm. This is easy to prove since every path comprising of

P and E edges has a corresponding path of T_2 and T_3 edges in the transformed graph. This might not be a simple path in that it might pass through the same access statement more than once in G'. However, if u is reachable from v in G, then $< u, R >$ is reachable from $< v, R >$ in G'. This implies that we will compute the same delay set.

If the total number of running threads is less than the number of threads used in the back-path, we have made a conservative error in finding the delay set. The delay set computed by this algorithm is conservative in assuming unlimited number of threads per program. The assumption is that the number of processors that the program would utilize at runtime is not known at compile-time. This process provides an executable that can be used on a machine of an arbitrary size. If the number of threads is limited to k, any back-path that involves more than k threads would not be a valid proof for sequential consistency violations.

If the machine size is known at compile-time, we can modify our algorithm to restrict the number of threads that a back-path visits to be less than the number of processors in the machine. This is easily accomplished by attaching an unit weight on the T_1 and T_2 edges and a zero weight on the T_3 edges. As a result of this transformation, a path of weight k in the graph visits exactly k program segments. The decision problem of deciding whether a delay edge between u and v is now resolved by determining the existence of a path of weight that is less than the number of processors.

8 Code Generation

In this section, we describe how the delay set information is used to maximize the pipelining benefits of the latency hiding techniques.

The input to this phase is the control flow graph, the delay graph computed by the back-path recognition algorithm, and the *use-def* graph for local variables. The control flow graph and the delay graph are available from the previous compiler pass. The *use-def* graph needs to be computed. The use-def graph represents the flow of data from definition to usage. Every definition of a variable is associated with a list of program statements where the value might be used. This graph can be obtained through standard compiler analysis techniques.

In this code generation process, we would like to satisfy the following constraints:

1. Delay constraints are observed.

2. Before every use of a local variable, the corresponding definition is complete.

Consider the program shown in Figure 16. The solid line is the delay edge, and the dashed line is def-use edge for the local variable x.

In this section, we will discuss how the constraints on completion of accesses is expressed in Split-C. We will describe different code generation techniques with varying complexity. We will also discuss the trade-offs that show up during code generation.

8.1 A Simple Code Generation Module

The simplest code generation algorithm works as follows.

The compiler generates a temporary counter variable for every remote access statement. One should bear in mind that an access statement might initiate multiple accesses during a program's

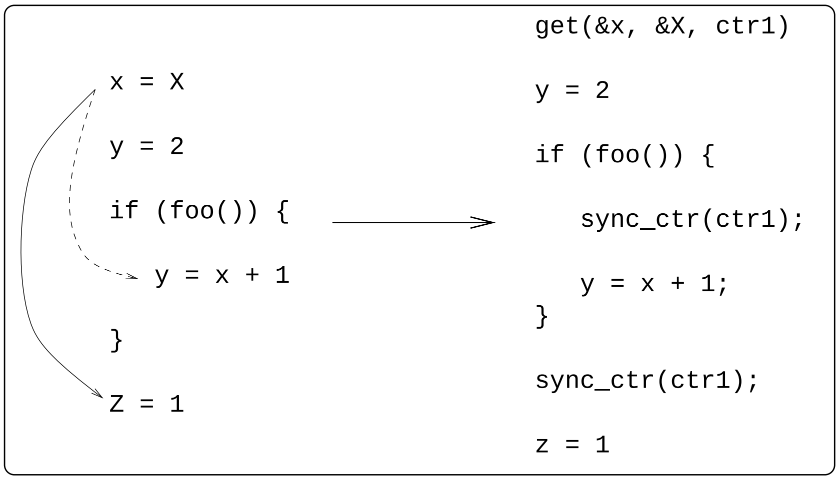


Figure 16: Code Generation

execution. The counter can be used to ensure completion of all the accesses that have been initiated by the access statement. A Split-C code generated for a sample program is shown in Figure 16. The counter variable is generated by the compiler. A split-phase get operation is initiated to fetch the value of X into the local variable x. A later `sync_ctr` operation on $ctr1$ ensures completion of all accesses initiated by the access statement.

The `sync_ctr` operation waits until the accesses are complete by waiting for the counter value to be reset. A property that makes code generation easy is that a `sync_ctr` operation behaves like a *null* operation if the program has already executed a `sync_ctr` on the same counter. In other words, a particular control path through the program can encounter multiple `sync_ctr` operations on the same counter. This suggests the following simple scheme for code generation.

Let a be an access statement in the program. Let $[a, b_1]$, $[a, b_2]$,..., $[a, b_k]$ be the set of delay constraints on this statement, and if a is a remote read operation, let $[a, c_1]$, $[a, c_2]$,..., $[a, c_l]$ be the set of def-use edges for the local variable being defined by the statement. The compiler converts a into a split-phase operation, and inserts a `sync_ctr` operation just in front of the access statements b_1, b_2,..., b_k, c_1, c_2,..., c_l. If, however, a `write` access does not have any delay constraints, we transform the `write` access into a `store` access, which is more efficient since it avoids acknowledging the completion of the access.

8.2 Pragmatics of Code Generation

The primary drawback of the simple code generation algorithm is the excessive use of the `sync_ctr` operation. Certain obvious improvements can be made to the simple scheme. However, it is not clear whether an optimal compile-time technique exists for code generation. As we will discover in this section, the code generation problem is similar in spirit to other compile-time techniques that need profiling information to generate near-optimal code.

Even though correctness of the program's execution is not violated by introducing extra `sync_ctr` operations, we would like to minimize their use since there is a cost attached to executing a `sync_ctr` operation. The first step is to reduce the number of program points at which `sync_ctr` operations are introduced.

Here is the modified algorithm for introducing `sync_ctr` operations:

1. Every remote access operation a is split into two operations: the corresponding split-phase initiate and a `sync_ctr` operation.
2. Let s be the `sync_ctr` operation associated with the split-phase initiate statement i. Rules are used for propagating s through the control flow graph in order to increase the number of instructions between i and s.
   (a) If s is ahead of b in a basic block in the control flow graph and if there are no delay or def-use constraints of the form $[i, b]$, then move s past b. If there are delay or def-use constraints of the form $[i, b]$, s is placed in front of b.
   (b) If s is at the end of a basic block, propagate s to all the successors of the basic block, and continue the motion of the different copies of s.
   (c) If s is ahead of another copy of s, merge the two s operations into a single s operation.

This algorithm propagates the `sync_ctr` operations as far away from the initiation as possible. Also, if the access a is constrained to complete before the set of access statements b_1, b_2,..., b_k and if for some statement b_l there is no possible flow of control that hits b_l without encountering one of the other b_i statements, then the algorithm does not introduce a `sync_ctr` operation ahead of b_l. The simple algorithm would have incurred the penalty of an extra `sync_ctr` operation. A dual set of rules exist for propogating the initiate operation to program points that would be executed earlier during program execution. The goal is to maximize the distance between the initiate and the sync operation.

However, the algorithm still suffers from two drawbacks. First, there could be still certain control paths that execute more than one `sync_ctr` operation (as in Figure 16). Second, if there is a delay constraint in which the initiation and the sync_ctr are nested within different conditionals and loops, our algorithm could execute unnecessary `sync_ctr` operations.

For example, given a delay constraint $[a, b]$ where b appears inside a loop, but a does not, we would not want to introduce a `sync_ctr` operation inside the loop since that would require the operation to be executed as many times as the loop would be executed. All but the first `sync_ctr` operation would be redundant. To avoid the cost of unnecessary sync operations, we could employ a loop-unrolling technique. We could separate the first iteration of the loop from the other iteration and introduce the `sync_ctr` operation only in the code for the first iteration.

The opposite problem occurs for a delay $[a, b]$ where a appears inside a conditional, but b does not. It is not clear where the `sync_ctr` operation should be introduced to ensure optimal performance. If we have the `sync_ctr` operation just ahead of b, we could suffer the penalty of executing the operation even when a access had not been executed. Note that this does not affect the correctness of the code due to the nature of Split-C counters. On the other hand, if we introduce the `sync_ctr` operation at the end of the conditional containing a, we might be hiding only part of the latency by prematurely waiting for its completion. Static analysis cannot help in choosing between the two alternatives. Relative costs of remote accesses and local memory operations (for updating counters) could be used as an heuristic for code generation.

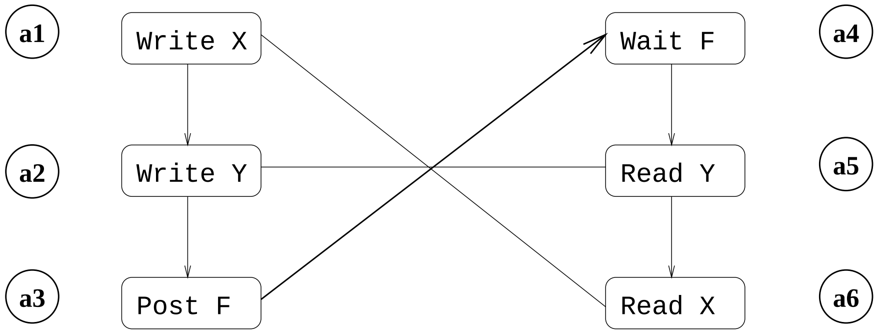


Figure 17: Synchronization operations

9 Using Synchronization Information

The delay set computed by algorithms presented in the earlier sections is obtained without analyzing synchronization. Synchronization constructs create mutual exclusion or precedence constraints on accesses executed by different processors. We can use this information to prune the number of back-paths discovered in our analysis, which would result in a decrease in the number of constraints imposed on the program. We consider an example in which precedence information increases the applicability of the pipelining optimization.

In Figure 17, we have two program segments that access the variables X and Y. A simple application of our back-path algorithm would prevent pipelining of these accesses. Since the variables are accessed in different orders within the two program segments, the analysis algorithm would impose delays between the accesses. However, if the synchronization behavior is analyzed and used, we will conclude that some of the delays are unnecessary. Processor 1 initiates a `post` operation after accessing these variables, and processor 2 does a `wait` before accessing these variables. Assume that the accesses in the first program segment have to be completed before the `post` operation. Since the other program segment will read the values only after the `post` operation is complete, we can pipeline these accesses without violating sequential consistency. This example illustrates one way in which our algorithm is overly conservative. We can trace this problem to the characterization of the conflict edges. The conflict edges are bidirectional since, in general, we are unable to predict the runtime execution order. However, the conflict edges may sometimes be ordered if we incorporate synchronization analysis into the algorithm. In this section, we will examine three synchronization analyses: post-wait synchronization, barriers, and locks.

9.1 Analyzing Post-Wait Synchronization

Post-wait synchronization is commonly used for producer-consumer dependencies in parallel programs. We exploit the strict precedence established between the operations executed by the producer before the `post` operation and the operations executed by the consumer after the `wait` operation[3]. We represent this precedence relationship by a directed precedence edge from the `post` operation to the `wait` operations. In our discussion, we use R to denote the set of precedence edges.

[3]In our analysis, we assume that it is illegal to `post` more than once on an event variable.

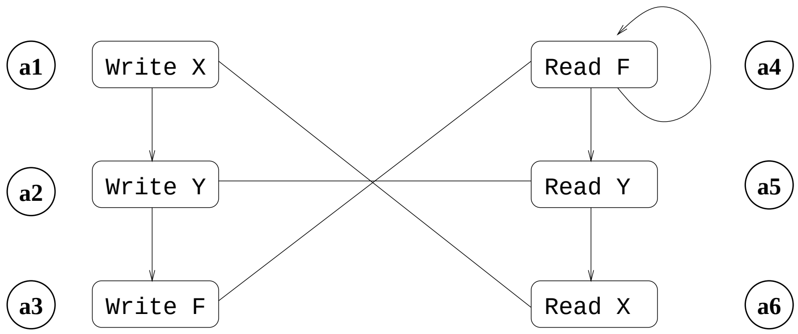


Figure 18: Synchronization operations as reads and writes

Consider the example in Figure 17 in more detail. When we apply the back-path algorithm, we obtain a delay set that requires the completion of a_1 before the initiation of a_2 (and the completion of a_5 before a_6). The delay set is: $[a_1, a_2], [a_2, a_3], [a_1, a_3], [a_4, a_5], [a_5, a_6], [a_4, a_6]$. The semantics of post-wait can be used to identify the precedence edge from a_3 to a_4, which leads to weaker delay constraints by directing the conflict edge between a_3 and a_4. If we require the delay set to contain $[a_2, a_3], [a_1, a_3], [a_5, a_6], [a_4, a_6]$, we can direct the other conflict edges $[a_1, a_6]$ and $[a_2, a_5]$, and thus destroy the remaining back-paths.

This example illustrates a process of gradual refinement of the precedence and the delay set information. We start with a minimum amount of precedence information based on synchronization operations. As we build the delay set, we obtain more precedence information that can then be used to provide directionality to the conflict edges. In our example, we discovered that a_1 precedes a_6 after including $[a_1, a_3]$ and $[a_4, a_6]$ in the delay set. This suggests that certain delay relations are more fundamental than others and that a systematic process is required for building the delay set. In our algorithms, we initially discover the delay restrictions between normal accesses and synchronization operations before computing delay restrictions between a pair of normal accesses. This, however, was not the case when we had no precedence information. The conflict edges were bidirectional to start with, and they remained bidirectional even as new delay relations were discovered since none of the conflict edges were ordered.

Our synchronization analysis works only if the programmer uses the synchronization primitives provided by the language. If the programmer builds synchronization operations using primitive reads and writes on shared memory locations, we would not be able to detect the synchronization. In Figure 18, we have the same program but with the `post` and `wait` operations replaced by primitive reads and writes on the flag variables. In this case our algorithm is still correct, but we would then not be able to prune the delay set.

The process of invalidating a back-path need not always involve providing an execution order to a conflict edge, as shown by the example in Figure 19. Since there are back-paths from a_3 to a_1 and from a_6 to a_4 and a_3 and a_4 are synchronization accesses, $[a_1, a_3]$ and $[a_4, a_6]$ belong to the delay set. This information, when combined with the precedence edge $[a_3, a_4]$, implies that a_1 precedes a_6 for any execution of the program. Since a back-path for a_1 corresponds to a possible runtime access sequence where all the accesses in the sequence execute before a_1, a_6 will never occur in a back-path for a_1. Therefore, we can remove a_6 while determining the existence of back-paths to a_1. Removal of a_6 destroys the back-path from a_2 to a_1, which otherwise would have resulted in

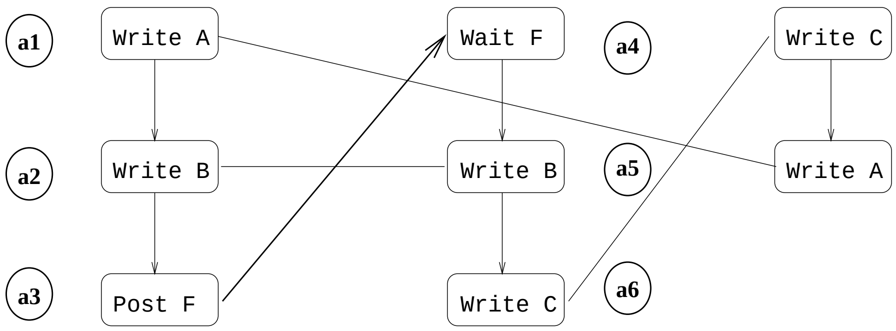


Figure 19: Synchronization analysis

$[a_1, a_2]$ being added to the delay set.

Based on the examples we have studied, we can propose a general scheme for finding the delay set. We initially find the delay restrictions between ordinary accesses and synchronization operations using the general back-path recognition technique. We then compute the complete precedence information by combining the delay restrictions and the precedence relation between the synchronization operations. To use information from post-wait operations, we need to determine the dominator tree of the control flow graph. A node u is said to dominate a node v if u appears on every path from the "entry" node of the graph to v. A node w is said to be the immediate dominator of the node v if there is no other dominator of v which is dominated by w. The dominator tree is one in which each node is connected to its immediate dominator. By building the dominator tree, we can determine efficiently whether a `post` operation would be executed after an access u, or whether u is always executed after a `wait` operation.

We now present the modified algorithm for computing the delay set.

1. Obtain the dominator tree for G based on the program order.

2. Compute the delay restrictions between normal accesses and synchronization accesses using the standard back-path algorithm. Let this delay set be D_1.

3. Compute the set of precedence edges, R, between the post and wait constructs.

4. For every pair of access statements a_1 and a_2, check whether there exists two other statements b_1 and b_2 that satisfy the following constraints.

   (a) a_1 dominates b_1 and b_2 dominates a_2,

   (b) $[a_1, b_1] \in D_1$ and $[b_2, a_2] \in D_1$, and

   (c) $[b_1, b_2] \in R$

   Add $[a_1, a_2]$ to R if b_1 and b_2 exist.

5. For every conflict edge (a_1, a_2), if $[a_1, a_2] \in R$, direct the conflict edge from a_1 to a_2.

6. To compute the delay constraint for a pair of access statements a_1 and a_2, apply the back-path algorithm on a graph where all access statements b that satisfy the condition $[a_1, b] \in R$ have been removed.

```
if  (...) {
   barrier();    ①
}
barrier();       ②
...
barrier();       ③
```

Figure 20: Inaccuracies in analysis of barrier statements

9.2 Analyzing Barrier Synchronization

The `barrier` statements can be used to separate the program into different phases that do not execute concurrently. To use a `barrier` for ordering the phases, we have to arrive at the `barrier` at the same time. If the `barrier`s are nested within conditionals and if there are unequal number of `barrier`s for different paths through the conditional, then different processors might be executing `barrier` operations that correspond to different program points. For example, in Figure 20, some of the processors might be executing the barrier statement that is numbered 3 while others are executing the number 2 barrier statement. This makes it difficult to partition the program into different phases. However, we can recognize the common situation in which the number of barrier statements executed is the same for every path through the control flow graph. We can then set up the precedence relations, and use the algorithm discussed earlier in the section.

9.3 Lock Based Synchronization

We can extend our synchronization analysis to locks, even though there are no strict precedence relations implied by the use of locks. The first task is to determine the delay constraints between normal accesses and synchronization accesses, which we denote by the set D_1. We then determine the set of accesses guarded by a lock. An access a is said to be *guarded* by the lock l, if the following conditions hold:

1. a is dominated by a *lock* l operation (which we will call b_1), and there are no intervening *unlock* l operations.

2. a dominates *unlock* l operation, which we will call b_2.

3. $[b_1, a] \in D_1$ and $[a, b_2] \in D_1$

If access statements a_1 a_2 are guarded by the lock l, we remove all other access statements that are guarded by the same lock before checking for a back-path from a_2 to a_1. This is a valid operation by the following reasoning. If $a_2, b_1, b_2, ..., b_k, a_1$ is a back-path, then the accesses corresponding to $b_1, b_2, ..., b_k$ must occur after a_2 and before a_1. It follows from our definition of being guarded by a lock that none of $b_1, b_2, ..., b_k$ can be guarded by the same lock and still appear in a violation sequence. This improvement to the delay set construction allows accesses within critical regions to be overlapped.

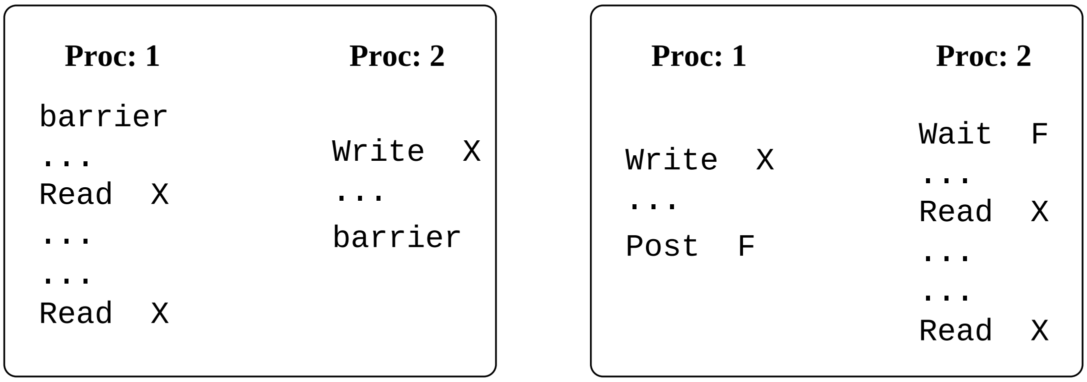


Figure 21: Phases

10 Eliminating Remote Accesses

In the previous sections, we studied methods for computing the delay set, which can then be used in determining when accesses can be pipelined. The criterion is whether the observer can detect violations of sequential consistency when certain accesses are made non-blocking. In this section, we extend the approach to eliminate remote accesses. Again, a remote access can be eliminated if we can ensure that sequential consistency would not be violated.

10.1 Eliminating Redundant Accesses within a Basic Block

We start by examining the problem when multiple accesses to a particular variable appear within a single basic block. Let a_1 and a_2 be read accesses to the same variable in the global memory. If the variable is a read-only variable (there are no writes to the variable), then we can eliminate the second read access and reuse the value obtained by the first read. This optimization is also applicable if the variable is a read-only variable in this *phase* of the program (that is, there are no write accesses that can take place at the same "time" as these two read accesses). That is, if all write accesses (to the same variable) occur either before the first read access or after the second read access, the two reads are going to get the same value. In order to incorporate this optimization, the compiler generates code to store the value obtained by the first read operation in a compiler generated temporary local variable.

Two examples are shown in Figure 21. In the first case, there is a barrier synchronization that marks the transition to a read-only variable. In the second case, the post-wait synchronization ensures that the read accesses take place after the write. If compile-time analysis can prove that the variable is not modified in some phase of the program, we can eliminate all but one read access. This could be due to a global phase (indicated by a barrier synchronization) or could be due to exclusive access to a variable (indicated by locking constructs). The synchronization analysis techniques presented in the previous section can derive this information. After synchronization analysis, if there are no delay constraints between two read accesses to a variable, it indicates the absence of conflicting writes and the value can be cached without impacting correctness.

It is not always necessary to have mutually exclusive access to the variable to exercise these optimizations. Just as we were able to pipeline global accesses as long as the optimizations do not

```
Proc: 1              Proc: 2

Read   X
...                  ...
(no global           Write   X
 accesses)           ...
...
Read   X
```

Figure 22: Intervening Local Operations

```
Proc: 1          Proc: 2          Proc: 3

Read   X         Write   Z        Write   X
...              ...              ...
Read   Z         Write   X        Write   Z
...              ...              ...
Read   X
```

Figure 23: Intervening Global Operations

expose violations of sequential consistency, we can eliminate repeated accesses to the same variable in the absence of certain patterns that might expose these optimizations. We study some specific cases before proposing the general technique.

In Figure 22, we have two read accesses to a variable that are separated by a local computation. Would the program produce inconsistent results if the second read access was eliminated and the value read initially was reused? It turns out that the program would not produce inconsistent results regardless of the access pattern of the other threads. There is no way to distinguish the reuse of a cache value from the situation in which the processor issues the first read access, completes the intervening instructions, and issues the later read before any other thread could modify the value. The delay set includes the processor edge between the two reads; however, this is to ensure that the second read obtains a value that is not "earlier" than the initially read value. A reuse of the cached value would trivially satisfy this requirement. Another view of this optimization is as a combination of prefetching and piggy-backing of accesses. The absence of intervening global accesses allows us to advance (or prefetch) the second read access to a point that is immediately after the first read access. We can then combine the two read accesses without violating sequential consistency.

In fact, reuse of a cached value can be effected even when there are intervening global accesses, as long as the previous analysis shows that the second read operation could be prefetched at the point of the first read. There should not be a sequence of delay edges from the first read to the

```
x1 = X
y = x1 + 1
z = Z
x2 = X
print x2 + y + z
```

```
x2 = x1 = X
y = x1 + 1
z = Z
print x2 + y + z
```

Figure 24: Caching by prefetching

```
x1 = X
y = x1 + 1
z = Z
x2 = X
print x2 + y + z
```

```
z = Z
x2 = x1 = X
y = x1 + 1
print x2 + y + z
```

Figure 25: Caching by postponing

second read. An example is shown in Figure 24. There are global accesses between the two reads of X. However, there are no delay restrictions on the completion of the second read access with respect to the intervening global accesses. This implies that if the second access can complete before the read of Z, such a behavior would not be observed. We can therefore "move" this access ahead of the other global accesses. We now have two adjacent read accesses to X and the value therefore needs to be read only once. This code movement is not required in practise and is useful only to illustrate the correctness of the optimization.

In Figure 25, there are no constraints on the first read access of X, and it can therefore be postponed to a later point in the program so that it appears next to the other global read. However, this is a much tougher optimization since it requires the postponement of all the computation that requires the value obtained by the first read operation.

In the examples presented so far, caching was used to eliminate redundant reads. Caching belongs to a class of optimizations that are illustrated in Figure 26. In each case, global accesses are eliminated. A read of a variable that has been recently updated by the same thread can be eliminated if the written value is still available. This is similar to constant propagation and is also a component of common subexpression elimination (since we can discover similarities between more complicated expressions that involve read accesses to this variable). Also, when a thread issues two successive writes to the same variable, we can buffer the earlier writes in a local variable and

```
x1 = X                 tmp = x1 = X
...          →         ...
x2 = X                 x2 = tmp
```

Caching

```
X = x1                 X = tmp = x1
...          →         ...
x2 = X                 x2 = tmp
```

Constant Propagation or Common Subexpression Elimination

```
X = x1
...          →         ...
X = x2                 X = x2
```

Write-backs instead of write-through

Figure 26: Transformations

write back only the last write access to the variable. This is equivalent to having a write-back cache instead of a write-through cache. The use of temporary variables (which are introduced by the compiler) is shown in the figure.

As mentioned earlier, these optimizations can be made only when code motion allows the second access to be made immediately after the first access, which in turn is possible only when there are no intervening global accesses that have to be executed synchronously with respect to these two accesses. In other words, there should not exist a sequence of two or more delay edges that forms a path from the first access to the latter one. A path comprising of a single delay edge does not restrict the incorporation of the prefetch optimization.

The algorithm for eliminating redundant accesses is outlined below:

1. Obtain the delay constraints on accesses using the algorithms presented in previous sections.

2. For every pair of accesses u and v that access the same variable and belong to the same basic block, determine whether there is a sequence of two or more delay edges that forms a path between u and v or whether there exists a global access w that appears after u and is constrained to execute before v. If there is no such pattern, eliminate a global access using one of the transformations shown in Figure 26.

```
data = NOT_READY;
while (data != READY)
    numTries++;
```

Figure 27: Infinite use of cached value

10.2 Caching across Basic Blocks

The caching optimization does not, in general, work across basic blocks. The primary difficulty is ensuring that the threads make progress and see the new values of variables some finite time after they have been updated. In Figure 27, for example, caching of *data* might result in an infinite loop. To avoid this problem, we can estimate conservatively at compile-time whether a cached value might be used infinitely often due to stale data. This decision process involves following def-use chains and computing the set of variables whose value might depend on the value obtained by the remote read. If there is a loop termination condition that depends on the value of any one of these variables, then the compiler should be conservative and not insert a caching optimization. The analysis required is complex, and we therefore do not implement caching across basic blocks in our prototype compiler.

11 Potential Benefits

We quantify the potential benefits of our approach by hand-coding the optimizations in a small set of applications. The execution times of these applications were improved by 20-50% through message-pipelining and one-way communication optimizations. These were measured on the CM5 multiprocessor. The relative speedups should be even higher on machines with lower communication startup costs or longer latencies (when the fraction of the latency that can be overlapped is higher). We present the results from four applications:

1. **FFT:** Computing the fast-fourier transform.
2. **Stencil:** 4-point stencil computation on a regular grid.
3. **CG:** Computing the conjugant gradient of a sparse matrix.
4. **EM3D:** Solving Maxwell's equations on an irregular grid.

Figure 28 gives the performance results of these experiments. The figure gives the normalized execution times with the unoptimized execution time set to 1. Thus, a relative speed of 0.5 corresponds to a factor of 2 speedup. Other optimizations, such as caching remote values, are also enabled by our analysis, and result in additional performance improvements on some of these applications.

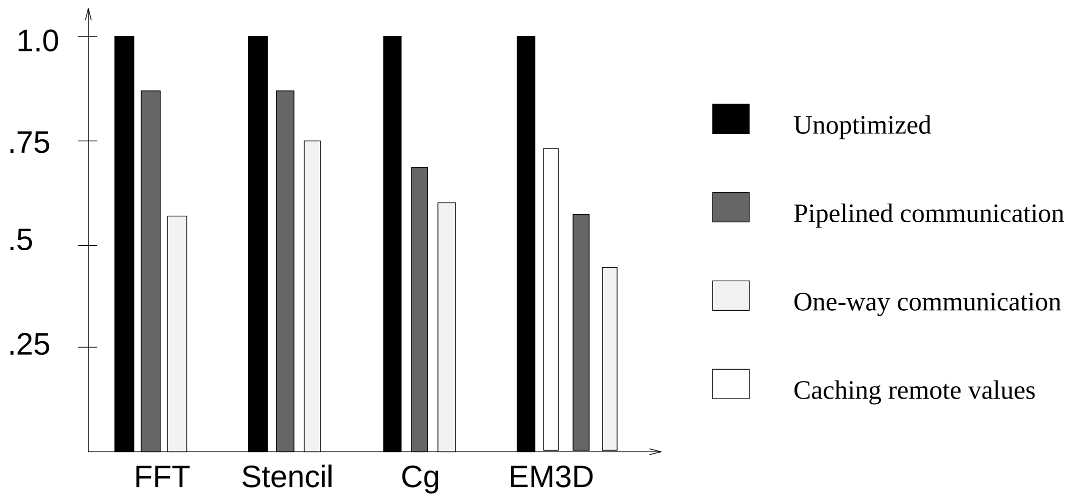


Figure 28: Normalized Execution Times

12 Related Work

As mentioned earlier, our work is based on the pioneering work by Shasha and Snir[17]. Since then, not many researchers have looked at the problem of optimizing explicitly parallel programs that exhibit control parallelism. Midkiff and Padua[14] describe eleven different instances where standard optimizations (like code motion and dead code elimination) cannot be directly applied to parallel programs. Midkiff et al[15] extend Shasha and Snir's algorithm to handle array based accesses by determining all the minimal cycles in the program and eliminating some of these cycles using array subscript analysis techniques. However, their analysis technique is computationally expensive even for programs with a small degree of parallelism since both the minimal cycle detection problem and the array subscript analysis problem have exponential running times. The algorithm presented in this paper for SPMD programs does not deal with array analysis. We believe their techniques for handling array subscripts could be incorporated into our SPMD framework. We currently handle array accesses using less sophisticated array analysis, but strong synchronization analysis that detects the different phases of the program and thus eliminates many of the conflict edges. In fact, all the applications mentioned in section 11 are "bulk-synchronous", and have `barrier` synchronizations at the end of every meaningful phase of the program. Therefore, the number of conflict edges discovered by our algorithm is minimal, and imposes minimal delay constraints on the program's execution.

Related work has been done in using synchronization constructs to extract more information about a program's behavior. Callahan and Subhlok[4] use synchronization information to frame data-flow equations for a parallel program. Grunwald and Srinivasan[9] have extended this analysis to study *post-wait* constructs appearing inside loops. They also do this in the context of data-flow analysis of parallel programs to discover optimizations like constant propagation across threads. Strict precedence information is required for these optimizations. Our analysis requires only mutual-exclusion information for decreasing the number of conflict edges, and hence we expect synchrnonization analysis to be extremely useful for discovering pipelining optimizations. It is also important to note that the optimizations discussed in section 10 deal with constant propagation and common-subexpression elimination within a single thread. These are important optimizations for generating good scalar code. To the best of our knowledge, the analysis discussed in this report is the first attempt at incorporating the standard scalar code optimizations for parallel programs.

Compilers and runtime systems for data parallel languages like HPF and Fortran-D[10] implement message pipelining optimizations. The Parti runtime system and associated HPF compiler use a combination of compiler and runtime analysis to generate code for overlapping communication, aggregating groups of messages, and other optimizations [3]. Compiling data parallel programs is fundamentally different than compiling SPMD programs. First, it is the compiler's responsibility in a data parallel setting to map parallelism of degree n (the size of a data structure) to a machine with $PROCS$ processors, which can sometimes lead to significant runtime overhead. Second, the analysis problem for data parallel languages is simpler, because they have a sequential semantics. Standard data-dependence techniques can be used in data parallel language to determine whether code-motion or pipelining optimizations are valid.

13 Conclusions

We have presented analysis techniques and optimizations for SPMD programs on distributed memory multiprocessors. The main optimization is masking latency of remote accesses by message pipelining and prefetching. Other optimizations such as common subexpression elimination and constant propagation are also enabled by code motion. The potential payoff of a few of these optimizations is quantified using hand optimizations on a small set of applications. The performance improvements are as high as a factor of two on the CM5, with even better performance expected on future architectures with lower communication startup.

The new form of analysis that is needed for explicitly parallel programs in a general (not data-parallel) execution model, is cycle detection, as introduced by Shasha and Snir. The analysis computes the constraints required on access completion to conform to the sequentially consistent model of execution for parallel programs. We show that their formulation of the analysis led to an NP-complete problem and, therefore, an algorithm that was exponential in the number of processors. Applied to an SPMD program, their algorithm relied on analyzing $PROCS$ copies of the code. We improve on their basic algorithm by reformulating the problem and gave a polynomial time algorithm that uses only two copies of the code and computes nearly the same set of cycles. We improve on the accuracy of the analysis by using synchronization information, and discover more opportunities for incorporating latency masking optimizations. This analysis is important since most parallel programs reduce the number of conflicting accesses through synchronizing operations. We also extend the analysis to incorporate standard scalar optimizations like common-subexpression elimination in parallel code. We also showed how to use this analysis to generate code for an abstract machine language, Split-C.

References

[1] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. An overview of the PTRAN analysis system for multiprocessing. In *Proceedings of the 1987 International Conference on Supercomputing*, 1987.

[2] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: Data structures for parallel computing. Computation Structures Group Memo 269, Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge, Massachusetts, February 1987.

[3] H. Berryman, J. Saltz, and J. Scroggs. Execution time support for adaptive scientific algorithms on distributed memory multiprocessors. *Concurrenty: Practice and Experience*, pages 159–178, June 1991.

[4] D. Callahan and J. Subhlok. Static analysis of low-level synchronization. In *ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 100–111, 1988.

[5] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in Split-C. In *Supercomputing '93*, pages 262–273, Portland, Oregon, November 1993.

[6] J. Demmel. LAPACK: A portable linear algebra library for supercomputers. In *Proceedings of the 1989 IEEE Control Systems Society Workshop on Computer-Aided Control System Design*, December 1989.

[7] M. Garey and D. Johnson. *Computers and intractability : a guide to the theory of NP-completeness*. W. H. Freeman Company, 1979.

[8] K. Gharachorloo, D. Lenoski, J. Laudon, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *17th International Symposium on Computer Architecture*, pages 15–26, 1990.

[9] D. Grunwald and H. Srinivasan. Data flow equations for explicitly parallel programs. In *ACM Symposium on Principles and Practices of Parallel Programming*, 1993.

[10] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiler optimziations for Fortran D on MIMD distributed-memory machines. In *Proceedings of the 1991 International Conference on Supercomputing*, 1991.

[11] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[12] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *17th International Symposium on Computer Architecture*, pages 148–159, 1990.

[13] S. Luna. Implementing an efficient global memory portability layer on distributed memory multiprocessors. Master's thesis, University of California, Berkeley, May 1994.

[14] S. Midkiff and D. Padua. Issues in the optimization of parallel programs. In *International Conference on Parallel Processing - Vol II*, pages 105–113, 1990.

[15] S. P. Midkiff, D. Padua, and R. G. Cytron. Compiling programs with user parallelism. In *Languages and Compilers for Parallel Computing*, pages 402–422, 1990.

[16] W. Oed. The Cray research massively processor system: T3D. Ftp from ftp.cray.com, Nov. 1993.

[17] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, April 1988.

[18] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *International Symposium on Computer Architecture*, 1992.

[19] K. Yelick, S. Chakrabarti, E. Deprit, J. Jones, A. Krishnamurthy, and C.-P. Wen. Data structures for irregular applications. In *DIMACS Workshop on Parallel Algorithms for Unstructured and Dynamic Problems*, Piscataway, New Jersey, June 1993.