Compiler Analyses and System Support for Optimizing Shared Address Space Programs

by

Arvind Krishnamurthy

Bachelor of Technology, Indian Institute of Technology, Madras, 1991
Master of Science, University of California, Berkeley, 1994

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor Katherine Yelick, Chair
Professor David Culler
Professor Gregory Fenves

1999

The dissertation of Arvind Krishnamurthy is approved:

Chair Date

Date

Date

University of California at Berkeley

1999

Compiler Analyses and System Support for Optimizing Shared Address Space Programs

Abstract

Compiler Analyses and System Support for Optimizing Shared Address Space Programs

by

Arvind Krishnamurthy
Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor Katherine Yelick, Chair

Within the parallel computing community, it is widely recognized that a simple, yet efficient programming model is the key to making large scale parallelism usable, but there is no general agreement on what that programming model is. While message passing is the model of choice for those interested in high performance, the inability to build and manipulate shared data structures is a serious limitation for programs with complex data structures. In this thesis, we show that a combination of language, compiler, and runtime support can provide a shared address space model, which is competitive in performance and more flexible for programming.

We first introduce new compiler analyses and optimizations for explicitly parallel programs that communicate through a shared address space. Any type of code motion on explicitly parallel programs requires this analysis to ensure that operations reordered on one processor cannot be observed by another. The analysis, called *cycle analysis* checks for cycles among interfering accesses. We improve the accuracy of the analysis by using additional information from *synchronization analysis*. We demonstrate the use of this analysis by optimizing remote access on distributed memory machines by automatically transforming programs written in a conventional shared memory style into a Split-C program, which has primitives for non-blocking memory operations and one-way communication. The optimizations include *message pipelining*, to allow multiple outstanding remote memory operations, conversion of two-way to one-way communication, and elimination of communication through data re-use.

We then consider the issues involved in the implementation of Split-C's communication primitives on a variety of large-scale distributed memory machines, such as the Thinking Machines CM-5, the Cray T3D/T3E, Intel Paragon, Meiko CS-2 and workstation clusters. We map the language primitives onto those hardware primitives that correctly implement the desired semantics. On machines where there are several implementation strategies for realizing a language primitive, we use a systematic micro-benchmarking methodology to choose the best option. Our implementations reveal several crucial issues, including protection, address translation, synchronization, responsiveness, and flow-control, which must be addressed differently under the different regimes and contribute significantly to the effective communication costs in a working system.

The performance improvements from our compiler optimizations are as high as 20-35% for programs running on the Berkeley-NOW cluster. On machines with lower communication startup and longer relative latencies, the benefits from our compiler optimizations would be even higher. Our work on runtime systems demonstrates that our communication primitives can be implemented efficiently on a wide range of distributed memory machines. In particular, we have developed implementations that attain the best round trip latencies on each one of our target machines. Our results therefore demonstrate the feasibility of our approach of having the compiler and runtime system work in cohesion to provide an efficient and portable shared address space abstraction.

Professor Katherine Yelick
Dissertation Committee Chair

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A common theme in computing is to develop through compiler and runtime support a programming abstraction that is more useful than that directly supported by the underlying hardware. While the requirements of the programming abstraction are primarily shaped by the consideration of ease of programming, issues of economics and scale influence the development of hardware resulting in discrepancies. In the domain of parallel computing, it is widely accepted that the abstraction of multiple threads interacting through shared memory provides a high degree of programmability without sacrificing on performance. However, scalability issues have resulted in the development of machines with physically distributed memory modules with fairly weak guarantees on the ordering of inter-processor interactions. Our effort reconciles the reality of distributed memory machines with the programmers' need for a shared address space. In this thesis, we address the problem of developing a shared address space abstraction for explicitly parallel applications that provides high performance and clean semantics while enhancing the programmability of many of the commercial parallel machines developed in the last decade.

Our research is influenced by certain trends in the architecture of large scale parallel machines. To achieve scalability of hundreds or thousands of processors in a parallel machine, a primary design guideline is to minimize the interaction between the processors and allow independent execution. A natural consequence is to have separate modules containing a processor (or a small number of processors) and its associated memory, with an interconnection network to support communication between the modules. Such an approach avoids the bottleneck of having all the processors access a single large memory, which would inevitably have higher access latencies and greater contention. Several architectures have demonstrated practical scalability using this approach beyond a thousand microprocessors, including the nCUBE/2, Thinking Machines CM-5, Intel Paragon, Meiko CS-2, and Cray T3D/T3E. Recently, researchers have also demonstrated high performance communication in Network of Workstations (NOW) using scalable switched local area network technology [64, 13, 24]. To minimize the cost of inter-processor interactions, these machines provide varying levels of architectural support for low latency communication. Furthermore, a variety of software techniques, such as the Active Messages mechanism [65] and Fast Messages [49], have been developed to minimize the software overheads in utilizing the communication network. Consequently, in spite of the loose coupling provided by the interconnection networks, it is feasible to implement low overhead communication operations. In this thesis, we will see how such low overhead operations allows for the development of software layers that can simulate shared memory on top of distributed memory modules.

The other significant architectural trend that motivates our research is the weakening of the guarantees provided by the hardware in executing a parallel program. For parallel programs, an intuitive model of execution, known as *sequentially consistent execution*, is one in which the parallel execution behaves as if it were an interleaving of the sequences of operations from each of the processors. The result of such an execution appears as if a single processor executes the parallel program, executing instructions from each process in some alternating order. In such an execution model, memory operations appear to execute in an atomic fashion in the order specified by the program. However, as we will observe in greater detail in a later chapter, the processor cannot provide such strong guarantees on program execution without lowering the overall system utilization. Most modern multiprocessor systems, therefore, weaken the requirement of atomic execution of operations and allow potential reordering of program operations. In such execution models, commonly referred to as *relaxed consistency models*, one process may observe another process's operations happening out of program order. These models are complex and lack the intuitive simplicity of sequential consistency. In this thesis, we will develop techniques for providing sequential consistency semantics to the programmer without lowering system performance. In general, these techniques may also be used to implement a stronger consistency model on top of a weaker one.

In summary, the primary goal of this research is to develop an efficient shared address space abstraction with sequentially consistent semantics on scalable multiprocessors. Our research is based on many important premises, which we validate and prove in this thesis. Fundamental to our effort is the hypothesis that a system could employ compile-time techniques to provide a shared address space abstraction with strong semantics on platforms with relaxed execution models. Our approach requires the compiler to be capable of analyzing parallel constructs in an explicitly parallel program. Equally critical to the success of our approach is the ability to provide efficient communication primitives primarily through software techniques on message passing machines. Our primitives must have sufficiently low overheads to provide the illusion of tightly integrated shared memory even though the communication is achieved using messages between the memory modules. In this dissertation, we demonstrate the feasibility of our approach by providing evidence in the form of prototype compilers as well as robust runtime systems for a variety of parallel machines.

1.1 Overview

In this thesis, we describe compiler and system support for parallel programming on distributed memory multiprocessors. We present program analysis techniques and optimizations for explicitly parallel programs that communicate through a global address space and the implementation of a prototype compiler that applies these optimizations to explicitly parallel programs. Our compiler generates code for a variety of parallel machines using a highly optimized runtime layer that is responsible for moving data within the global address space. We demonstrate that to achieve good performance, while preserving clean semantics, the language, the compiler, and the runtime system have to work in concert.

The primary feature of our source languages is an explicit global address space abstraction, which provides the illusion of cache-less shared memory on distributed memory machines. We consider SPMD (Single Program, Multiple Data) dialects of Java and C. The language features themselves fall somewhere between higher level data-parallel and implicitly parallel languages and lower level message-passing systems in terms of ease of programming. High performance is feasible precisely because it is an explicitly parallel, middle-level language. Since the parallelism is explicit, the most important coarse grained parallelism is exposed by the programmer. Section 1.2 compares our language approach to others.

Our compiler analysis is novel in its use of synchronization analysis to identify the access patterns of concurrent threads. This information is critical in introducing optimiza-

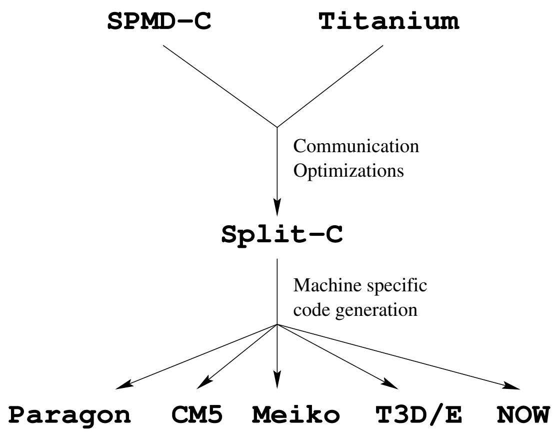


Figure 1.1: A two-step translation: the first step involves introduction of communication optimizations for programs written in SPMD dialects of Java and C, while the second step tackles the efficient implementation of the Split-C communication primitives generated by the first step.

tions that replace heavy-weight synchronous communication by less expensive asynchronous counterparts. We discuss the issues involved in translating our source languages into Split-C [25], a language with explicit low-level communication operations. We use Split-C as a generic abstract machine to illustrate the various applications of our compiler analyses. While this thesis explores the use of these analyses for optimizing programs written using a shared memory programming style, the compiler techniques are equally applicable for message passing programs as well. Section 1.4 gives an overview of the new analysis required for parallel programs, while Section 1.5 describes the optimizations that are enabled by the analysis.

The role of the runtime systems is to implement the low-level communication operations required by our optimizing compiler on various MIMD distributed-memory machines with workstation-level processor nodes. This class includes machines like the Thinking Machines CM-5, Cray T3D/T3E, Intel Paragon, Meiko CS-2 and workstation clusters, such as the one developed by the Berkeley NOW project. We consider the issues involved in the implementation of the Split-C primitives on these machines to complete our task of developing an efficient shared address space abstractions. Split-C's communication primitives are designed to support languages that distinguish global and local memory operations through the use of shared or global annotations on variables. We consider a variety of implementation strategies with varying degrees of specialization to the underlying hardware. We demonstrate that a small set of predefined communication primitives, which are different from the traditional send-receive primitives, can provide high performance without sacrificing flexibility. Our implementations also provide us with a context for evaluating the architectural support for communication on our target machines. Section 1.6 gives an overview of the implementation issues for a light-weight communication layer that underlies our language implementation.

Our efforts target users of high performance computing, which includes most programmers of scalable machines. These programmers currently use message-passing, either because their problems are too irregular to fit a strict data-parallel model, or because the performance of code generated by higher level languages has been disappointing. Data parallel languages, such as High Performance Fortran (HPF), target regular applications and consequently do not support applications with asynchronous communication or irregular data structures, such as dynamic meshes or unbalanced trees. Even for regular applications, optimizations are limited by layout analysis across procedure call boundaries, executable alignments, and other high-level features. Our approach avoids these compilation problems by using a simple layout mechanism and giving the programmer control over data reorganization and scheduling. Thus, one has the control of message-passing with the additional ease of programming that comes from a global address space.

1.2 Why Compile Explicitly Parallel Programs?

The key features of our language are: (1) explicit parallelism that conforms to the SPMD model of computation without dynamic thread creation, (2) a global address space for building shared data structures, (3) a type system that distinguishes local and remote data, (4) support for programmer control over data-layout and scheduling, and (5) linguistic support for explicit synchronization operations. Our language, therefore, provides a research context that is markedly different from research efforts such as parallelizing compilers, data parallel computing, or even message passing solutions for explicitly parallel programs.

There has been a large investment in compiler technology for automatic parallelization. The work in this community is aimed primarily at the discovery and exploitation of implicit parallelism in sequential scientific codes, and in particular the analysis and transformation of loop nests written in Fortran. Currently available systems perform best on programs that express dense matrix algorithms, but even on this limited domain the parallelism is often insufficient to fully utilize a modern distributed memory parallel machine.

Static compiler analyses for parallelization suffer from the inaccuracies that arise from data-dependence analyses resulting in conservative decisions at compile-time. Consequently, the compiler might have to generate sequential code for significant chunks of the application. Compiler technology to automatically or semi-automatically parallelize large, complex applications (instead of single algorithms) and to handle irregular or sparse algorithms (instead of regular, dense algorithms) is still in the earliest stages of development.

The numerous efforts in parallel language design can be divided roughly into the following categories: implicitly parallel, data parallel, message-passing, and shared-memory languages with dynamic thread packages. Parlog [20] and Sisal [62] are examples of implicitly parallel languages, and while programs in these languages naturally contain large amounts of parallelism, they share some of the difficulties of automatic parallelization. Although the discovery of parallelism is not an issue, sophisticated compiler and runtime support is required to manage resources and eliminate inefficiencies of the high level language constructs. Important decisions such as data layout and scheduling are still made by the compiler with non-optimal decisions resulting in considerable performance penalties.

The most widely used data parallel languages, Fortran-D [32], CM-Fortran, C* [51], and High Performance Fortran (HPF) [31], are variants of Fortran with linguistic support for data parallelism and user-controlled array layouts for distributed memory. These languages are primarily designed to handle regular applications. The only distributed data structures are multidimensional arrays, and the data-parallel control model does not allow different processors to work on different kinds of subproblems. Applications with irregular meshes, sparse matrices, or unbalanced trees are not directly supported in the language. Also, the kind of asynchronous interactions that are required by programs with control parallelism cannot be easily expressed in these languages. Finally, in many of these languages, the compiler is still responsible for the discovery of coarse-grained tasks from the fine-grained parallelism expressed in the programs. The compiler analysis for these optimizations requires identifying dependencies and interactions between adjacent parallel loops, a task which could be as complex as identifying parallelism in a sequential program.

Current message-passing systems are not integrated languages—rather, they are standard sequential languages with libraries of parallel procedures. Because the parallel operations are not part of the programming language, it is impossible for a compiler to perform substantial analysis or optimization wherever the parallel operations are used. The MPI design [28] embodies many of the ideas from other message-passing systems. MPI is designed as an end-user library for performance programmers, and one of its main goals is portability. Because it omits machine-specific features and also incurs overheads such as buffer copies [65], it gives up some performance. Our system offers portability and higher performance via a compilation phase in which the parallel operations and machine-specific features are exposed to program analysis and optimization.

1.3 Compiling Parallel Programs with Sequential Compilers

Most parallel programs are written using either message passing or thread libraries, and are compiled with a standard sequential compiler such as GNU's **gcc** or vendor supplied C or Fortran compilers. The other popular approach to compiling explicitly parallel programs is to modify the frontend of an existing compiler to translate the parallel constructs into calls to library routines, which are inserted during construction of the parse tree [18, 15, 14, 17]. The parallel constructs of the language still remain encapsulated in a runtime library. Strictly speaking, this is not a separate pre-processor, but the effect is almost the same. Compiling parallel programs with such compilers has several problems that we address in our research. In both approaches, the optimization and code generation phases of compilation have no knowledge of the parallel semantics of the language. Instead, operations such as remote memory operations are treated as black box library calls. Thus, many opportunities for program optimization are missed, which results in suboptimal performance.

The first problem with using a library or preprocessor approach is that the parallelism constructs themselves are not optimized. For example, consider the implementation of a non-blocking remote memory read, which we will refer to as a *get* operation. Completing the *get* requires a round-trip message through the network; meanwhile, the processor does not block, but continues with the next operation after the *get*. The process blocks only when the program reaches a synchronizing operation, known as *sync*. When the *sync* is executed, the processor waits until the outstanding *get* completes. An optimizing compiler can reduce the waits due to synchronization by scheduling *get* operations as early as possible in the instruction stream, and by scheduling a *sync* operation as late as possible. Performing these optimizations is not a trivial task. To do it correctly, a compiler must account for possible dependences between the memory operations as well as resource constraints, which requires analysis of a parallel program graph. Such an optimization is not possible when the compiler is developed primarily as a preprocessing tool.

The second problem with using a preprocessing frontend to a sequential compiler is that many traditional optimizations employed by sequential compilers are incorrect in an explicitly parallel setting. Optimizations that perform code motion, such as common subexpression elimination and constant propagation, might be incorrect for an explicitly parallel program due to inter-processor interactions that might access or modify program variables in a non-deterministic manner. In Chapter 2, we will examine many instances of incorrect optimizations and the techniques required to avoid them.

Another problem with using a sequential compiler for a parallel language is that the parallel extensions actually inhibit conventional sequential optimizations. To the sequential C compiler, the parallel primitives of the language appear as library function calls. Because C compilers can assume nothing about the behavior of library functions, many standard sequential optimizations cannot be performed across library function call boundaries. A compiler that understands the semantics of the primitive parallel operations would not have this defect.

It is worth noting that the problems caused by using sequential compilers for parallel programs are not unique to shared memory languages. In fact, these same problems exist for the community writing message-passing programs today, and to some extent for higher level compilers that generate message-passing C or Fortran. Existing message-passing systems are simply libraries of function calls. When a programmer writes message-passing code, and compiles it using a sequential compiler, performance suffers because the quality of the code generated is poor for the reasons outlined above. In fact, one of the major complaints of parallel application writers is that current compilers generate poor "node code" (the program that runs on a single machine). High-level language compilers, such as the Fortran-D/Parti system perform optimizations like message pipelining, but because they generate SPMD C or Fortran, scalar optimizations in the C or Fortran compiler are still limited by the presence of library calls.

Our approach successfully addresses many of the difficulties faced by these systems. The global address space abstraction is made explicit in the language. Therefore, our compiler is aware of the semantics of operations that result in communication events. The optimization of remote memory operations is integrated with the corresponding transformations on code operating on local data. The remote memory operations themselves are implemented in many different forms each with different cost functions, and the compiler is responsible for choosing the most efficient communication primitive while preserving program semantics. The remote memory operations are not implemented on top of a generic communication layer. Instead, the runtime layer is specialized to the underlying machine and adds very little overhead to the raw machine performance. With the compiler and the runtime system interfacing through a set of well-defined operations, the design provides a high degree of adaptability to changes in machine parameters and application characteristics.

Although some parallel shared-memory languages have separate compilers (as opposed to preprocessors), most of these systems treat parallel constructs as opaque library calls. There is little previous work on optimizing explicitly parallel programs and the work that has been done is mostly foundational. Shasha and Snir provide the basis for our work on memory optimizations, although they consider only straight-line code and did not implement their algorithm [58]. Midkiff and Padua [47, 46] surveys the basic problems; a paper by Callahan presents optimizations for programs with limited synchronization constructs [16]. None of these groups has addressed optimizations of remote memory, and there is no complete implementation of which we are aware.

1.4 Program Analysis for Parallel Programs

It is widely recognized that *dependence analysis* is the basic analysis required for optimizing parallel programs. Dependence information can be represented as a graph where the nodes are program statements and there is an edge from X to Y if X precedes Y in the program text, both X and Y may reference a common memory location l, and at least one of X or Y writes l. The dependence graph gives a conservative estimate of the necessary order of execution of program statements: if there is an edge from X to Y then X should execute before Y. Any execution of statements that preserves the dependence graph will preserve program semantics.

There are many variations on dependence analysis to meet the needs of different situations. Historically, most of the research effort in dependence analysis has been motivated by the goal of automatically parallelizing sequential FORTRAN programs. Thus, it is not surprising that the best understood case of dependence analysis is the analysis of array subscripts in sequential FORTRAN loop nests. The situation for an explicitly parallel language is much different and presents its own set of interesting problems.

The problem of dependence analysis of explicitly parallel code has received very little previous attention, except for the restricted case of data parallelism. The introduction of unrestricted explicit parallelism adds a new dimension to dependence analysis. Not only is it necessary to consider the dependence relationships between statements executing within a single process, it is also necessary to consider the shared accesses, called *conflicts*, between statements executing on different processes. Because the language is asynchronous, the execution order of statements executing on different processors is not predetermined. The only constraints on the relative order of statements on different processors arise from program synchronization points. Thus, to carry out effective and accurate data dependence analysis it is necessary to utilize information about process synchronization. In this thesis, we explore a variety of techniques for determining data dependency information in an explicitly parallel context. We use synchronization analysis to obtain an accurate compile-time prediction of the conflicts that might occur during run-time. The information gathered from this analysis is used for a variety of program transformations such as masking the latency of remote memory operations. We now present an overview of the optimizations enabled by our analyses.

1.5 Compiler Optimizations for Parallel Programs

The principal drawback of 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 are implemented using messages through an interconnect network and are thus high latency operations. On distributed memory machines, the cost of these operations vary from 100 cycles on the T3D[7] to 400 cycles on the CM5[65] and 1000 cycles on the Berkeley NOW. Transformations that either hide this latency or eliminate some of the messaging produce substantial benefits. Though our focus is on optimizing global address space programs for distributed memory machines, we expect our analyses to be useful for a wide range of explicitly parallel programs, from shared memory programs on shared memory multiprocessors to message-passing programs on large scale distributed memory machines.

Prefetching and Pipelining. Remote read operations require a round-trip message through the network. Meanwhile, the processor need not block if the result of the *read* is not required for the next operation; it can continue with the next operation after the *read*. An optimizing compiler can reduce the waits due to synchronization by transforming a blocking *read* into a non-blocking *get* followed by a *sync*, by scheduling *get* operations as early as possible in the instruction stream, and by scheduling a *sync* operation as late as possible. Remote writes typically require a round-trip message as well. The reason is to ensure that the write is complete before subsequent operations are performed. In particular, a round trip is the simplest way to ensure that writes appear in program order. However, in certain situations, blocking *writes* can be transformed to non-blocking *puts*, resulting in pipelined communication.

Eliminating Acknowledgements. A write request requires a response to indicate the completion of the remote store operation. However, in certain situations the acknowledgement of completing the operation is not required by the program. The program can continue execution without requiring the knowledge of when the write actually happens. Especially for programs written in a bulk synchronous programming style, writes do not have to complete until end of a phase at a global synchronization point. In such situations, the response message could be eliminated, thus transforming the round-trip protocol into a more efficient one-way communication. By this transformation, we not only reduce network contention by reducing the number of packets but also reduce the processing time spent by the processors in generating and handling the acknowledgement traffic.

Eliminating Remote Accesses. A read of a remote value that has recently been read could be eliminated if the compiler had saved the value previously read. Also, reading a remote value that has recently been written by the same processor can be avoided if the written value is still available. Similarly, when a thread issues two successive writes to the same variable, the earlier writes can be buffered in a local variable and the final value written to the remote copy at a later point. An important variant of this optimization is communication aggregation, where accesses to adjacent locations are combined into a single communication event.

In addition to these transformations, our analyses enable various other optimizations that are based on code-motion. In sequential languages, traditional optimizations such as common-subexpression elimination are considered correct if they respect the local data-flow dependencies. For explicitly parallel programs, the optimizations need to respect inter-processor data dependencies as well. While our prototype implementation does not provide such transformations, our analyses are necessary for correctly implementing such optimizations in a parallel setting.

1.6 System Support for Communication

While our compiler is responsible for choosing the communication primitives that preserve program semantics, there still remains the task of obtaining efficient implementations of the communication primitives. We have developed highly optimized implementations of our communication layer for all of our target platforms. Our communication system targets six important large-scale parallel platforms that have varying degrees of architectural support for communication: the Thinking Machines CM-5, Intel Paragon, Meiko CS-2, Cray T3D/T3E, and Berkeley NOW. The CM-5 provides direct user-level access to the network, the Paragon provides a network processor (NP) that is identical to the compute processor (CP), the Meiko and NOW provide an asymmetric network processor that includes the network interface (NI), and the Cray Machines provide dedicated hardware, which acts as a specialized NP for remote reads and writes. Against these hardware alternatives, we consider a variety of implementation techniques for remote memory operations that our compiler will generate. These range from general purpose active message handlers to specialized handlers executing directly on the NP or in hardware.

In most sophisticated computing systems, there is a gap between the performance intended by the designer and that demonstrated by the actual hardware, and a larger gap between the machine performance and that extracted by a language implemented on the machine. The first arises due to interactions between components of the system, unanticipated dependencies, and the pragmatics of the actual implementation. The second compounds the first with the costs of gaining access to the hardware primitives within the code-generation strategy of the compiler, while respecting the semantics of the language. The language must gain access to the most efficient hardware primitives within the code-generation strategy of the compiler, while respecting the semantics of the language. Therefore, an architecturally interesting system poses many difficulties in the language implementation and conflicts with the goal of having a compiler that can easily be ported between platforms. Our implementations therefore reveal several crucial issues, including protection, address translation, synchronization, responsiveness, and flow-control, which must be addressed differently under the different regimes and contribute significantly to the effective communication costs in a working system.

Our runtime systems work is largely orthogonal to the implementation of communication mechanisms in distributed shared memory machines, which seek to avoid unnecessary communication by exploiting address translation hardware to allow consistent replication of blocks throughout the system [35, 38, 39, 42], and operating system studies, which seek the same end by extending virtual memory support [43, 4, 19]. In these efforts, communication is caused by a single load or store instruction, and the underlying hardware or operating system mechanisms move the data transparently. We focus on what happens when the communication is necessary. So far, distributed shared memory techniques have scaled up from the tens of processors toward a hundred, but it is widely accepted that the thousand processor scale will be reached only by clusters of these machines in the foreseeable future. Our efforts overlap somewhat with the cooperative shared memory work, which initiates communication transparently, but allows remote memory operations to be serviced by programmable handlers on dedicated network processors [39, 50].

1.7 Contributions

In this thesis, we describe the implementation of a parallel language using novel compiler techniques and highly optimized communication layers on a variety of machines. The contributions of this thesis are:

- Design of compiler analyses for inter-processor interactions in a parallel program. Our research is the first instance of using inter-processor dependencies and synchronization information for performing correct communication and code-motion transformations of explicitly parallel programs. Our analyses techniques are efficient as they exploit the common code image property of our source programs.

- Implementation of optimizing transformations in the context of compilers for SPMD dialects of C and Java. Our backend generates code for the low-level communication model defined by the Split-C language. We present performance results for programs written in the SPMD dialect of C.

- Efficient implementations of the global address space constructs on many scalable machines including the Thinking Machines CM-5, Intel Paragon, Meiko CS-2, Cray T3D/T3E, and Berkeley NOW. Our implementations also expose some of the issues involved in the design of architectural support for communication.

The remainder of this dissertation is organized as follows. In Chapter 2, we develop the theoretical framework for enforcing the correctness criteria for a parallel execution. In Chapter 3, we examine the compiler techniques required to apply the theoretical results developed from the preceding chapter. Chapter 4 details the compiler transformations used in generating code for the Split-C backend. Chapters 5 and 6 focus on the issues involved in obtaining an efficient implementation of Split-C on a variety of parallel machines. Chapter 7 concludes.

Chapter 2

Machine Models for Parallel Program Execution

In this chapter, we develop the theoretical framework required by our compiler analyses. We begin by discussing the problems that arise from the reordering of operations in a parallel program due to either compiler optimizations or hardware mechanisms. We then develop novel algorithms that allow us to determine when reordering is correct. We consider the interleaving behavior of operations from concurrent threads to identify situations where reordering does not affect program correctness. We also analyze synchronization constructs to identify the precedence or mutual exclusion constraints imposed on program execution. Information gathered from this analysis increases the applicability of compiler optimizations.

2.1 Reordering of Operations in a Parallel Program

The execution model for most sequential languages is that the operations that constitute the program are executed in the order specified by the program. However, both hardware and compilers for sequential computers allow reordering of the program operations if the resulting execution is indistinguishable from an execution that preserves the program order. Typically, this notion of correctness requires reasoning about operations that either cause or observe side-effects such as assignment (write to) or evaluation of (read from) program variables. A sufficient and yet simple set of guidelines for reordering operations while preserving correctness are:

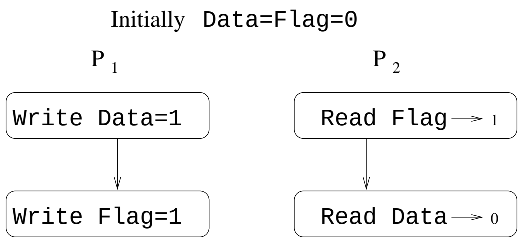


Figure 2.1: Violation of sequential consistency: P_1 stores the value 1 into variables *Data* and *Flag*. On P_2, the read of *Flag* returns 1, while the read of *Data* returns 0.

- The execution preserves the program order of assignments to a particular program variable.
- An evaluation of a program variable occurs only after the execution of all previous assignments to the same variable and before the execution of all subsequent assignments.

For parallel programs, these guidelines must be met for each process in the program; we will say that such programs satisfy their *local dependencies*. In addition, the programmer often intuitively relies on the notion of *sequential consistency*: the parallel execution must behave as if it were an interleaving of the sequences of operations from each of the processors [40] with the individual sequences preserving the per-processor program order. If only the local dependencies within a processor are observed using the guidelines mentioned above, the program execution might not be sequentially consistent. Midkiff and Padua [47] describe eleven instances where a uniprocessor compiler would generate either incorrect or inefficient code. In particular, if one of the threads of a parallel program were compiled using a compiler for a sequential language, the compiler could introduce certain transformations that violate sequential consistency at run-time.

An example to illustrate sequential consistency is shown in Figure 2.1. The program is indeterminate in that the read of `Flag` may return either 0 or 1, and if it is 0, then the read to `Data` may return either 0 or 1. However, if 1 has been read from `Flag`, then 1 must be the result of a subsequent read from `Data`. If the two program fragments were analyzed by a sequential compiler, it might determine that the reads or writes could be reordered, since there are no local dependencies. If either pair of the accesses is reordered, the execution in which the read of `Flag` returns 1 and the read of `Data` returns 0, might result. Compiler optimizations based on code motion could produce such incorrect code.

Most optimizations involving code motion may result in such incorrect executions, whether the programs run on physically shared or distributed memory and whether they have dynamic or static thread creation. Automatically parallelized sequential programs or data parallel programs with sequential semantics do not suffer from these pitfalls, because every pair of accesses has a fixed order, which can be determined at compile-time. The additional problem for explicitly parallel programs comes directly from the possibility of non-determinism, whether or not the programmer chooses to use it.

2.1.1 Hardware Reordering Mechanisms

The compilation problem is compounded by the reordering mechanisms employed by the modern microprocessors that are used in most parallel machines. Even if the compiler does not reorder the shared memory accesses, reordering may take place at many levels in a multiprocessor system. At the processor level, since a superscalar may issue an instruction as soon as its operands are available, writes to different locations might be issued out of program order. Most processors have write buffers, which allow read operations to overtake preceding write operations. Reordering may also take place at the network level in distributed memory multiprocessors, because some networks adaptively route packets to avoid congestion. Also, two accesses sent to two different processors may be handled out of order, since network latencies may vary. The underlying motivation behind such architectural features is to increase efficiency by exploiting all of the available parallelism within a processor and between the processor and network. Since these processors were built primarily to be employed in sequential computers, the processors typically only guarantee that the local dependencies are observed.

Fortunately, machines usually come with support to enforce ordering constraints, such as a memory barrier instruction or a write-buffer flush to enforce ordering between memory operations, or a test for completion of a remote operation. However, these new instructions, which are commonly referred to as *fence insructions*, must be explicitly inserted by the compiler or the programmer. If a standard uniprocessor compiler is used for generating code, these special instructions would not be automatically inserted.

In summary, the seemingly pragmatic choice of using an uniprocessor compiler to compile an individual thread of a parallel program could result in incorrect executions. A partial solution to this problem is to disable certain kinds of optimizations involving code motion or register allocation during the compilation process. Since most compiler optimizations are built using code motion as a fundamental transformation, this solution is clearly unacceptable from a performance perspective. Also, this solution does not prevent runtime reorderings introduced by the processor.

2.1.2 Relaxed Consistency Models: A Partial Solution

The traditional approach to resolve these compilation problems has been to define "relaxed" correctness criteria for excution of parallel programs. In these weaker models, a parallel execution does not have to be an interleaving that preserves the ordering of operations from each of the individual threads. Instead, certain operation could execute before operation appearing earlier in the instruction stream on the same thread. A *relaxed consistency model* is a contract provided by the hardware and the compiler as to when the system would preserve the program order and when it might not. The possibility of reordering is exposed to the programmer. If certain reorderings need to be prevented, the programmer needs to introduce fence operations explicitly into the application code.

The research community has proposed a large set of weaker consistency models in the last decade [29, 1, 11, 12, 33]. The primary challenge addressed by many of these efforts is to provide the strongest possible guarantee on execution while ensuring that as many of the hardware optimizations are captured by the potential reorderings allowed by the contract. It is, however, widely recognized that programming using relaxed consistency models has many pitfalls. Most of the difficulties associated with programming with such models are due to the complexity of reasoning about the program logic in the presence of certain kinds of reordering. In fact, many programs are written assuming that the underlying system does not reorder critical accesses. In a recent study [54], the authors discovered that many well known systems such as the Altavista search engine and the Petal File System would run incorrectly on platforms that do not support sequential consistency.

Even if the programmer is cognizant of reordering by the machine, relaxed consistency models are very hard for programmers to understand. Furthermore, there are very few programming tools to aid the development effort especially when writing an explicitly parallel program using standard uniprocessor development tools. If a traditional uniprocessor compiler such as *gcc* is used for compilation, the only technique available to the programmer is to use "volatile" annotations to disable certain optimizations. While such annotations inform the compiler to treat the annotated variables as special uncacheable variables, they rarely translate into the appropriate fence operations. A more robust approach has been to embed the fence operations in library functions, which have to be appropriately invoked by the programmer. However, these techniques are mostly ad-hoc and do not aid the programmer in reasoning about possible execution scenarios. While most relaxed consistency models include a classification of program operations (such as labeled operations in the release consistency model), there are currently no programming tools for expressing such attributes about program variables.

For software developers, there is the additional complexity of producing software that works across platforms that support different consistency models. In spite of considerable research in this area, there is no consensus on a particular consistency model, and as a result, each architecture defines its own consistency model. This lack of consensus further exacerbates the programming difficulties.

In this thesis, we present a different approach. Our system presents a sequentially consistent execution model to the programmer. However, our system does not suffer from the performance pitfalls traditionally associated with sequential consistency. Our compiler performs many of the reordering optimizations while still exploiting the hardware features for masking the cost of high latency operations. The fundamental enabling technology in achieving this result is the use of compiler analysis of both cross-processor accesses to shared variables and of synchronization constructs. Our compiler analyzes all of the threads of a given parallel program to identify when reordering of operations could actually be observed. If the compiler can prove that reordering would not be observed and hence not affect program correctness, it is free to perform code motion or to generate code that allows the processor to reorder instructions dynamically. Conversely, the analysis is used to introduce fence instructions that are necessary to prevent incorrect reorderings by the processors.

2.2 Analysis of a Parallel Execution

In the remainder of this chapter, we consider the execution of a parallel program under different machine models. We consider parallel execution on systems that directly support sequential consistency as well as systems with weak memory operations such as non-blocking load and store operations synchronized using machine-specific fence operations.

We begin by studying the correctness properties of a parallel execution, where an execution is specified as a collection of instruction sequences, one sequence for every thread in the parallel program. Each instruction is either a memory access (which could be a *read* or a *write* of a program variable) or a simple arithmetic operation. There are no instructions for transfer of control. One could view an instruction sequence as a trace of program execution where all the control branches have been resolved and the addresses of accessed program variables have been computed.

Ultimately, we are interested in analyzing programs written in high level languages. The motivation behind examining a parallel execution, which is a dynamic notion based on low level instructions, is to separate the correctness aspects of a parallel execution on a particular machine model from the compiler issues of gathering information about a parallel program. We now briefly define the roles of two different mapping problems: 1) performing instruction selection that preserves correctness for a given execution and 2) conservatively approximating the possible runtime executions by compile-time information.

In this chapter, given a set of instruction sequences from different processors, we identify which of those operations may be replaced by non-blocking ones without changing the program behavior. If the use of non-blocking memory operations could result in incorrect executions, fence instructions are inserted into the instruction streams. We also consider the important problem of using as few fence operations as possible to avoid the performance degradation that could result from its indiscriminate use.

Given an understanding of the correctness properties of instruction sequences with precise information on variables accessed and operations executed during an execution, we will, in later chapters, solve the compiler problem of analyzing a static representation of the program. In particular, a statement in a program written using a high level language could correspond to many different machine instructions potentially accessing different program variables during program execution. We need to identify the kinds of instruction sequences that would be generated during runtime. We will examine the compile-time analyses as well as the linguistic features necessary to obtain accurate information using fast algorithms.

2.2.1 Basic terminology

In this section, we define the terminology that we will use in the rest of this document. This discussion is based on the framework originally developed by Shasha and Snir [58]. We refer the reader to their paper for proofs of some of the theorems presented in this section.

We analyze the execution of an explicitly parallel program specified by a fixed number of program segments, each of which is executed on a single processor. A parallel execution P on n processors is given by n sequences of instruction executions $P_1, \ldots, P_n$. Given a processor execution $P_i = a_1, \ldots, a_m$, we associate with P_i a total order $a_1 < a_2 < \ldots < a_m$. The *program execution order*, P, is the *partial order* formed by taking the union of these P_i's. A *partial order* is an irreflexive, asymmetric, transitive relation. A partial order is usually represented by a directed acyclic graph with edges depicting the ordering imposed by the relation.

In a *shared memory execution*, reads and writes happen atomically in the order they are issued, with no predetermined ordering between accesses issued by different processors. More formally,

Definition 1 *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 [40].*

This definition also results in the following formulation. An execution is sequentially consistent with respect to P if there exists a total order S of the operations in P, i.e., $P \subseteq S$, such that S is a correct sequential execution resulting in the same program behavior of read and write operations.

For example, in Figure 2.2, if the read to Y returns a new value written by P_1, then the read of X must also return the value written by P_1. If a machine has a correct implementation of atomic reads and writes to shared variables and also preserves the order in which instructions are executed, the resulting execution S satisfies the constraint $P \subseteq S$.

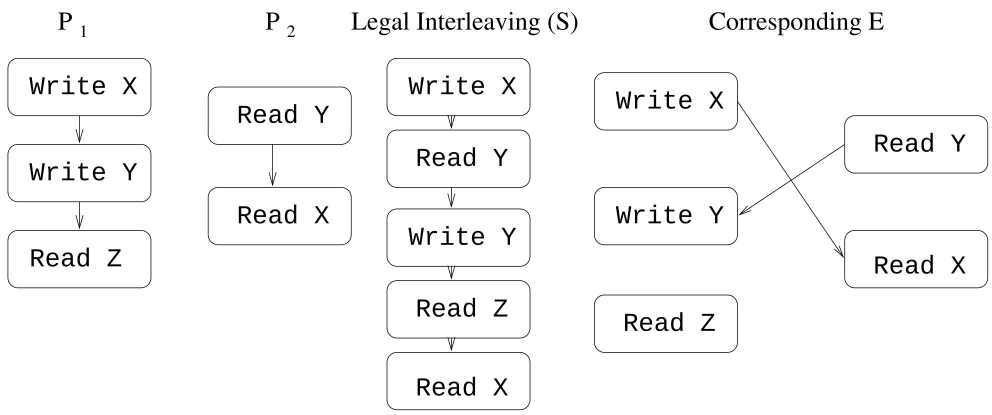


Figure 2.2: A shared memory execution and one possible total order S: Executions in which the Write to Y appears to happen before the Write to X would not be legal.

System Contract 1 *Given a shared memory program, a correct machine must produce only sequentially consistent executions for that program.*

S is a dynamic notion based on a particular execution, which captures the order in which operations from different threads interleave at program execution time. While S is comprehensive in exposing all the details of a given execution, a subset of the ordering relations is sufficient for our correctness properties. We define a total order E_v on the accesses issued to variable v and a partial order over all variables, E, which is the union of these E_v's. We observe that two different S orderings would produce the same results for read operations if the corresponding E relations do not differ. E, which we will refer to as the *execution order*, encapsulates all of the inter-processor interactions. One could therefore view E as the essential portion of S. While one could build a theoretical framework using S alone, having this separate notion of E allows for a simpler formulation. In particular, P and E serve as two different orderings on the operations of a parallel execution. P orders operations using a processor-centric view, while E orders operations using a memory-centric view. An execution order E corresponds to a sequentially consistent execution only when the graph corresponding to the relation $P \cup E$ is acyclic [58]. The edges of this graph represent relationships specified by either P or E and is therefore a general directed graph. A cycle is such a graph corresponds to conflicting ordering relationships of two accesses and indicates that the order in which the accesses were executed in memory is inconsistent with the order in which they were issued by the processor.

Definition 2 $E = \{[u, v] | [u, v] \in S$ *and* u, v *access the same variable* $\}$

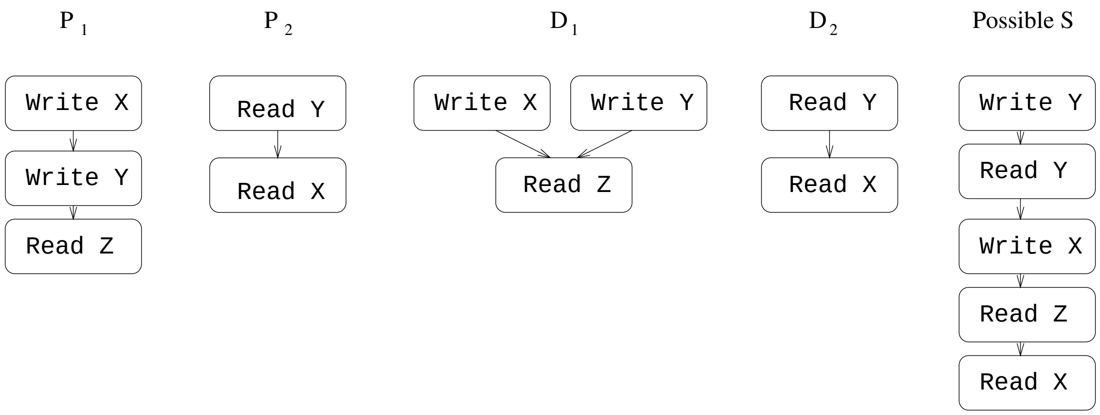


Figure 2.3: A weak memory execution with D containing some of the edges in P. Since the writes in P_1 are not ordered, they may execute in either order. However, an execution in which the read of Z appears to happen before either one of the writes would be illegal.

Theorem 1 *S is a correct execution of P if and only if S defines an execution order E such that $P \cup E$ is acyclic [58].*

System contract 1 does not specify the behavior of program executions when reads and writes are not atomic operations. We now consider a *weak memory* execution of parallel programs with non-atomic operations. In order to extend the system contract for executions with weak memory accesses, rather than relying on a particular instruction set with non-blocking memory operations and synchronizing accesses, we use a more general framework proposed by Shasha and Snir [58]. A *delay set* D specifies some pairs of memory accesses as being ordered, which says that the second operation must be delayed until the first one is complete. For example, in Figure 2.3, P_1 specifies the accesses issued by a processor, and D_1 specifies the delay constraints for executing the accesses. In general, given an execution P, D is a subset of the ordering given by P, i.e., $D \subseteq P$. A weak memory execution of a parallel program P with delay restrictions D is *weakly consistent* if there exists a total order S such that $D \subseteq S$ and S is a correct sequential execution of P. In the rest of this discussion, we will assume that the system guarantees correct execution of the delay restrictions.

System Contract 2 *Given a weak memory program, a correct machine must produce only weakly consistent executions for that program.*

2.2.2 Cycle Detection

In this section, we address the problem of enforcing sequential consistency while executing on a weak memory system. Given a set of instruction sequences, we determine which of those operations have to be ordered to ensure sequential consistency.

During a parallel execution, E is not explicitly recorded by the hardware or any other mechanism, but it is inferred based on the program behavior of reads and writes to shared variables. For weakly consistent executions, S is a superset of D and E. Since E and D are partial orders on the set of accesses issued during a parallel execution, there could be many different solutions for S that are consistent with E and D. In Figure 2.4 a possible execution of the two threads with no delay restrictions is one in which the read of x returns the new value and the read of y returns the old value. There are many possible interleavings that could result in such an execution including ones in which the ordering of writes is not preserved. When there is no possible interleaving that satisfies the property $P \subseteq S$, there is a violation of sequential consistency.

Figure 2.1, which is repeated here as Figure 2.5, illustrates an example of a weak memory system violating sequential consistency. A valid interleaving cannot be found since the "happens before" relation, which is $P \cup E$, contains a cycle. In this case, the figure-eight formed by $P \cup E$ is the cycle that violates sequential consistency. All violations are due to such cycles, although in general the cycles may extend over more than two processors. Recall that P orders accesses from a single processor while E orders accesses to a single variable. A cycle in $P \cup E$ implies that S cannot satisfy the dual requirements of being a total order as well as be a superset of P and E.

A cycle might manifest itself in $P \cup E$ only for certain kinds of instruction sequences. Figure 2.6 shows an example where a cycle does not occur for all four possible orientations of E. An S that satisfies the criterion $P \subseteq S$ can be found for any given E for which $P \cup E$ is acyclic, or equivalently, even if the read or write accesses are reordered, there exists a valid interleaving that results in the same program behavior. The program structure makes it impossible to distinguish a valid interleaving from one that violates sequential consistency.

We make certain observations about cycles that cause violations. The cross-processor edges in the cycle are between "conflicting" accesses, which are accesses to the same variable by two different processors such that the accesses produce different results for different orderings. A read-write or write-write pair could be termed as conflicting. A

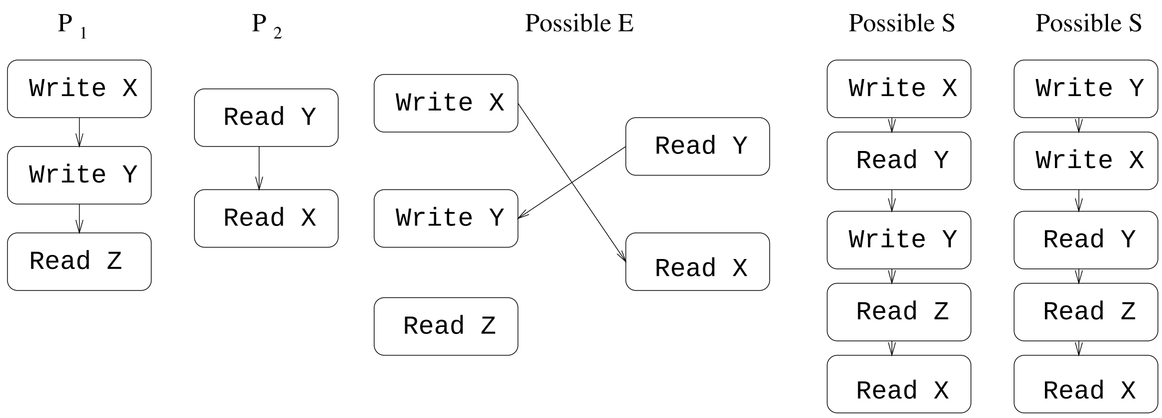


Figure 2.4: P, E, and S for a simple execution. E can be inferred from the values returned by reads and orders accesses from different processors. There could be multiple solutions for S given a particular E.

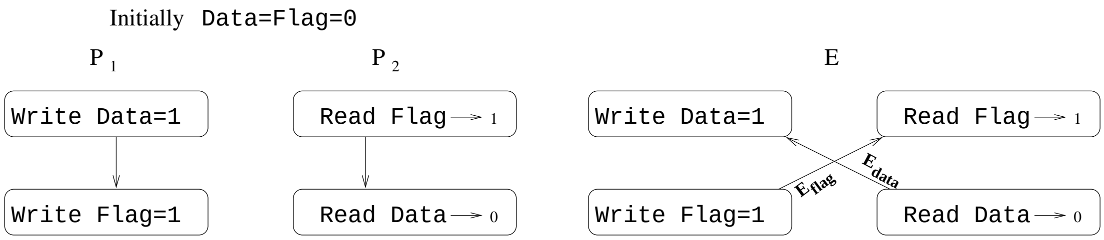


Figure 2.5: Possible violation of sequential consistency: If the read of $Flag$ returns 1, the read of $Data$ should see the new value.

read-read pair does not constitute a conflicting pair as there are no side-effects to distinguish the ordering of the reads. Similarly, writes from different processors that store the same value do not conflict.

We define the *conflict set* C to be a conservative approximation to these interferences: C contains all unordered pairs of shared memory operations a_1, a_2, such that a_1 and a_2 are issued by different processors, both access the same shared variable, and at least one is a write operation. C is an approximation as it is assumes that every write-write pair to be conflicting even though writes of the same value are not.

We now determine which accesses issued by a processor have to be ordered to ensure sequential consistency. We say that a delay set D is *sufficient* for P if, on any machine that satisfies the second system contract, all possible executions of P are sequentially consistent. Note that if we take D to be P, which means that we block on every remote memory access, it forces our machine to produce a sequentially consistent execution. Our goal during program analysis is to find a much smaller D that still ensures sequential consistency.

Shasha and Snir proved that there exists a delay set, $D_{S\&S}$, that is minimal such that no proper subset of $D_{S\&S}$ is sufficient to avoid violations of sequential consistency. Furthermore, they showed that $D_{S\&S}$ can be identified by considering the cycles in $P \cup C$. The primary idea is that if $P \cup C$ does not contain any cycles (as in Figure 2.7), then $P \cup E$ would not contain any cycles since $P \cup C$ is a conservative superset of $P \cup E$. We reformulate their result with the following definitions.

Definition 3 *A path $[a_1, \ldots, a_m] \in P \cup C$ is a* **simple path***, if for any access a_i in the path and P_k is the processor that performs a_i, then the following hold:*

1. *If a_{i+1} is also on P_k, then for all other accesses a_j ($j \neq i$ and $j \neq i+1$), a_j is not on P_k.*

2. *If a_{i-1} and a_{i+1} exist ($i \neq 1$ and $i \neq n$) and $[a_{i-1}, a_i] \in C$ and $[a_i, a_{i+1}] \in C$ (which means that a_{i-1} and a_{i+1} are not on P_k), then for all $j \neq i$, a_j is not on P_k.*

Thus, except for the end-points of the path, a simple path is one that visits each processor at most once, with at most two accesses per processor during a visit. A special case of simple paths points to a potential violation of sequential consistency.

Definition 4 *Given an edge $[a_j, a_i]$ in some P_k, a path $[a_i, a_1, \ldots, a_m, a_j] \in P \cup C$ is called*

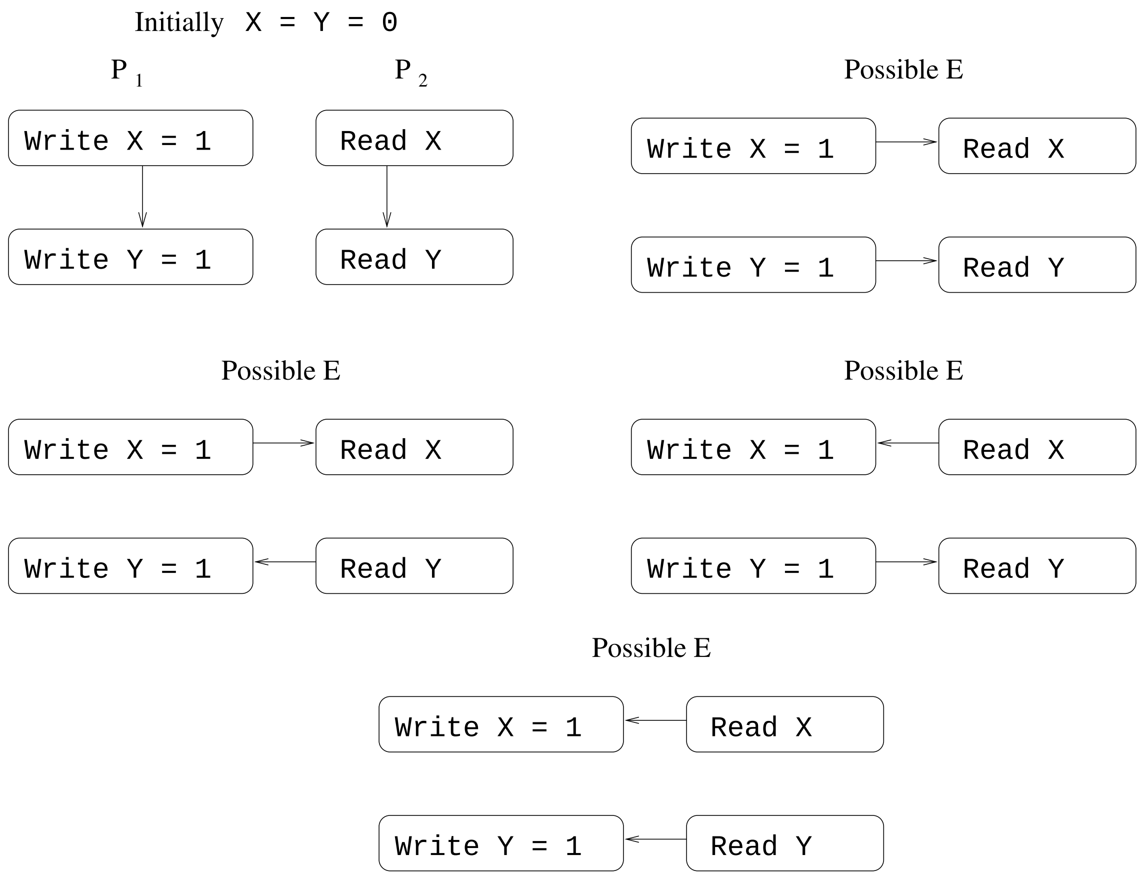


Figure 2.6: Example of a parallel program that does not have a cycle in $P \cup E$ for any orientation of E.

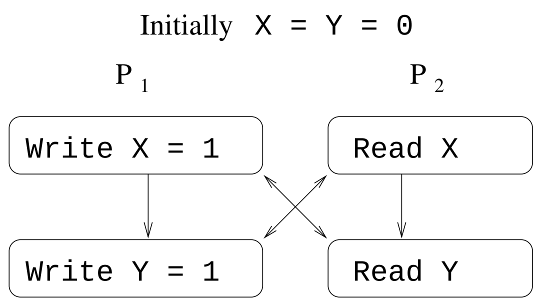


Figure 2.7: Example of a parallel program that does not require any delay constraints. C contains all unordered read-write or write-write operations on the same shared variable.

a back-path *for* $[a_j, a_i]$ *if* $[a_i, a_1, \ldots, a_m, a_j]$ *is a simple path. The cycle* $[a_i, a_1, \ldots, a_m, a_j, a_i]$ *is known as a* simple cycle*.*

Shasha and Snir define a particular delay set, denoted here $D_{S\&S}$, which is sufficient and minimal in certain respects. Their observation is that every instance of a violation of sequential consistency corresponds to a simple cycle in $P \cup C$. Informally, a back-path for $[a_m, a_1]$ is a potential execution sequence that allows other processors to observe a violation if accesses a_m and a_1 are reordered.

Definition 5 *Given a parallel program P and its conflict set C,* $D_{S\&S} = \{[a_i, a_j] \in P | [a_i, a_j]$ *has a back-path in* $P \cup C\}$*.*

Theorem 2 *[58]* $D_{S\&S}$ *is sufficient to produce sequentially consistent executions of P on systems with correct implementations of weak memory operations.*

This characterization suggests the existence of an algorithm for computing $D_{S\&S}$. We will consider the issues in obtaining an efficient algorithm for computing $D_{S\&S}$ in a Chapter 3.

2.3 Synchronization Analysis

In this section, we will see that the delay set computed in the previous section (using the framework developed by Shasha and Snir) does not take into account synchronization constructs and is therefore overly conservative.

It is correct to treat synchronization constructs as simply conflicting memory accesses, but this ignores a valuable source of information, since synchronization creates mutual exclusion or precedence constraints on accesses executed by different processors. If our algorithm is aware of the semantics of the synchronization operations, a smaller delay set could be computed. For example, Figure 2.8 shows two program segments that access the variables X and Y along with post-wait synchronization on F. A *wait* operation, as the name suggests, waits for a flag to be set by a *post* operation. A correct implementation of this synchronization construct would introduce a inter-processor synchronization. If this synchronization information is ignored, the delay set $D_{S\&S}$ would contain edges between both pairs of accesses to X and Y, thereby prohibiting any overlap. However, if synchronization is taken into account, the delays between the reads and writes are seen to be unnecessary, as long as the *post* will be delayed for the writes to complete and the reads cannot be started until the *wait* completes.

In this section, we modify the delay set computation to incorporate synchronization analysis for constructs that result in precedence constraints between accesses from different processors. Our synchronization analysis presumes that synchronization constructs have been matched across processors, and standard synchronization constructs such as post-waits and barriers have been processed to generate a set of precedence edges between accesses. We will describe this process in a later section. However, we observe that this analysis only helps if the programmer uses the synchronization primitives provided by the language. If the programmer builds synchronization using reads and writes (as in Figure 2.9), we would not be able to detect the synchronization. In this case our analysis is still correct since the synchronizing accesses would involve conflicts, but we would not be able to prune the delay set. We could try to recognize user-defined synchronization such as spinlocks and raise them to the same level as system defined primitives. However, since such an analysis is undecidable in the general case, we provide synchronization primitives as language constructs and analyze their usage to obtain precedence and mutual exclusion information.

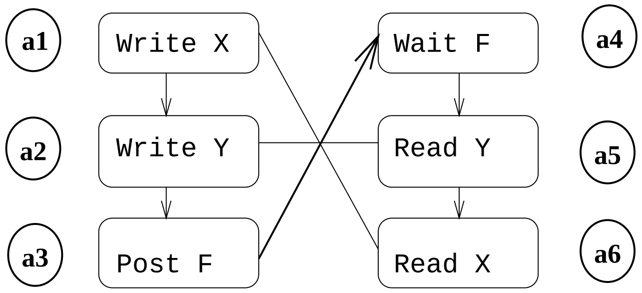


Figure 2.8: Example illustrating use of post-wait synchronization. Inter-processor conflict edges are represented by thin lines, while the post-wait precedence edge is represented by a thick line. The post-wait synchronization enables the ordering of conflict edges, which results in fewer back-paths.

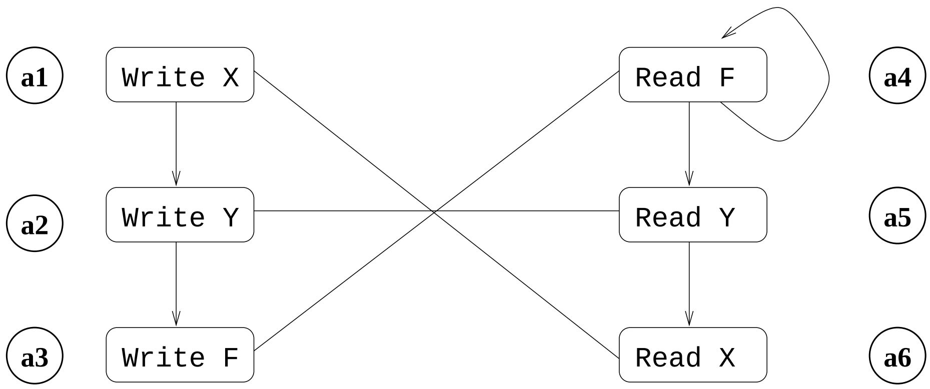


Figure 2.9: Synchronization operations as reads and writes.

2.3.1 Incorporating Precedence Information

Post-wait synchronization is used for producer-consumer dependencies. The consumer executes a *wait*, which blocks until the producer executes a *post*. This synchronization creates a strict precedence between the operations that are forced to execute before the *post* and the operations that are delayed to execute after the *wait*. In our discussion, we use a precedence relation R that captures the *happens-before* property of accesses. Matching post-wait pairs are one example of pairs in R.

Definition 6 *A* precedence relation R *is a set of ordered pairs of accesses,* $[a_1, a_2]$*, such that* a_1 *is guaranteed to complete before* a_2 *is initiated.*

System Contract 3 *Given a weak memory program with delay set* D *and precedence relation* R*, a correct machine must produce only those executions for which there exists a total order* S *of all the program operations such that* $\{D \cup R\}^* \subseteq S$*.*

We now consider a few examples that use post-wait synchronization, and then present the modified delay set construction.

Example 1: Consider the computation of the delay set for the program in Figure 2.8 where post and wait are treated as conflicting accesses. $D_{S\&S}$ is $\{[a_1, a_2]$, $[a_2, a_3]$, $[a_1, a_3]$, $[a_4, a_5]$, $[a_5, a_6]$, $[a_4, a_6]\}$, which will force completion of a_1 before the initiation of a_2 and a_5 before a_6. The semantics of post-wait synchronization require a precedence edge from a_3 to a_4, which eliminates one direction of the conflict edge between a_3 and a_4 and leads to a smaller delay set. It is sufficient to use the smaller delay set $\{[a_2, a_3], [a_1, a_3], [a_4, a_5], [a_4, a_6]\}$, because taken with the precedence ordering $[a_3, a_4]$, the other conflict edges $[a_1, a_6]$ and $[a_2, a_5]$ are ordered by transitivity, thus destroying the remaining back-paths.

As this example illustrates, we will compute the delay set and precedence relation through a process of refinement. Initially, the precedence relation contains only those edges that directly link a *post* and a *wait*. We then create an initial delay set D_{init} with those edges from $D_{S\&S}$ that involve at least one synchronization construct. This approach means that some delay edges—those involving precedence—are more fundamental than others. Once D_{init} is computed, the precedence relation R is expanded to include the transitive closure of itself and D_{init}. The example provides two key insights into how we could use synchronization information:

- By providing a directionality to a conflict edge, we impose more restrictions on the interleaving of accesses from different processors, which results in a smaller delay set.
- The precedence relation R serves as the catalyst for discovering other precedence constraints in the program. Note that a synchronization construct will not result in a fence for every outstanding memory operation, but only for those accesses that share a back-path with the synchronization operation.

Example 2: The process of pruning back-paths does not always involve directing a conflict edge. R could be used for removing certain accesses that are not qualified to appear in back-paths, and thus decrease the number of back-paths that we discover. In Figure 2.10, there are simple paths from a_3 to a_1, from a_6 to a_4, and from a_2 to a_1. Furthermore, since a_3 and a_4 are synchronization accesses, $[a_1, a_3]$ and $[a_4, a_6]$ belong to the initial delay set D_{init}. This information, when combined with the precedence edge $[a_3, a_4]$, implies that a_1 precedes

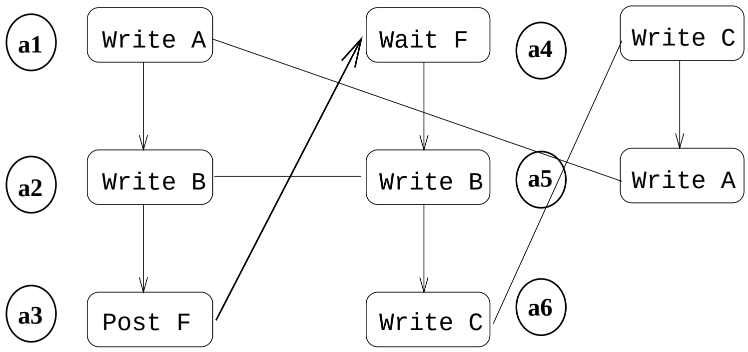


Figure 2.10: An example where synchronization analysis disqualifies certain accesses from appearing in back-paths.

a_6 for any execution of the program. Since a simple-path to a_1 corresponds to a runtime execution where all the accesses in the sequence execute before a_1, a_6 will never occur in a simple-path to a_1. We therefore remove a_6 while searching for simple-paths to a_1. Removal of a_6 destroys the simple-path from a_2 to a_1, which otherwise would have resulted in $[a_1, a_2]$ being added to the delay set.

Example 3: Another way by which back-paths can be invalidated is illustrated in Figure 2.11. The presence of post-wait synchronization around the accesses to the variable C provides an orientation to the conflict edge between the two accesses to C. This orientation ensures that the execution order of any back-path that traverses the conflict edge in the reverse direction would not happen in an actual execution. In the previous examples, a back-path between two accesses is nullified by ensuring the completion of the earlier access at a synchronization point that appears later in the same instruction sequence. In this example, the synchronization operations occur in other instruction sequences, but they still provide sufficient precedence constraints to prevent certain interleavings from occurring.

2.3.2 Computing a Minimal Delay Set

Using these examples as motivation, we propose a general scheme for finding a minimal delay set D_{synch} for correct execution of instruction sequences with synchronization operations. Recall that a back-path for a pair of accesses $[a_1, a_2]$ is a potential execution sequence that allows other threads to observe the reordering of accesses a_1 and a_2. Since certain interleaving patterns cannot occur in the presence of synchronization, we will now

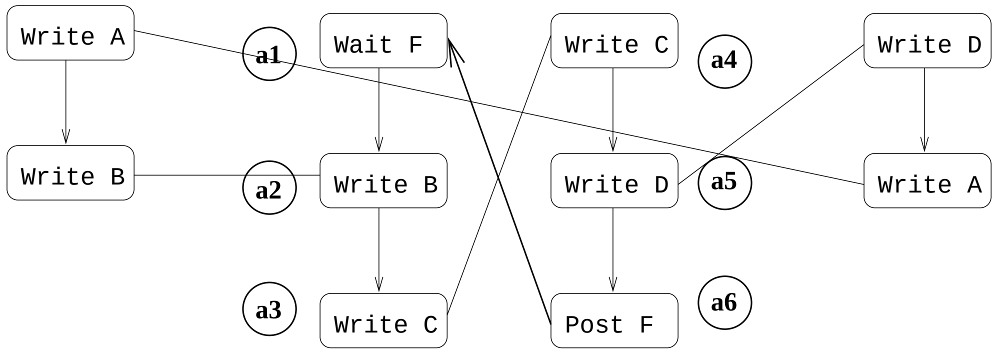


Figure 2.11: Synchronization constructs orient a conflict edge.

refine our notion of a back-path.

Definition 7 *Consider a set of instruction sequences P with delay set D and precedence relation R. Given an edge $[a_m, a_1]$ in some P_k, a path $[a_1, \ldots, a_m] \in P \cup C$ is called a* synchronized back-path *for $[a_m, a_1]$ with respect to a delay set D, if $[a_1, \ldots, a_m]$ is a simple path and $[a_j, a_i] \notin \{D \cup R\}^*$ for all $1 \leq i < j \leq m$. (The notation A^* denotes the transitive closure of a set A.)*

A back-path might not be a synchronized back-path if there are sufficient delay and inter-processor precedence restrictions to prevent the corresponding interleaving from occurring. Therefore, an iterative process of finding the delay set must pick the "right" order for identifying which P edges belong to the delay set. P edges tagged as delay edges earlier in the algorithm could make it unnecessary to include other P edges in the delay set. However, determining the optimal order could be difficult and possibly intractable. We take a different approach in computing the minimal delay set. Our approach is to identify an arbitrary delay set, which is sufficient to guarantee sequential consistency, and prune the set until the set cannot be further pruned and still produce only correct executions.

However, certain parallel programs might not have an unique minimum delay set. Figure 2.13 shows two different solutions for the delay set for the parallel program illustrated in Figure 2.12. The edges $[a_3, a_5]$, $[a_4, a_5]$, and $[a_6, a_7]$ need to be in the delay set. If $[a_8, a_9]$ is included in the delay set, it is not necessary to include $[a_1, a_2]$ in the delay set; there are no synchronized back-paths for $[a_1, a_2]$ as $[a_6, a_5]$ then belongs to $\{D \cup R\}^*$. The roles are reversed if $[a_1, a_2]$ is included in the delay set instead of $[a_8, a_9]$.

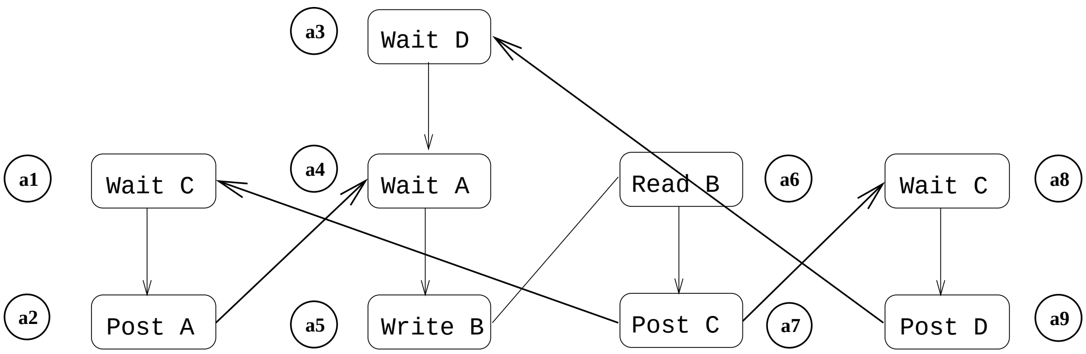


Figure 2.12: A parallel program with two different minimal delay sets.

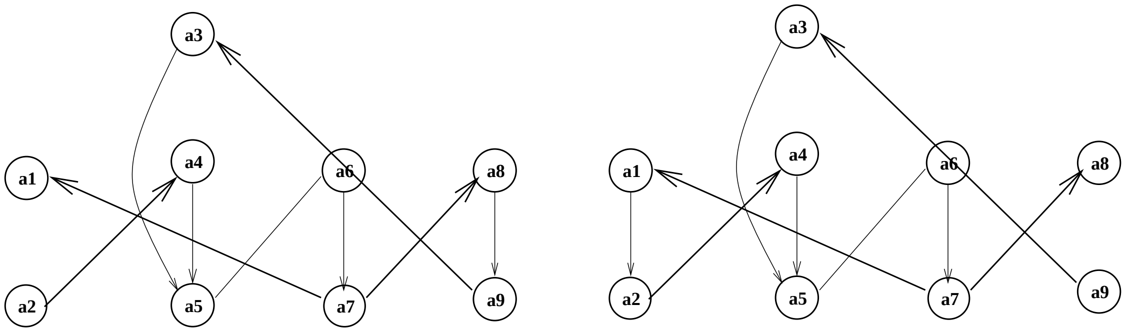


Figure 2.13: Two different minimal delay sets for the parallel program shown in the previous figure. The edge $[a_8, a_9]$ in the first delay set does not appear in the second delay set, while $[a_1, a_2]$ appears in the second but not the first.

Unfortunately, finding a delay set with the minimum cardinality could be computationally expensive. In fact, we can show that this problem is NP Hard (see appendix). Instead, we present an algorithm that computes an initial approximation to a minimal delay set, which can then later be pruned to remove redundant delay restrictions. This approach provides a minimal delay set and not the minimum delay set. We also observe that the smallest delay set does not always result in the best performance. Certain delay restrictions could be more expensive than others, and we will discuss in a later section the issues related to code generation based on a performance model of a given machine.

1. Compute initial delay restrictions D_{init} by restricting the cycle detection analysis (from the previous section) to pairs that include one synchronization access.

2. Compute the set of precedence edges, R_{init}, between synchronization operations.

3. 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) $[a_1, b_1] \in D_{init}$ and $[b_2, a_2] \in D_{init}$, and

   (b) $[b_1, b_2] \in R_{init}$

   Add $[a_1, a_2]$ to R.

4. The original conflict set C contained unordered pairs. Order the pairs that have a precedence as follows: Let $C_{orient} = C - \{[a_2, a_1] \mid [a_1, a_2] \in R\}$.

5. $D_{synch} = D_{init}$.

6. For every pair $[a_i, a_j] \in P$, let $X = \{\ b \mid [a_i, b] \in R\}$. If $[a_i, a_j]$ has a back-path in $P \cup C_{orient} - X$, then $D_{synch} = D_{synch} \cup \{[a_i, a_j]\}$.

This algorithm computes a delay set that is sufficient but not minimal. However, given a delay set that is not minimal, it can easily be pruned to obtain a minimal set by gradually removing delay restrictions while preserving the property that no synchronized back-paths are introduced for the P edges that are not in the delay set.

1. $D_{refine} = D_{synch}$, where D_{synch} is the delay set computed by the previous algorithm.

2. For every element $[a_1, a_2] \in D_{refine}$, if $[a_1, a_2]$ satisfies the following two properties, then $D_{refine} = D_{refine} - d$.

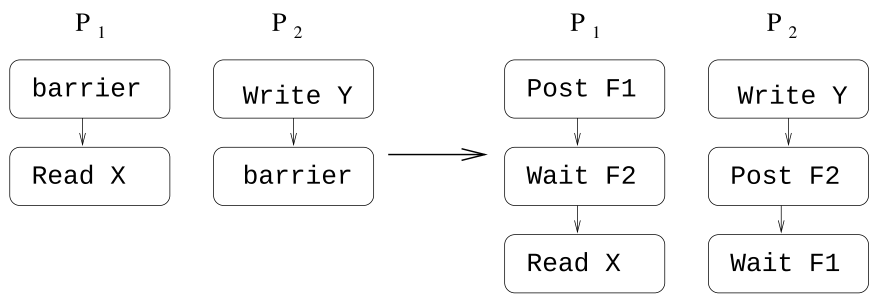


Figure 2.14: Barrier synchronization expressed using post-wait operations.

- There are no synchronized back-paths from a_2 to a_1 with respect to the delay set $D_{refine} - d$
- For every access pair $[b_1, b_2] \in P - D_{refine}$ there are no synchronized back-paths from b_2 to b_1 with respect to the delay set $D_{refine} - d$.

The purpose of having a two-step process for computing a minimal delay set is to develop a reasonable approximation to a minimal delay set before pruning the delay set. The pruning step is a costly operation as it checks for synchronized back-paths, which are more expensive to determine than a normal back-path. Also, if the approximation is sufficient for obtaining good performance, the second step could be skipped to improve speed of analysis.

2.3.3 Barrier Synchronization

Barrier statements can be used to separate the program into different phases that do not execute concurrently. The analysis for *barriers* is similar to that of post-wait synchronization, since crossing a barrier introduces a precedence relation. In fact, as Figure 2.14 illustrates, a barrier operation can be implemented using post-wait synchronization. As before, we add to D_{init} the delay edges between accesses and *barrier*s before we compute the delay set for the rest of the program.

2.3.4 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. We again compute D_{init} for pairs of accesses that include a synchronization construct. We then determine the set of accesses

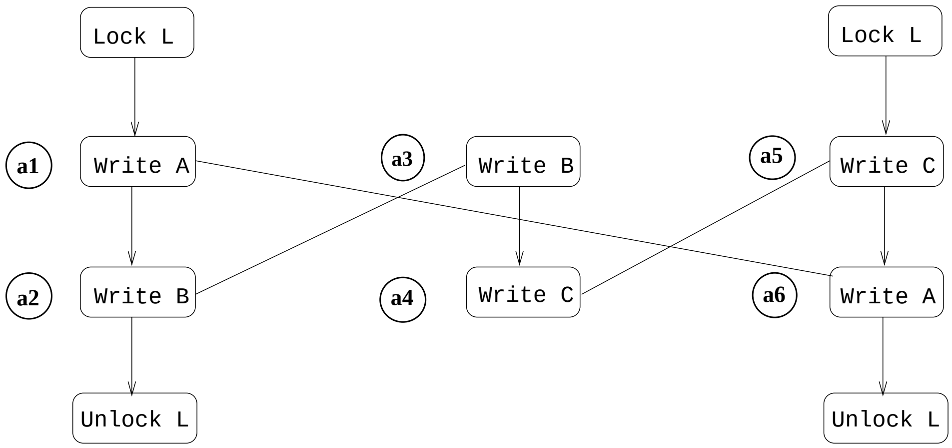


Figure 2.15: Eliminating accesses guarded by the same lock. While checking for back-paths between a_1 and a_2, accesses a_5 and a_6 are not considered. However, there does exist a valid back-path between accesses a_3 and a_4.

guarded by a lock. An access a is said to be *guarded* by the lock l, if the following conditions hold:

1. a is executed after a *lock* l operation (which we will call l_1), and there are no intervening *unlock* l operations.

2. a is followed by an *unlock* l operation, which we will call u_1.

3. $[l_1, a] \in D_{init}$ and $[a, u_1] \in D_{init}$

If accesses a_1 and a_2 are guarded by some lock l, we need not consider other accesses that are guarded by the same lock while checking for a simple-path from a_2 to a_1 (see Figure 2.15). It is valid to ignore such accesses by the following reasoning. If $a_2, b_1, b_2, ..., b_k, a_1$ is a simple-path, then the accesses corresponding to $b_1, b_2, ..., b_k$ must occur after a_2 and before a_1 in some legal execution of the program. It follows from our definition of being guarded by a lock and from the semantics of lock/unlock operations that none of $b_1, b_2, ..., b_k$ can be guarded by the same lock and still appear in a violation sequence. Since the set of accesses to be eliminated is defined using D_{init}, this step can be easily incorporated into the algorithm for computing D_{synch}. In particular, we can modify the following step (step 6):

Computing D_{synch}: For every pair $[a_i, a_j] \in P$, let $X = \{\ b \mid [a_i, b] \in R\}$. If $[a_i, a_j]$ has a back-path in $P \cup C_{orient} - X$, then $D_{synch} = D_{synch} \cup \{[a_i, a_j]\}$.

Modified D_{synch} Computation: For every pair $[a_i, a_j] \in P$, let $X = \{ \ b \mid [a_i, b] \in R\} \cup \{ \ b \mid a_i, a_j, b$ are guarded by lock $l\}$. If $[a_i, a_j]$ has a back-path in $P \cup C_{orient} - X$, then $D_{synch} = D_{synch} \cup \{[a_i, a_j]\}$.

With this discussion on mutual exclusion, we have completed developing a theoretical framework for analyzing the correctness of an execution and determining the delay restrictions that have to be imposed for correct executions. While most of this discussion dealt with runtime traces of operations from different processors, we need to recast our results into compiler analyses. We begin such a task in the next chapter.

Chapter 3

Compile-Time Analysis

We now examine the compiler techniques required to analyze the execution of parallel programs. There are many challenges in achieving this result. First, the theoretical framework developed in the previous sections dealt with a dynamic execution. The primary hurdle is to adapt these characterizations to work with a static description of a parallel program. We discuss the compile-time problem of conservatively approximating the possible runtime executions in Section 3.1. Second, our techniques need to be efficient in analyzing large programs running on a potentially large number of processors. However, the delay set characterizations presented in the previous chapter require computationally expensive algorithms. In particular, computing the delay set for MIMD programs is NP Hard (Section 3.2) requiring algorithms that are exponential in the number of program threads (Section 3.3). Fortunately, if we relax the minimality requirement for computing delay sets, we can analyze SPMD programs (in which the same code runs on each processor) with polynomial time algorithms that work even if the number of processors is not known at compile time (Section 3.4). The final challenge is that many of the correctness criteria described in the previous sections require substantial information about the program. In Sections 3.6 and 3.7, we will discuss techniques for gathering sufficient information to make our analyses viable.

3.1 Computing Delay Sets Statically

We now describe how to find back-paths from a static description of program behavior. Recall that our objective is to develop compile-time techniques for determining the delay restrictions to be imposed to obtain sequentially consistent executions. The compiler should detect at compile-time whether a cycle could occur in $P \cup C$, and refrain from introducing non-blocking operations if a cycle could indeed develop. 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 compiler introduces conflict edges, C, between access statements that could potentially access the same program variable at runtime. The compiler will detect back-paths, which are a string of access statements, to identify potential cycles in a runtime execution.

Our approach is to build a compile-time graph that captures the runtime graph of $P \cup C$ such that the compile-time graph would contain a cycle for every runtime cycle that might occur in $P \cup C$. Such a compile-time graph would have the following properties:

- Every runtime access is associated with an unique access statement in the compile-time graph.

- If an access a_2 might be performed after an access a_1 by a processor P_k at runtime, then their corresponding access statements s_1 and s_2 also are ordered in the compile-time graph. P is therefore approximated by the control flow graph. The control flow graph includes a representation of conditionals and loops of the original program, which may include paths that will not be taken at runtime.

- If two access statements s_1 and s_2 could potentially access the same variable in a conflicting manner, there is a conflict edge between the two statements in the compile-time graph. At compile-time, it might not be possible to accurately 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.

These properties would ensure that the compile-time graph is a strict "superset" of the runtime graph. Any runtime cycle can also be detected in this graph. The compiler can check for cycles using these approximations for P and C. If it detects a back-path for a pair of access statements, it can impose a delay restriction to guarantee that all accesses corresponding to the earlier access statement have completed execution before executing the later statement. (The details of imposing such a delay restriction are discussed in

Chapter 4.) The control flow graph itself might have cycles (due to loops and conditional jumps). Such a cycle in the control flow graph could result in back-paths, which involve access statements from a single processor. Since a reordering of accesses can be observed only by the interaction of accesses from two or more processors, such cycles are ignored while looking for potential violations.

These compile-time representations introduce certain complications. First, P now includes conditionals that affect what instruction sequences will be executed by the program. An incorrect evaluation of the conditional, which could occur due to incorrect placement of fences, might cause the wrong set of statements to be executed. We will use a simple argument to prove that such a scenario cannot occur. Note that for each conditional, the cycles we detect in the presence of the code contained in the branches is a superset of the cycles we would detect when such code is removed. When such code is removed, what is left is straight-line code with the variable accessed in the conditional being the last access issued. Since our cycle analysis guarantees a sequentially consistent value for this variable, we can be assured that for every branch taken during runtime there exists some sequentially consistent execution that would be consistent with the branching behavior. The second complication arises due to approximations in identifying conflict edges. By considering conflict edges between access statements, insteads of accesses, to variables that are potentially aliases, the compile-time approximation to C could be therefore very conservative. We have developed special techniques to discard certain cycles introduced due to this approximation.

3.2 Cycle Detection for MIMD Programs is NP Hard

Although Shasha and Snir do not specify the details of an algorithm for cycle detection, they claim there is a polynomial time algorithm for detecting cycles in a program that "consists of a fixed number of serial program segments. [58]" In practice, one does not typically compile a program for a fixed number of processors: either the language contains constructs for dynamically creating parallel threads, or there is a single program that will be compiled for an arbitrary number of processors (in the case of SPMD programs). We can use the approach proposed by Shasha and Snir on SPMD programs by making a copy of the program for each of $PROCS$ processors (assuming that $PROCS$ is known), but we show the computation needed for the Shasha and Snir formulation is NP-complete requiring algorithms with a running time that is exponential in the number of processors[1].

Theorem 3 *Given a directed graph G with n vertices, we can construct a parallel program P for n processors such that there exists a Hamiltonian path in G if and only if there exists a simple cycle in P.*

3.3 Exponential Time Algorithm for MIMD Programs

We present a simple exponential-time algorithm for detecting the back-paths in a MIMD program with the number of program threads fixed at compile time. This extends the approach of Shasha and Snir to programs containing branches and imperfect aliasing information as well as synchronization primitives. Given its limited applicability due to its exorbitant running time, we do not expect to use this algorithm in practice.

Recall that a back-path is a path that visits each instruction sequence at most once. To preserve this property[2], we impose an ordering on the threads and orient the conflict edges from lowered numbered to higher numbered threads. Let P_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 $P_1, P_2, ..., P_k$ 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 P_{p_j} to P_{p_i} where $i < j$. Remove control flow edges in P_s to eliminate cycles in the control flow graph. Also, remove all conflict edges from the thread P_s to the other threads except for those edges that involve the accesses u and v. This step ensures that a back-path does not visit an instruction sequence more than once.

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.

[1] The construction and the proof for this theorem is in the appendix.

[2] The algorithm for detecting back-paths is computationally expensive because of the need to preserve this property of visiting each thread at most once.

The intuition behind the algorithm is to guess an ordering of the threads in the violation sequence. 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 running time of the algorithm grows polynomially in terms of the number of access statements.

3.4 Cycle Detection for SPMD Programs

We now present a more efficient algorithm for computing the minimum delay set in an SPMD program. The algorithm is based on the idea of back-paths, but uses only two copies of the SPMD code, rather than one for each processor. The algorithm eliminates the condition that a back-path must pass through each program segment at most once. This changes the minimality property, but has little effect in practice as we will see in this section. We first describe a transformation of the given control flow graph and then present an algorithm for detecting back-paths in the resulting graph. We show that the delay edges computed for the transformed graph are the same as in the more expensive MIMD approach.

In an SPMD program graph $P = \{P_1, ..., P_n\}$, all P_i are identical. Let V be the set of access statements in each P_i. We define a conflict set C_{SPMD} as pairs of access statements that belong to V such that both statements access the same variable and at least one of them is a *write*. The conflict edges are bi-directional, so we write (u, v) for the pair of edges $[u, v]$ and $[v, u]$.

For a given program graph (P, V), we generate a graph P_{SPMD} with nodes V_{SPMD} and edges E_{SPMD}, defined as follows. V_{SPMD} is two copies of the accesses in V, which we label L and R for left and right.

$$
\begin{aligned}
V_{SPMD} &= \{v_L |\ v \in V\} \cup \{v_R |\ v \in V\} \\
T_1 &= \{(u_L, v_R) \mid (u, v) \in C_{SPMD}\} \cup \{(v_L, u_R) \mid (u, v) \in C_{SPMD}\} \\
T_2 &= \{(u_R, v_R) \mid (u, v) \in C_{SPMD}\} \\
T_3 &= \{(u_R, v_R) \mid [u, v] \in P\} \\
E_{SPMD} &= T_1 \cup T_2 \cup T_3
\end{aligned}
$$

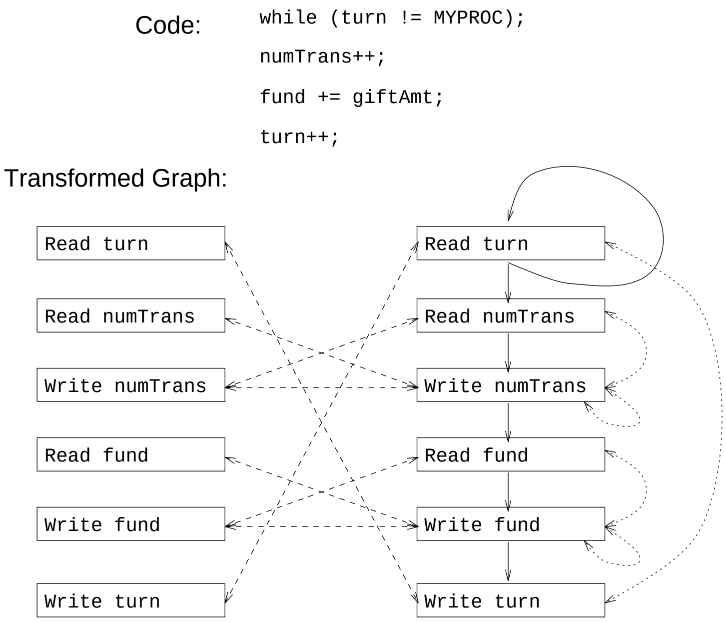


Figure 3.1: Cycle detection using two copies of the original program

This transformed graph has two copies of the original program. A backpath will have endpoints in the left part of P_{SPMD} and internal path nodes in the right part. The T_1 edges connect the left and right nodes. The T_2 edges are conflict edges between right nodes. The T_3 edges are program edges that link the right nodes. The left nodes have 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 terminated with a T_1 edge. Figure 3.1 illustrates the construction for a simple program.

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 P_{SPMD}. We compute the set D_{SPMD} that consists of all edges $[u, v]$ having a path from v_L to u_L. Our algorithm runs in polynomial time: if m is the number of statements in the program that initiate remote accesses, the delay set can be computed in $O(m^3)$ time.

3.5 Minimality of D_{SPMD}

Shasha and Snir prove that their algorithm (which is the MIMD algorithm for straight line code and perfect alias information) computes a delay set that is, in some sense, minimal. In this section we compare the delays sets computed by both algorithms for their minimality properties. There is a potential source of inaccuracy in the analyses that can lead to a non-minimal delay set as the detected backpaths may be longer than the total number of processors corresponding to a violation cycle that visits some instruction sequence more than once.

We define the *length* of a backpath as the number of conflict edges in the path and give the following results.

Theorem 4 *Given an SPMD program for which the longest backpath P_{SPMD} is less than or equal to $PROCS$, $D_{SPMD} = D_{S\&S}$.*
Proof: *Straightforward from the construction of P_{SPMD}.*

Our algorithm is correct regardless of the assumption on the longest backpath. To see why ours is more conservative in the (probably rare) case in which the program contains a long backpath, consider such a program. Our algorithm, as described, will compute a delay set for an arbitrary number of processors. If a program with a backpath of length n is run on $PROCS < n$ processors, the execution order identified by that backpath has insufficient number of processors to actually take place. Thus, the delay edge added for that backpath is unnecessary.

Even this difference between the algorithms is not fundamental; if the value of $PROCS$ is known at compile time, as Shasha and Snir assume, our backpath detection algorithm can search for backpaths shorter than $PROCS$, which will be exactly those produced using cycle detection on $PROCS$ copies of the program. The cycle detection analysis, as proposed by Shasha and Snir, produces a minimal delay set if the program segments comprise of straight-line code. Modulo the requirement of knowing the number of processors at compile-time, our algorithm observes the same minimality property.

There is one more substantial difference between the algorithms: we analyze SPMD code, while Shasha and Snir consider MIMD code. Given an MIMD program with $PROCS$ different programs, we can convert it into an SPMD program with a branch for processor number around each program segment. Our analysis will run in polynomial time on this transformed MIMD program while theirs will be exponential on the original program. Assuming one starts with a branch-free MIMD program, their analysis would be minimal but ours would not, because of the branches introduced in the conversion process. In practice, programmers rarely write programs with $PROCS$ different pieces of code, so the loss of minimality relative to practicality of the analysis is a good trade-off to make. In summary, the same theoretical results on minimality can be obtained for both algorithms, assuming the programs are branch-free and the number of processors is known at compile-time.

3.6 Overcoming Conservativeness in Conflict Edges

We will now discuss briefly a technique for discarding cycles introduced due to compile-time inaccuracies in computing conflict edges. Consider the following loop fragments:

```
Processor 1:                          Processor 2:

for (i=0; i<100; i++)                 for (j=0; j<100; j+=2)
    A[i] = 0;                             B[j] = 0;
```

If the compiler suspects that the two arrays could be aliased, conflict edges are introduced in a conservative manner resulting in delay restrictions for every access in the loop. However, in this example, a runtime cycle could never develop as both arrays **A** and **B** are accessed in a sequential fashion, so the delay restrictions are unnecessary. Since any alias analysis will be conservative for any reasonable program, it appears that our cycle analyses would work only if we have accurate alias detection algorithms. However, as most alias detection algorithms are complicated and are being actively researched, we employ a simpler method to overcome certain inaccuracies in conflict edges.

Our method is based on the following observation. A violation of sequential consistency corresponds to a cycle in $P \cup C$ where the end-points of the cycle correspond to accesses to the same variable. If we were to take the differences in the physical addresses of the variables accessed by adjacent accesses in this violation cycle and compute their sum, such a sum should be zero. Recall that as conflict edges access the same variable, it is sufficient to do such a calculation only on the P edges that appear in a cycle. We therefore annotate each P edge with the difference in the addresses referenced by its two accesses and consider only those cycles for whom the sum of these attributes is zero. In our example, we would annotate the P edges connecting adjacent iterations of the two loops with positive values $+1$ and $+2$ to denote the difference in indices of elements accessed in successive iterations. Since all the P edges are annotated with positive values, the compiler cannot identify any cycles whose edges have attribute that total to zero, and hence there are no

```
void foo() {

   barrier();
   for (i=0; i<n; i++) {
      ...
      barrier();
   }
   ...
   barrier();
}
```

Figure 3.2: Inaccuracies in analysis of barrier statements

violating cycles in $P \cup C$. This approach is sufficient for most programs that have regular access patterns.

3.7 Analysis of Synchronization Constructs

Another aspect of the compiler mapping problem is the issue of mapping the program's synchronization constructs to runtime events. The synchronization analysis described in the previous chapter assumes there are no ambiguities about synchronization objects; if we had imperfect information, the synchronization statements must be treated conservatively as write accesses. We discuss two specific instances of this problem. The first issue shows up in the analysis of barrier synchronizations, while the second issue deals with the safety aspects of synchronization variables.

To use barriers for computing precedence, we need to line up the barrier statements in the program text that will execute together at runtime. Since barriers can occur in branches and loops, the problem of lining up barriers is undecidable in general. Figure 3.2 shows a sample program where it is likely that all the processors will execute the final *barrier* statement at the same time, but to prove that assertion at compile-time, our analysis needs to prove that the function *foo* gets called by all the processors at the same time and the loop inside the function executes the same number of iterations on different processors. Rather than using sophisticated analysis to line up barriers [3][34], we use a simple runtime solution that works well for many real programs. We add a run-time check to each barrier to determine whether these are the ones lined up during compilation. The compiler produces two copies of the code, one with optimized non-blocking accesses and the other without such optimizations. If the processors are indeed synchronized and executing the barrier operations as predicted, the optimized version of the code is run. This approach to analyzing barriers also allows us to overcome separate compilation issues for many real programs. If, however, the language disallows such a generalized use of barrier synchronization, even the runtime checks and code duplication could be eliminated. Titanium, which is one of our source languages, enforces the property that all the processors arrive at the same barrier statement specified in the program. This property is enforced by the type system simplifying the compiler analysis.

The second set of issues deal with the proper use of synchronization variables used by *post-wait* synchronization and locking constructs. If a variable used for *post-wait* synchronization is accessed through standard read or write operations, the compiler might not be able to detect the precedence relationships in the program. Similarly, if the lock variables are used without a strict locking discipline, situations such as issuing unlock operations on locks the processor does not own might arise. Again, during compilation, any such misuse of locks would result in conservativeness in the analysis. We therefore would like language level restrictions to guarantee that synchronization variables cannot be indiscriminately modified by the program. The language could employ an extended type system to prevent access of synchronization variables through standard read/write operations in the program. A locking discipline could also be enforced through runtime checks to ensure that a lock cannot be unlocked by a processor that has not acquired the lock through a previous lock operation. A better solution is to have lexically scoped locking constructs in the language. Titanium, since it is a variant of Java, provides such constructs.

With this discussion on synchronization constructs, we conclude the description of our compiler analyses. In summary, we have shown that the task of finding the minimal delay set for MIMD programs could be NP-Hard. However, for SPMD programs, we have developed techniques for identifying the minimal delay set using polynomial-time algorithms. The MIMD algorithm might serve as the basis for future work in programs with dynamic thread creation, but the SPMD results are more practical and solve our specific concern of optimizing SPMD dialects of Java and C. As a further note on the generality of our results, we observe that many scientific applications written in languages that support dynamic thread creation are written in the more restrictive SPMD style by spawning a fixed number of threads at program startup [60]. Our SPMD analysis is also applicable to programs written in this style.

Chapter 4

Code Generation and Optimization

The notion of a delay set can be used to generate code for a variety of multiprocessor memory models. This includes the *put*, *get* model provided by Split-C and the Cray T3D and T3E [37, 57], weak CC-Numa models such as those provided on the Stanford Dash [42], the Kendall Square Research KSR1 [35, 59], and the Wisconsin Wind Tunnel, other modern machines that have hardware support for prefetching read and non-blocking write operations such as the MIT Alewife [2], J-Machine [48], and StarT [6], and systems with software support for shared memory with weak consistency protocols such as Treadmarks [4] and Munin [19].

In this chapter, we examine the issues in generating code for a put/get/store model using the Split-C definition of these primitives as the specific target. The mapping is independent of the source language, as it defines code generation for any compiler representation with a control flow graph augmented by a delay set. We have developed optimizing compilers for SPMD variants of both Java and C. Our compilers generate Split-C code, instead of a specific machine language, providing a high degree of portability. We use Split-C for concreteness and to separate some of the lower level code generation issues from the generation of optimized memory or communication operations. The Split-C compiler is responsible for mapping the Split-C operations to the primitive operations supported by the target machine, and the Split-C runtime system is responsible for implementing the memory operations on top of Active Messages, MPI, or other communication mechanisms. Before examining the transformations introduced by our backend, we present a brief overview of our source languages and the target language, Split-C, which is a parallel programming language in its own right.

4.1 Source Language: Titanium

The Titanium language is designed to support high-performance scientific applications. Titanium gives its users access to modern program structuring through the use of object-oriented technology, that enables its users to write explicitly parallel code to exploit their understanding of the computation. Titanium is a parallel SPMD dialect of Java. Titanium processes can transparently read and write data that resides on other processors. Titanium implementations run on both distributed-memory and shared-memory architectures.

Titanium is based on Java for several reasons. First, Java is a relatively small and clean object-oriented language (compared, for instance, to C++) and therefore is easy to extend. Second, Java is popular and is itself based on popular languages (C/C++). Learning Titanium requires little effort for those acquainted with Java-like languages. Also, Java is a safe language, and so is Titanium. Safety enables the user to write more robust programs and the compiler to perform better optimization.

The modifications to Java have been influenced by our primary goal of high performance. Performance is a fundamental requirement for computationally demanding scientific applications. Many of Titanium's features reflect this goal. For instance, the execution model is explicitly parallel SPMD, thus eliminating the need for a parallelizing compiler as well as avoiding the overhead of mapping a large number of threads to a smaller fixed number of processors. Any distributed data structure is fully defined by the programmer, who has complete control over its layout across process boundaries. In addition, the programmer may use type modifiers in variable declarations to convey locality information that the compiler may find hard to infer reliably from static analysis. Also, the Titanium compiler translates Titanium into Split-C or C to avoid incurring the costs of interpreted execution as well as enjoy the benefits of some of the traditional sequential optimizations provided by C compilers. Finally, Titanium has built-in features such as true multi-dimensional arrays and iterators, points and index sets (including irregular ones) as first-class values. We now discuss only those features of the language that are relevant to this research.

Local and global references. The storage associated with a Titanium process is called a *region*. Each object is contained within a single region. Local variables and objects created by **new** are contained in the region of the process that allocates them. References to objects in other regions may be obtained through communication primitives.

By default, all references in Titanium are assumed to be *global*, i.e. they may point to objects in any region. The programmer can declare that a variable v always points to objects of type T in the same region as the current process by declaring that variable with T **local** v. Similarly, the **local** qualifier can be used to declare that an object's field points to an object in the same region. On distributed-memory machines, local pointers are significantly more efficient than global pointers: they take less space, and access to objects is faster. On SMPs global and local pointers are equivalent.

Communication. Processes communicate with each other via reads and writes of shared object fields, by cloning whole objects, or copying parts of arrays. One-to-all communication is supported by the *broadcast* method with implicit global synchronization of all processes. There is also a *barrier* method. The compiler verifies that uses of these global synchronization operations maintain the structural correctness of the program. Process-to-process synchronization is handled as in Java with *synchronized* methods and the *synchronized* statement.

4.2 Source Language: SPMD-C

Like Titanium, this language also provides a global address space abstraction for SPMD programs. However, the global address space abstraction is provided in the source language only through a distributed array construct and through shared scalar values; the global address space cannot be accessed through global pointers, which are supported by Titanium. Disallowing global pointers allows us to implement our analysis without requiring full-blown pointer alias analysis. (Such alias analysis techniques are currently being implemented inside the Titanium compiler.) However, there are no restrictions imposed on the use of pointers into the local address space. The type system prevents the creation of local pointers into the global address space, and this allows us to ignore local memory accesses and local pointers in our analysis. Our source language design allows us to write meaningful parallel programs with optimized local computational kernels and global phases that communicate using either shared variables or distributed arrays.

4.3 Target Language: Split-C

Split-C is an explicitly parallel SPMD language for programming distributed memory machines using a global address space abstraction. Split-C is similar to Titanium in certain aspects. The parallel threads interact through reads and writes on a shared address space that contains distributed arrays and shared objects accessible through global pointers. However, unlike our source language, Split-C has a variety of split-phase (or non-blocking) memory operations, which are used by the compiler backend for enhancing performance. Our compiler can therefore be viewed as a transformer that converts programs with shared memory accesses into programs that contain split-phase memory accesses.

Given global pointers *src1* and *dest2*, and local pointers *src2* and *dest1* of the same type, the split-phase versions of read and write operations on the global objects are expressed as:

```
get(dest1, src1)
put(dest2, src2)
/* Unrelated computation */
sync();
```

In the first assignment statement, a *get* operation is performed on *src1*, and in the second, a *put* is performed on *dest2*. Neither of these operations are guaranteed to complete until the program execution passes the *sync* statement. The values of *dest1* and *dest2* are undefined prior to the *sync* and any modifications to *src1* or *src2* before the *sync* render *dest1* or *dest2* undefined even after the *sync*. A *get* operation initiates the read of a remote location, but it does not wait for the value to be fetched. Similarly, a *put* operation does not wait for the acknowledgement that the write occurred on the remote processor. The *sync* operation delays the execution for previous non-blocking accesses issued by the processor to complete. On a distributed memory machine, the *get* and *put* operations are implemented in hardware or software using low-level messages sent across the interconnection network. Therefore, split-phase operations facilitate communication overlap, which can be a significant performance advantage for programs with fine grained communication. The *sync* construct provides less control than one might want, because it groups all outstanding *put*s and *get*s from a single processor, stopping computation until all have completed. Split-C also provides finer grained mechanisms in which a *sync* object, implemented by a counter, is associated with each memory operation. Application level programming in Split-C is normally done using *sync*, but as a compiler target the *sync* objects are useful. Our earlier example could be modified to use these explicit counter objects to allow synchronization on only one of the outstanding remote accesses.

```
get_ctr(dest1, src1, ctr1);
put_ctr(dest2, src2, ctr2);
/* Unrelated computation */
sync_ctr(ctr1);
/* Computation unrelated to the put */
sync_ctr(ctr2);
```

Split-C also provides a *store* operation, which is a variant of the *put* operation. A *store(dest, src)* operation generates a write of value defined by *src* to a remote memory location pointed to by *dest*, 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. By transforming a *put* to a *store*, we not only reduce network contention by reducing the number of packets but also reduce the processing time spent in generating and handling the acknowledgement traffic.

4.4 Code Generation for Split-C Backend

In this section we describe the transformations introduced by our Split-C backend. We begin our discussion with a summary of the data structures used by the backend.

The primary data structure used by the backend is the control flow graph (CFG) of the program. The control flow graph organizes the program operations into a collection of *basic blocks* of instructions, each with uninterrupted control flow. The basic blocks are linked using predecessor and successor links to capture the control flow corresponding to program constructs such as conditionals and loops. In addition, *def-use* information of each program variable is obtained through standard dataflow analysis. Each definition of (assignment to) a program variable has a list of pointers to the nodes in the control flow graph where the variable may be used. Each node in the control flow graph also maintains a list of delay constraints imposed on its execution. These delay constraints correspond to the entries in the delay set computed using synchronization analysis.

```
x = X
y = 2
if (foo()) {
  y = x + 1
}
Z = 1
```

Source code

```
get_ctr(&x, &X, ctr1)
y = 2
if (foo()) {
   sync_ctr(ctr1);
   y = x + 1;
}
sync_ctr(ctr1);
z = 1
```

Target code

Figure 4.1: Code Generation: Upper case letters are shared variables, lower case letters are local variables, the solid line is a delay edge, and the dashed line is a def-use edge.

During the code generation process, both the delay constraints and local data dependencies must be observed. The generated code contains *put*, *get*, and *store* constructs, as well as various types of synchronization statements. The simple *sync* statement descibed earlier will force completion of all previous *put*s and *get*s issued by the processor. We also utilize Split-C's *synchronizing counters* to wait for the completion of a subset of outstanding accesses.

The first step in code generation is to split remote accesses into initiation and completion events. A remote *read* of X into y is transformed into *get_ctr(y, X, counter)* followed by *sync_ctr(counter)*, where `counter` is either a new or reused synchronizing counter. This transformation is always legal, but analysis is needed to move the two operations away from each other, thereby allowing communication overlap.

4.4.1 Separating Initiation from Completion.

The algorithm for moving a *sync_ctr* operation a_{sync} away from its corresponding initiation a_{init} involves repeated applications of the following rules. A similar strategy is used for issuing the initiations as early as possible to obtain the benefits of prefetching.

1. If a_{sync} is at the end of a basic block, propagate a_{sync} to all the successors of the basic block and continue the motion on each copy of a_{sync}.

2. If a_{sync} is in the middle of a basic block, let a' be the operation that immediately follows it.

   (a) If there is a delay or def-use constraint of the form $[a_{init}, a']$, terminate the movement of a_{sync}.

(b) If a' is another copy of the synchronization a_{sync}, delete the a_{sync} copy.

(c) Otherwise, move a_{sync} past a'.

The above algorithm moves *sync_ctr* operations as far away from initiation as possible. This algorithm automatically discards certain delay constraints if they are enforced by other restrictions. For example, if there are three different accesses (a, b, and c) that are constrained to execute in a serial manner, the delay constraint (a, c) is enforced by the restrictions on (a, b) and (b, c).

As shown in Figure 4.1, this algorithm can lead to multiple synchronization points in the same control path. The duplication may result in unnecessary overhead, but is legal because *sync_cntr* operations are idempotent. Similarly, if a *sync_ctr* is propagated into a loop body, it will be executed in every iteration, even though the first execution is sufficient. To avoid the cost of unnecessary sync operations, we could employ a loop-unrolling technique in which the first iteration of the loop is separated others 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 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 optimally choose between the two alternatives. Our compiler uses heuristics to avoid these overheads. To choose the best placement, the compiler would require data on average executions of control paths and machine parameters, which can be gathered using a combination of runtime techniques and profiling information.

4.4.2 One-Way Communication Optimization.

An optimization that is beneficial on many distributed memory machines is to transform two-way communication into one-way communication. Since the *sync* operations can be used to synchronize remote *put* operations, the *put* operation generates an acknowledgement that signals the completion of the write on the remote processor. If the source processor does not impose any delay constraints on the completion of a *put* access for correct

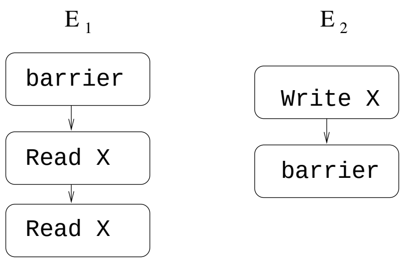


Figure 4.2: Barrier synchronization guarantees that X is read-only in the second phase.

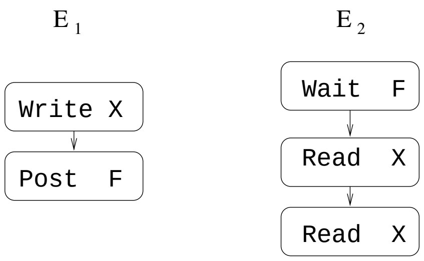


Figure 4.3: X can be cached by E_2 since the updates to X are guaranteed to be complete.

execution, the acknowledgement is not required, and therefore the *put* access can be replaced by the *store* operation[1] A more realistic scenario for optimization is when the completion of a *put* access is required at a barrier synchronization point. We could still transform the *put* into a *store* as long as the barrier synchronization waits for all the stores across the machine to complete. The individual acknowledgements are replaced by a single machine-wide check for the completion of all remote accesses during barrier synchronization. However, this requires a stronger form of barrier synchronization. Fortunately, on most tightly integrated machines like the CM5 and the T3D, this form of barrier synchronization is relatively inexpensive, and the benefit obtained by eliminating the acknowledgement traffic outweighs the additional cost.

4.5 Eliminating Remote Accesses

Delay sets are necessary for any transformation that involves code motion including the motion of memory access initiation away from its completion. In this section we consider a second class of transformations for distributed memory machines, which lead to

[1]Though the source processor does not require notification of the access's completion, the access cannot be completely eliminated as the value it produces might be required by other processors.

the elimination of remote accesses through a kind of common-subexpression elimination. The idea is to cache remote values that are being written or read multiple times in compiler generated local variables. This transformation happens after each remote access has been split into its corresponding initiation and completion events. Remote accesses to the same variable may be combined if certain kinds of code motion can be performed on their initiation and completion events. We begin by examining when these optimizations are valid within a basic block. We then describe the correctness criteria for optimizing across basic blocks.

4.5.1 Optimizations within a Basic Block

When there are repeated accesses within a single basic block to the same variable X, and X is not being written concurrently, the value of X can be cached. By virtue of synchronization analysis, it is easy to determine whether X is being concurrently written. Two examples are shown in Figures 4.2 and 4.3. In the first case, there is a *barrier* that marks the transition to X being read-only, and in the second, the *post-wait* synchronization ensures that the *get*s are issued only after the *put* is complete. The synchronization analysis described in the earlier chapter identifies these synchronization regions and orders the conflict edges between the *get*s and *put*s to X. Consequently, there is no delay edge between the two *get*s, and the second access can be eliminated.

Mutually exclusive access is sufficient but not necessary for elimination of repeated *get*s. It may be possible to reuse a previously read value even when there are intervening global accesses to the variable, as long as it is legal to move the second *get* up to the point of the first one. The algorithm used for remote access elimination is essentially the reverse of that used for *sync_ctr* propagation: the second *get* is moved backwards in the code until it reaches a operation that shares a delay edge or local dependence. If this propagation is successful, we will end up with a sequence like:

```
get(local1, X, counter1)
...
sync_cntr(counter1)
get(local2, X, counter2)
...
sync_cntr(counter2)
```

At this point, if there are no non-local accesses initiated between the two *get*s, the second *get* is replaced by a local assignment of `local1` to `local2`, and the second `sync_cntr` is

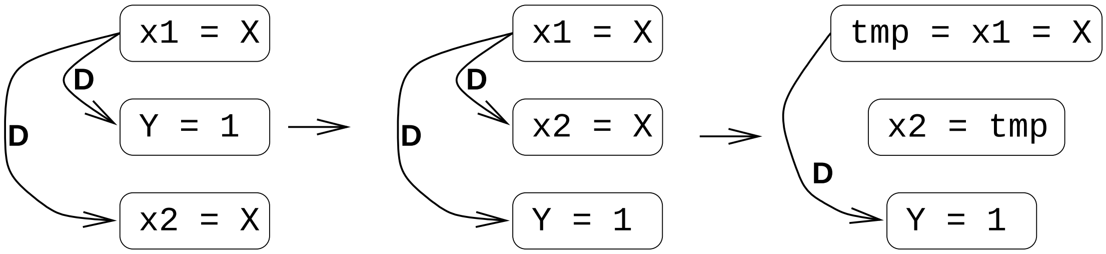


Figure 4.4: An intervening access could be moved out to make a caching transformation feasible.

eliminated, along with any of its copies. This transformation is possible even if there was an intervening non-local access provided the access could be executed either before the first read or after the second read (Figure 4.4). However, if the intervening access has delay constraints that prevent its relocation, a caching transformation would be invalid (Figure 4.5). We now present the steps involved in this transformation as a modification of the code motion algorithm for *sync* operations presented in the previous section. As before, we assume that an access a has been split into a_{init} and a_{sync}. We also assume that a' is the operation that immediately follows it inside the basic block.

1. If there is a delay or def-use constraint of the form $[a, a']$ and a and a' are accesses to different variables, terminate the movement of a_{sync}.

2. If there is a delay or def-use constraint of the form $[a, a']$ and a and a' are accesses to the same variable:

   (a) If a and a' correspond to different basic blocks, no optimizations are performed.

   (b) If there is a remote access b that is constrained to execute after a and before a', no optimizations are performed.

   (c) Otherwise, combine the two accesses by transforming the non-local accesss a' to an access to the value fetched by a.

3. Otherwise, move a_{sync} past a'.

The examples presented so far eliminate redundant reads, which is similar to saving a value in register. The technique can be applied to a variety of other communication-eliminating optimizations as illustrated in Figure 4.6. For simplicity, these optimizations are shown as transformations on the higher level code, with temporaries introduced to minimize

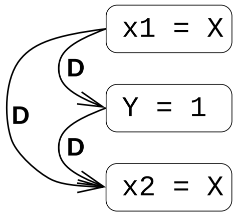


Figure 4.5: Caching is invalid when the two reads are separated by a remote access that is constrained to execute in between the two reads.

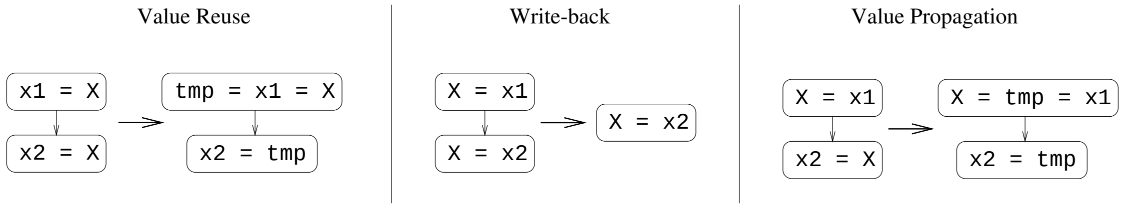


Figure 4.6: A set of transformations to eliminate remote accesses that are similar to standard uniprocessor optimizations like common subexpression elimination. Upper case letters denote global variables, and lower case ones local variables.

conflicts during code motion. Reading a remote variable that has recently been written can be avoided if the written value is still available. When a thread issues two successive writes to the same variable, the earlier writes can be buffered in a local variable and the final value written to the remote copy at a later point. This is equivalent to using write-back cache, rather than a write-through cache. Note that since there could be intervening reads to the same variable that have been transformed into reads from local memory through the value propagation transformation, the first write access cannot be eliminated entirely.

4.5.2 Optimizations across Basic Blocks

We cannot always apply these transformations to a pair of accesses that belong to different basic blocks. Here is a simple example that illustrates the dangers associated with applying the value reuse transformation across basic blocks.

```
x = X + 1;
...
while (X != 3);
```

Clearly, if we replace the remote reference to X in the spin-loop by a local reference to x, the program may stall. To address this problem, we classify variables into two types: data variables and control variables. Informally, the control variables are variables that

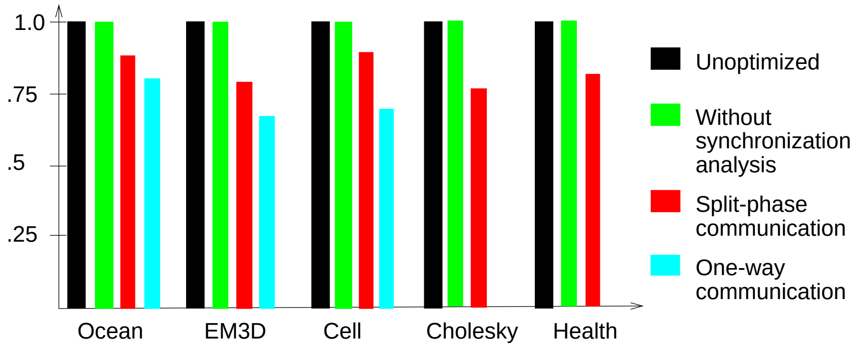


Figure 4.7: Execution times of our benchmarks normalized so that the execution time of the code generated without cycle analysis is 1. Thus, a relative speed of 0.5 corresponds to a factor of 2 speedup.

affect the control flow of the program while the data variables do not. The following definition describes the criteria used for the classification.

1. Let $CV = \{v \mid v$ appears in a conditional or loop header$\}$.

2. Repeat until there is no change to CV:
$$CV = CV \cup \{\ u \mid \exists\ v \in CV \text{ such that } u \text{ is used to compute } v\}.$$

The classification occurs after synchronization analysis, which partitions the program into different phases. Therefore, we might discover that certain variables that are used as control variables in certain phases of the program might be data variables in other phases. After classifying the variables, we apply the transformations that eliminate remote accesses only on the data variables.

4.6 Experimental Results

We quantify the benefits of our analysis by studying the effect of the optimizations on a set of application kernels that use a variety of synchronization mechanisms. These applications are written using SPMD-C. At this time, we do not have performance results for our Titanium compiler as the alias analysis techniques required to analyze Java references are currently being implemented[2]. A brief description of the applications is given below:

[2]The cycle analysis algorithm (along with analysis of synchronization constructs) have been implemented inside the Titanium compiler and produce correct transformations on simple code fragments.

Ocean: This benchmark is from the Splash benchmark suite[60]. **Ocean** studies the role of eddy and boundary currents in large-scale ocean movements. The primary data structures are grids that are updated using stencil-like computations. Distinct phases of the program synchronize using barriers.

EM3D: Em3d models the propagation of electromagnetic waves through objects in three dimensions [45][3] The computation consists of a series of "leapfrog" integration steps: on alternate half time steps, changes in the electric field are calculated as a linear function of the neighboring magnetic field values and *vice versa*. The alternate half time steps are separated using barrier synchronization.

Epithelial Cell Simulation: Biologists believe that the geometric structure of the embryo emerges from a few simple, local rules of cell movement. This application is a cell aggregation simulation that allows scientists to posit such rules. At each time-step of the simulation, a Navier-Stokes solver calculates the fluid flow over a large grid by performing 2-D FFTs. Barrier synchronization is used frequently in this application.

Cholesky: Cholesky computes the factors of a symmetric matrix. The primary data structure is a lower triangular matrix, which is distributed in a blocked-cyclic fashion. The computation is structured in a producer-consumer style. Synchronization is enforced at the granularity of blocks using post-wait operations on flags.

Health: This benchmark is from the Presto application suite. Health simulates the Colombian health service system, which has an hierarchical service-dispensing system. Exclusive access to shared data structures is guaranteed by the use of locks. Communication occurs along the implicit tree structure imposed by the hierarchical system.

Our prototype compiler automatically introduces the message pipelining and one-way communications optimizations for all the applications. The execution times of these applications were improved by 20-35% through message-pipelining and one-way communication optimizations. The measurements were done on a 32 processor Berkeley NOW, a cluster of 167 MHz UltraSparc workstations. Figure 4.7 gives the performance results of these experiments. The base program is the original unoptimized program. Our synchronization analysis results in much smaller delay sets, which in turn enables greater applicability of the message pipelining and one-way communication optimizations.

[3] We use a version of Em3d that uses arrays for storing electric and magnetic field values and is written in a style that is more akin to Fortran and HPF programming models. Hence, the program has performance characteristics that are different from those of the pointer-based versions described in [25].

Chapter 5

Global Memory Operations using Coprocessor Support

In previous chapters, we examined the compiler techniques required to implement a global address space with sequentially consistent semantics on top of weaker memory models, such as the one provided by Split-C. In the remainder of this thesis, we consider the issues involved in the implementation of the Split-C memory model on distributed memory machines. With an efficient implementation of Split-C, we would have completed our task of developing a global address space abstraction with strong semantics and high performance.

The fundamental system requirement for implementing Split-C primitives is the ability to move data around and signal the completion of the data transfers by updating synchronization counters. Split-C can be implemented on top of messaging layers such as Active Messages, Fast Messages, NX, and MPI, as well as machine specific mechanisms such as the Cray T3D and T3E's global memory operations. Using such mechanisms, Split-C runs on many parallel machines including the Thinking Machines CM5, the Intel Paragon, the IBM SP1 and SP2, the Cray T3D and T3E, the Meiko CS-2, and clusters of Intel or UltraSparc workstations.

In this chapter, we focus on machines with hardware support for communication in the form of a dedicated "network processor" (NP) to assist the compute processor (CP) during communication events. In particular, we consider the Intel Paragon and Meiko CS-2, distributed memory machines which have demonstrated practical scalability to hundreds of processors, and the Berkeley Network of Workstations (NOW), a cluster connected by a scalable switched local area network technology [65, 13, 24]. The Paragon provides a network processor that is symmetric with the compute processor, while the Meiko and NOW provide an asymmetric network processor that includes the network interface (NI). Against these hardware alternatives, we consider a variety of implementation techniques for global memory operations, ranging from general purpose active message handlers to specialized handlers that execute directly on the NP or in hardware. For the purpose of contrast, we also consider an implementation on the Thinking Machines CM-5, a machine without specialized hardware for communication. Our efforts reveal several crucial issues and trade-offs that must be addressed differently on different system designs, and also serve as an architectural evaluation. The primary purpose of this implementation exercise, however, is to identify the most efficient implementation strategy for each of our target platforms.

The rest of the chapter is organized as follows. Section 5.1 provides background information on our target architectures. Section 5.2 sketches the basic implementation strategies. Section 5.3 provides a qualitative analysis of the issues that arise in our implementations and their performance impacts. Section 5.4 substantiates this with a quantitative comparison using microbenchmarks. Finally, Section 5.5 draws conclusions.

5.1 Machines

In this section we detail the key architectural features of our candidate machines. We consider three machines, all constructed with commercial microprocessors and a scalable, low-latency interconnection network. The processor and network performance differs across the machines, but more importantly, the machines differ in the processor's interface to the network. Figure 5.1 gives a sketch of the node architecture on each machine.

Intel Paragon: In the Paragon [30], each node contains one or more compute processors (50 MHz i860 processors) and an identical CPU dedicated for use as a network processor. Our configuration has a single compute processor per node[1]. The compute and network processors share memory over a cache-coherent memory bus. The network processor, which runs only the OS kernel and is always in kernel (or system) mode, provides communication

[1]Some of the other installations of Paragon contain two compute processors per node.

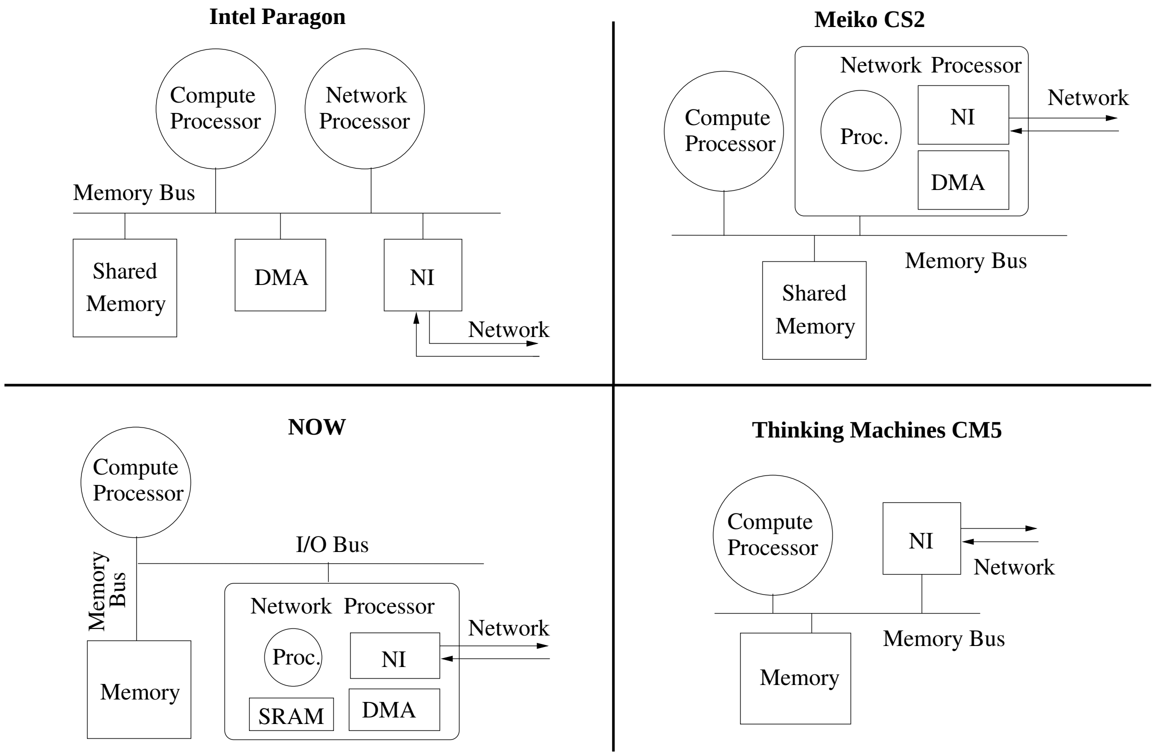


Figure 5.1: Structure of the multiprocessor nodes.

through shared memory to user level on the compute processor. It is also responsible for constructing and interpreting message tags. Also attached to the memory bus are 2 DMA engines and a network interface. The network interface provides a pair of relatively deep input and output FIFOs (2KB each), which can be driven by either processor or by the DMA engines. The network is a 2D mesh with links operating at 175 MB/s in each direction.

Meiko CS-2: The Meiko CS-2 [8] node contains a special-purpose "Elan" network processor integrated with the network interface and DMA controller. The network processor is attached to the memory bus and is cache-coherent with the compute processor, which is a 40 MHz three-way super-scalar SuperSparc processor. The network processor functions both as a processor and as a memory device, so the compute processor can issue commands to the network interface and get status back via a memory exchange instruction at user level. The network processor has a dedicated connection to the network; however, it has only modest processing power and no general purpose cache, so instructions and data are accessed from main memory. The network is comprised of two 4-ary fat-trees that have a link-level bandwidth of 70 MB/sec.

Berkeley NOW: The Berkeley-NOW [5] is a cluster of UltraSparc workstations connected together by Myrinet [13]. The CP is a 167 MHz four-way super-scalar UltraSparc processor. The Myrinet NI is an I/O card that plugs into the standard SBus. It contains a 32-bit CISC-based "LANai" network processor, DMA engines, and local memory (SRAM). The CP can access the NP's local memory through uncached loads/stores on memory mapped addresses. The NP can access the main memory through special DMA operations initiated through the I/O bus. The Myrinet network is composed of crossbar switches with eight bidirectional ports, and the switches can be linked to obtain arbitrary topologies. The network provides a link bandwidth of 80 MB/sec.

Thinking Machines CM-5: The CM-5 [41] has the most primitive messaging hardware of the four machines. Each node contains a single 33 MHz Sparc processor and a conventional MBus-based memory system. The network interface unit provides user-level access to the network. Each message has a tag identifying it as a system message, interrupting user message, or non-interrupting user message that can be polled from the NI. The compute processor sends messages by writing to output FIFOs in the NI using uncached stores; it polls for messages by checking network status registers. Thus, the network is effectively a distributed set of queues. The queues are quite shallow, holding only three 5-word messages. The network is a 4-ary fat tree that has a link bandwidth of 20 MB/sec in each direction.

5.2 Message Handler Placement Strategies

Global memory operations fundamentally involve three processing steps: (1) the processor issues a request, (2) the request is served on a remote node by accessing memory, possibly updating state or triggering an event, and (3) a response or completion indication is delivered to the requesting node. Between these steps, information flows from the CP through network processors (NPs), network interfaces (NIs), and memory systems. Consequently, a basic issue in any implementation is choosing which set of hardware elements are involved in the transfer and where the message is handled. In this section, we outline four possible strategies for implementing global memory operations.

Compute Processor as Message Handler (Proc): In our first approach, the message handlers are executed on the CP. In the simplest implementation of this strategy, the CP injects the message into the network (through the network interface), and the CP on the remote node receives the message, executes the handler, and responds (as shown in Figure 5.2 for a remote read operation).

The same basic strategy can be employed with network processors, using them simply as smart network interfaces (as shown in Figure 5.3). The communication between a CP and an NP on a node occurs through a queue built in shared memory. The CP writes to the queue, and the NP, which is constantly polling the queue, retrieves the message from memory and injects it into the network. The message is received by the NP on the remote side, enqueued into shared memory, and eventually handled by the remote CP, whereupon a handler executes and initiates a response. In this implementation, the network processor's task is to move data between shared memory and the network, guarantee that the network is constantly drained of messages, and perform flow control by tracking the number of outstanding messages.

Network Processor as Message Handler (NP): In our next approach, the message handlers are executed on the NPs thereby reducing the involvement of the CPs (see Fig-

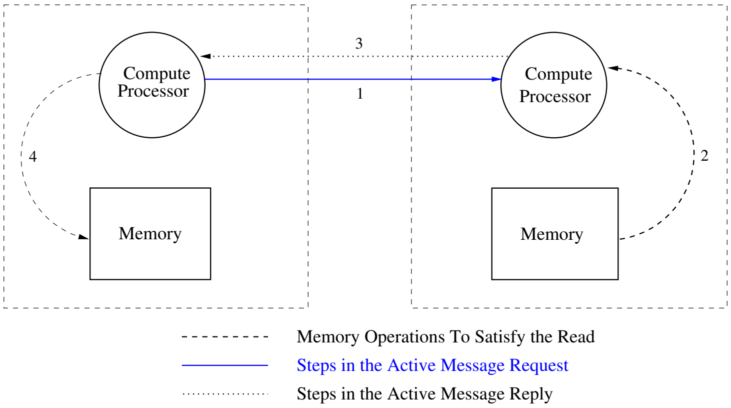


Figure 5.2: Handler execution on the CP.

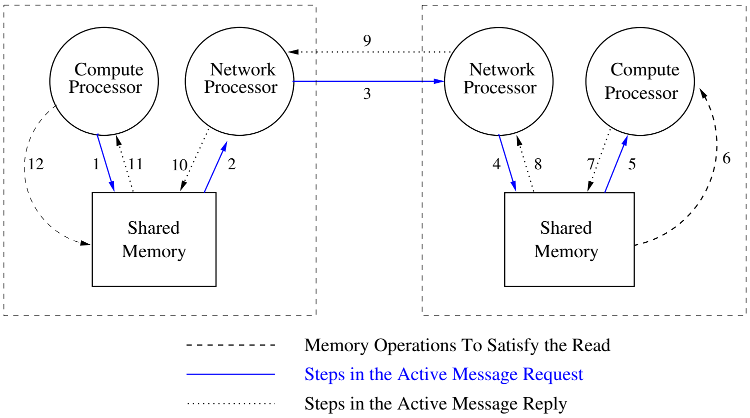


Figure 5.3: NPs treated simply as network interfaces.

ure 5.4). The request initiation is similar to the base implementation; the CP uses shared memory to communicate the message to the NP, which injects the message into the network. The NP on the remote node receives the message, executes the corresponding handler, and initiates a response without involving the CP. The NP on the requesting node receives the response and updates state to indicate the completion of the remote request without involving the CP. This strategy streamlines message handling by eliminating the involvement of the compute processors.

Message Injection by the Compute Processor (Inject): In this approach, the CP on the requesting node directly injects messages into the network without involving the NP (as shown in Figure 5.5). The remote NP receives and executes the message before initiating

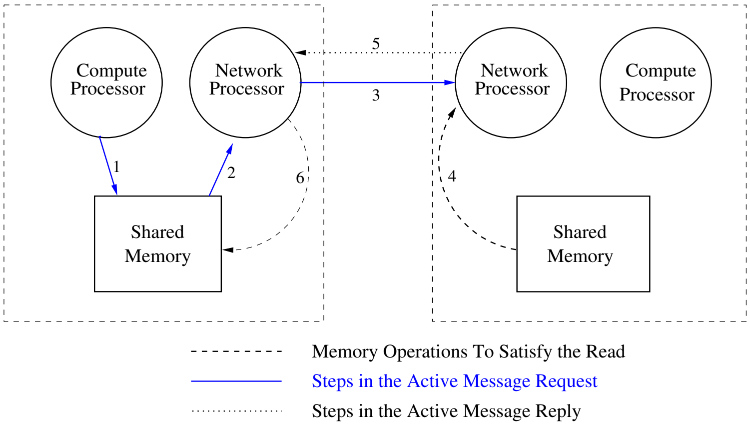


Figure 5.4: Handler execution on the NP.

the response, which is eventually received and handled by the NP on the requesting node. This approach streamlines message injection by eliminating the NP's involvement. However, in this strategy, since both the CP and the NP can inject messages into the network, the network interface must be protected to ensure mutually exclusive access.

Message Receipt by either Processor (Receive): Our next approach differs from the Inject strategy in one aspect; the compute processors are also allowed to receive and handle the messages. As with the Inject strategy, the CP directly injects requests into the network. However, on the remote node, the request is serviced by either the CP or the NP depending on which processor is available. Similarly, when the reply comes back, it is handled by either one of the two processors on the source node. In the Inject approach, during a global memory operation, there are three points at which an NP is involved in a message send/receive while the CP is involved only once. Since interfacing with the NI is typically expensive, this asymmetry in roles could potentially lead to a load imbalance during a global communication phase. The Receive strategy corrects this asymmetry by dynamically balancing the load between the CP and the NP.

Prototype Implementations We have implemented many versions of the Split-C language with each version highly optimized for the underlying hardware. On machines without network processors, such as the Thinking Machines CM-5, only the base strategy is possible; all message handlers are executed directly by the compute processors. On the Meiko and the NOW, we have an implementation that executes the message handlers on the CP

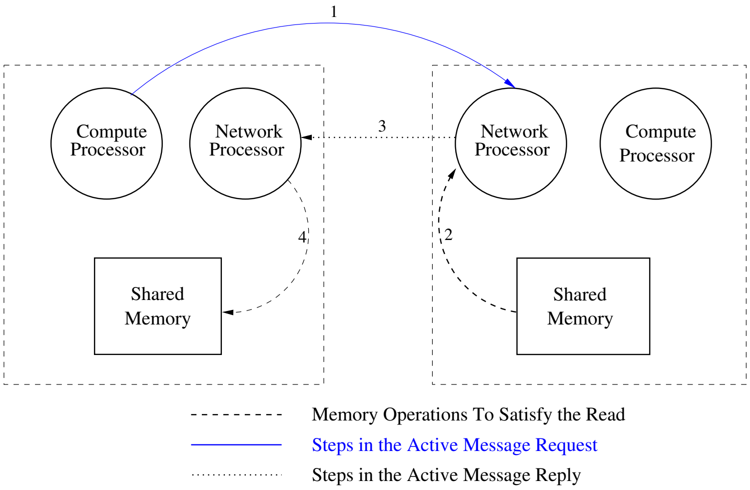


Figure 5.5: CP directly injecting messages into the network.

and another that executes them on the NP [55, 56]. The CP does not interact with the network directly in either case. On the Paragon, we have an implementation for each of the four different message handler placement strategies discussed in this section. These implementations have raised a host of issues involved in the use of network processors. We will address them qualitatively and quantitatively in the next sections.

5.3 Issues and Qualitative Analysis

In this section we present a set of interacting issues that shape the implementation of global address operations. Our goal is to identify the hardware and software features that affect both correctness and performance. This provides a framework for understanding the performance measurements in the next section.

There are three dimensions to communication performance. The *latency* of an operation is the total time taken to complete the operation. If the program must wait until the operation completes, this is the most important figure of merit. However, if the program has other work to do, then the important measure is *overhead*, which is the processing time spent issuing and completing a communication event. If there are many concurrent communication events occurring at the same time, then the time for each event to get through the bottleneck in the communication system, called the occupancy or *gap*, may matter most, since it determines the effective communication bandwidth. Our perspective is influenced by the LogP model [23], although we use a different definition of latency, which includes overhead.

We divide the set of implementation issues into three categories: those that are determined primarily by the machine architecture, those that are determined by the particular language implementation, and those that are only determined by application usage characteristics.

5.3.1 Architectural issues

We begin with the family of architectural issues in a bottom-up fashion, starting with the basic factors involved in moving bits around. We then consider the processors themselves, and work upwards toward system issues of protection and address translation.

Interface to the network processor and the network interface

The interface between a processor and the network is a notorious source of software and hardware costs. We consider two design goals in this section, minimizing overhead and minimizing latency, which suggest opposing designs.

To minimize latency, we should streamline the communication process by reducing the number of memory transfers between processors. One expects the latency to be minimized if requests are issued directly from the CP into the network, the remote operation is handled and serviced directly out of the network, and the response is given directly to the requesting processor. In fact, the latency advantage depends strongly on the specifics of the interface between the processor and network processor and the interface to the network. Moreover, this design optimizes for latency at the cost of higher overhead. To minimize the communication overhead for the CP, the solution is to offload as much work as possible to a separate NP. The offloaded work of transferring data into (or out of) the network involves checking various status indicators, writing (or reading) FIFOs, and checking that the transfer was successful. On all three machines this transfer is expensive. Thus, we would expect the reduction in overhead to be significant as the cost of transfer to the network is traded off against communication with the NP.

Surprisingly, on the Paragon, the cache coherency protocol (between the CP and the NP) results in at least four bus transfers for an eight-word processor-to-NP transfer, at a cost greater than a processor-to-NI transfer [44]. On the Meiko, because the NI is integrated into the NP, the CP can not directly send into the network. Communication between the processor and NP involves a transfer request being "written" to the NP with a single exchange instruction, whereupon the NP pulls the message data directly out of the processor's cache and writes it into the network. The NOW configuration is similar to that of Meiko's with the NP having a dedicated connection to the NI. The difference is that the CP and the NP do not share memory. Instead, the CP and the NP communicate through a queue located in the NP's local memory, which is accessible to the CP through memory mapped addresses.

A correctness issue also arises if the processor injects requests directly into the network and the NP is to handle requests and inject responses; the network interface must be protected to ensure mutually exclusive access. This issue does not arise on the Meiko or NOW, since they do not allow direct access to the NI from the processor. In our Paragon implementation, explicit locks are used to guarantee exclusive access.

Relative performance and capability

One of the stark differences in our target platforms is the processing power and capabilities of the NPs. One expects performance to improve if handlers are executed on the NP. However, if the NP is much slower than the processor, as on the Meiko, then it may be best for the NP to do only simple operations; it may be faster to have the NP pass complex requests to the processor than to execute the request itself. The optimal choice depends not only on the relative speed of the two processors, but also their relative load; if the processor is busy, then the best performance may be achieved by executing the handler on the slower NP. Of course, the request must always be passed on to the remote processor if it demands capabilities not present in the NP. For example, the Meiko NP does not directly support floating-point or byte operations, while the NOW NP has no floating point support.

Protection

Protection issues arise at each step of a global access operation. The network interface must be protected from incorrect use by applications. Messages sent by one application must arrive only at target remote processes for which it is authorized. Hosts must continue to extract messages to avoid network deadlock. Finally, handlers must only access storage resources that are part of the application. The traditional solution in LANs and early message passing machines was to involve the operating system on sending and receiving every message (Ncube/2 [63]). This requirement can be eliminated with more complete architectural support, described below, and with coarser system measures, such as gang scheduling of applications on partitions of the machine.

The Paragon has primitive hardware support for protection. It does not distinguish user and systems messages or messages from different processes, and there is no safe user-level access to the NI. Instead of invoking the OS for each communication operation, protection is enforced by passing all messages through a shared buffer to the NP, which runs at system level and provides protection checks, resource arbitration, and continually drains the network. In all but the base implementation, remote operations are serviced directly on the NP, which performs protection checks on access to user addresses. In the implementations in which the CP directly injects requests into the NI, there is a protection loophole; these experiments reflect the performance that would be possible if the Paragon were to adopt measures similar to the Meiko. The NOW implementations of Split-C gang-schedules parallel jobs. It does not support time-slicing currently, so a parallel program's traffic is insulated from others. The control program on the NP is protected from user tampering, and it performs protection checks on user supplied addresses.

The Meiko protection mechanism is more sophisticated. A collection of communicating processes possess a common communication capability. All messages are tagged with the capability, which is used to identify the communication context on the NP. This context includes the set of remote nodes to which the application is authorized to send messages and the virtual memory segment of the local process that is accessible to remote processes. Time-slicing the NP processor allows applications to make forward progress, so arbitrary user handlers may be run directly on the NP.

Address translation

A key issue in all of our implementations is how the address of a globally accessible location is translated. A global pointer is statically distinct from a local pointer, so the address translation is potentially a joint effort by the compiler-generated code, the remote handler, and the hardware. If remote operations are served by the CP, the virtual to physical translation is performed by the standard virtual memory mechanism, and page-fault handling is decoupled from communication. If remote operations are served by the

NP, many alternatives arise.

On the Meiko the user thread on the NP runs in the virtual address space of the user process. (The NP contains its own TLB, and its page table is kept consistent with that used by the processor.) Thus, the address translation on request issue is the same as for the Proc strategy.

On the Paragon, if the message handler is run on the NP, it runs in kernel mode. In the current version of OSF/AD, it executes directly in the physical address space. Thus, all accesses to user address space are translated and checked in software on the NP. In principle a TLB could be used to accelerate this translation, but since the nodes are potentially time-sliced, the NP would still need to check that it contained a mapping for the target process of the message and adjust its mapping or emulate the context as appropriate. The remote page-fault issue arises as well, but the message is explicitly aborted by the handler.

On the NOW, since the NP is attached to the I/O bus, it can access main memory only through valid I/O space addresses. However, since the I/O bus supports only 28-bit addresses, only a portion of the user's address space can be mapped into the I/O space at any given time. Consequently, if an access is made to an address that is not currently part of the I/O space, the NP passes the request to the compute processor.

DMA support

All of the issues above come together in DMA support for bulk transfer operations. On the Paragon and Meiko, DMA offers much greater transfer bandwidth than small messages. The Paragon operates on physical addresses and has a complex set of restrictions for correct operation, so it must be managed at kernel level. To improve overall network utilization, the NP fragments large transfers into page sized chunks. The Meiko provides more sophisticated DMA support by allowing the user to specify arbitrary sized transfers on virtual addresses. The DMA engine is part of the NP and automatically performs address translation and fragments the transfer into 512 byte chunks, which are interleaved with other traffic. On the NOW, a bulk transfer requires the participation of two DMA engines; the host DMA moves data between main memory and the NP's local memory, while the NI DMA is responsible for moving data between the NP's local memory and the network. It operates in the I/O address space and requires an additional copy to (or from) the user space.

5.3.2 Language implementation issues

We now turn to the family of issues at the language implementation level, given that the network processor can execute handlers and has specific architectural characteristics that can be fully exploited.

Generality of handlers

If the network processor runs in kernel mode, as on the Paragon, it can only run a fixed set of "safe" handlers.[2] The protection and address translation capabilities of the Meiko NP make general handlers possible, but its poor performance makes it usable only for highly specialized handlers. Atomic handlers are challenging to implement efficiently on the NP. Expensive locking may be required to ensure exclusive access to program states by the CP and NP. Our approach is to always execute the complex atomic operations on the compute processor to avoid costly locking.

Synchronization

Non-blocking operations, such as get, put, and store require some form of synchronization event to signal completion. This is easily implemented with counters, but if operations are issued by the processor and handled by the NP, then the counters must be maintained properly with minimal cost for exclusive access. An efficient solution is to split the counter into two counters, using one counter for the compute processor increments and the other for the network processor decrements. No race condition can occur since each processor can only write to one location, and can read both. The sum of the two counters produces the desired counter value. Implementing this on the Meiko and the Paragon is best accomplished by having each half of the counter in a separate cache line to avoid false sharing.

Optimizing for specialized handlers

If some of the handlers are to be executed on the NP, they can be optimized for their task and the specific capabilities of the NP. On the Paragon, the translations of frequently accessed variables, e.g., the completion counters, can be cached for future use. Message formats are specialized for the NP handlers to minimize packet size. One-way operations,

[2] This set might be enlarged by using sandboxing or software fault isolation techniques [10, 66].

such as stores, can be treated specially to reduce the number of reverse acknowledgments needed for flow control. The Meiko allows for optimizations of a different sort, as the handler code can be mapped directly onto some specialized operations supported by the NP, such as remote atomic writes [56].

5.3.3 Application issues

Considering architectural and language implementation issues in isolation, one can construct a solution that attempts to minimize the latency, overhead, and gap for the individual global access operations, striking some balance between the three metrics. However, the effective performance of these operations depends on how they are actually used in programs. Two issues that have emerged clearly in this study are responsiveness and the frequency of remote events.

Responsiveness

The prompt handling of incoming messages is important for minimizing latency. One way to ensure messages are handled as they arrive is for the message to trigger an interrupt. Unfortunately, few commodity processors have fast interrupts, so where possible we utilize polling of the network interface. If the CP is responsible for polling and fails to do so because it is busy in compute intensive operations, the effective latency can increase dramatically. If remote operations are handled on the NP, it can be responsive to these requests, regardless of the activities of the processor.

Frequency of remote events

Remote events are operations where control information is transmitted to a remote process along with data. The simplest remote event we consider is the signaling store, which informs the remote processor how much data has been stored into it by incrementing a counter. Synchronization operations, such as fetch&add and more general atomic procedures involve more extensive operations within the remote address space. Many event driven applications use a distributed task queue model, where communication causes a new event to be posted on a queue.

If a program invokes very few remote operations, architectural support for communication has very little impact on performance. If a program is communication intensive and if all remote operations are variants of read and write, which involve only the remote memory and do not interact with the remote processor, then, unsurprisingly, devoting hardware to serve these operations will improve performance. However, specific support for simple read and write operations does little to support remote events, such as stores, since they are relatively expensive to implement as multiple operations on the remote memory space. Thus, the latency, overhead, and gap of these remote event operations varies significantly across our platforms, but the effective impact on performance depends on how frequently they are used in applications, which also varies dramatically.

5.3.4 Summary

Each implementation strategy on each platform must address the issues raised in this section, it represents a particular point of balance in the opposing trade-offs present. In this section, we have examined in qualitative terms how the available architectural support, the language implementation techniques, and the program usage characteristics influence the performance of global access operations. Given this framework, let us examine the performance obtained on each of the operations in isolation and the resulting application performance.

5.4 Performance Analysis

In this section we present detailed measurements of our ten implementations to provide a quantitative assessment of the issues presented in the previous section. We divide our discussion into three parts. First, we examine the raw communication performance of active messages and Split-C primitives. We also study bulk synchronous communication patterns, where multiple processors simultaneously exchange messages. Next, we examine bulk transfers and the achieved bandwidth. Finally, we study a microbenchmark that illustrates the impact of attentiveness to the network.

5.4.1 Performance of Split-C primitives

Table 5.1 shows the round-trip latency, gap, and overhead for active messages, as well as get, put, and store operations under the various implementations. The same information is also presented graphically in Figure 5.6. The upper three groups test a single

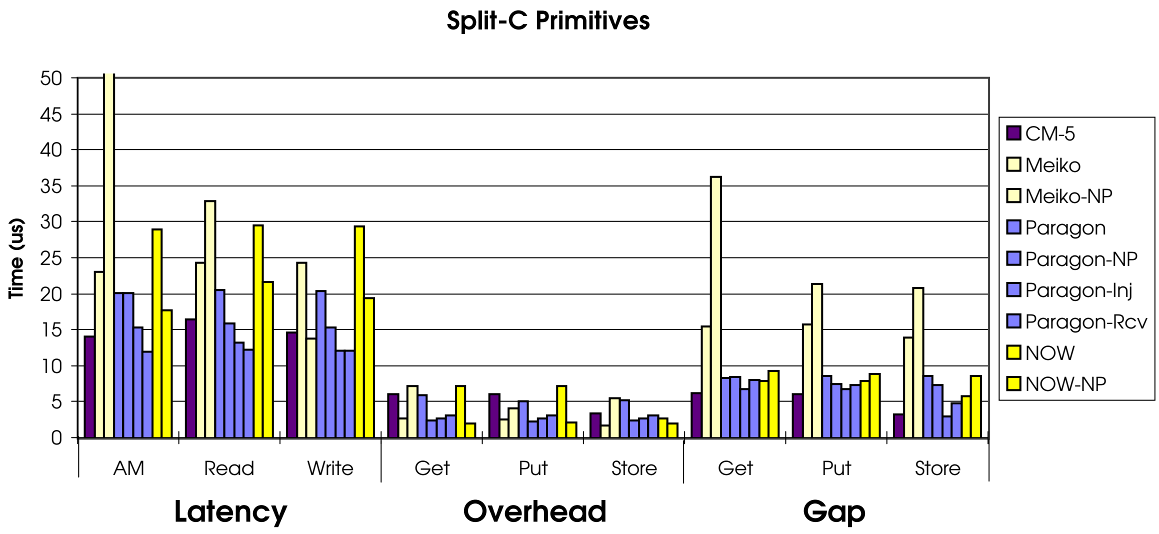


Figure 5.6: Cost of basic Split-C operations, for different versions of Split-C (times in us).

		CM-5	Meiko		Paragon				NOW	
		(Proc)	Proc	NP	Proc	NP	Inject	Recv	Proc	NP
RT Latency	AM	14.0	23	52	20.1	20.1	15.3	12.0	28.9	17.7
	Read	16.5	24.3	32.9	20.5	15.9	13.2	12.2	29.5	21.6
	Write	14.6	24.3	13.8	20.4	15.3	12.1	12.1	29.3	19.4
Overhead	Get	6.0	2.6	7.1	5.9	2.4	2.7	3.1	7.1	2.0
	Put	6.1	2.5	4.1	5.0	2.3	2.6	3.1	7.1	2.1
	Store	3.4	1.7	5.4	5.2	2.4	2.7	3.1	2.6	2.0
Gap	Get	6.2	15.5	36.1	8.3	8.4	6.7	8.0	7.9	9.2
	Put	6.0	15.8	21.3	8.5	7.5	6.7	7.3	7.9	8.8
	Store	3.3	13.9	20.8	8.5	7.3	2.9	4.8	5.8	8.6
Gap (to 2)	Get	6.1	15.7	24.7	9.0	8.4	6.7	7.4	15.7	18.4
	Put	5.9	15.6	20.1	8.5	7.6	5.7	7.3	15.7	17.7
	Store	3.4	14.0	19.8	8.2	7.5	2.9	3.2	11.5	11.7
Gap (exch)	Get	9.3	28.4	47.2	16.0	13.5	13.2	11.9	15.7	15.2
	Put	9.3	27.8	23.8	15.8	12.7	12.4	11.3	15.7	13.9
	Store	7.3	15.4	24.2	15.6	14.1	8.0	7.5	8.9	10.3

Table 5.1: Basic Split-C operations, for different versions of Split-C (times in us). Proc indicates the compute processor implementation, NP the network processor implementation, Inject the implementation where the compute processor directly injects messages, and Receive where it also directly receives responses.

requester and single remote server. The lower two involve communication among multiple processors.

Round-trip Latency: To evaluate system impact on latency, we consider three types of operations, all of which wait for an acknowledgement of completion. The first is a general active message, measured for the case of a null handler, and the others are blocking memory operations, read and write. The latency measurements exposes two of the issues raised earlier: total round trip time is minimized by avoiding the use of NPs during injection (thereby reducing memory transfers), and the use of specialized hardware to support particular operations significantly improves their performance.

Both the CM-5 and the Paragon show that faster communication is possible when messages are directly injected into the network. We see that the CM-5 latency is small compared to other architectures, since the CP directly injects and retrieves messages from the network. The effect of reducing memory transfers between the CP and the NP is most clearly seen on the Paragon. On the Paragon, each implementation improves on the previous versions. The advantage of Paragon-NP over Paragon-Proc is especially remarkable given the additional overhead of software address translation in Paragon-NP. The further improvement of Paragon-Inject implies that the benefit of directly injecting into the network outweighs the additional cost of locks, which are required for mutually exclusive access to the NI. On the NOW, the NP strategy eliminates the message transfer between the NP and the CP; however, the NP can access the CP's memory only through expensive DMA operations issued over the I/O bus. The lower latencies for the NP implementation imply that eliminating the message transfer more than compensates for the cost of the DMA operation.

The Meiko-NP implementation has higher latencies for active messages and reads, since the NP is much slower than the CP. However, the write operation on Meiko-NP is much faster than the read, since the Meiko has hardware support for remote write. In contrast, the four Proc implementations of read and write (on the CM-5, Meiko, Paragon, and NOW) are built using active messages, so they take the time of a null active message plus the additional time needed to read or write.

The round-trip measurements on the Paragon also bring out protection and address translation issues. The Paragon Proc and Paragon NP implementations show no difference in latency for active messages, because user supplied handlers cannot be executed on the NP due to inadequate protection. For the Paragon-NP and Inject versions, reads are slower than writes because a read reply requires an extra address translation step for storing the value read into a local variable. (In the Receive implementation, the difference between the read and write costs disappears since the replies are handled on the CP.)

Overhead: One expects that the overhead when writing directly to the NI will be greater than if an NP is involved; surprisingly, this is not always the case. Contrary to expectation, the overhead costs for the NP and the Inject versions on the Paragon are similar, which implies that it is as efficient to write to the NI as writing to shared memory on this platform. As expected, the Paragon and the NOW implementations that use the NP for message handling have much less overhead, since the CP does not handle the reply. On the other hand, the Meiko-NP overhead is higher than the Meiko-Proc, because of an implementation detail. To avoid constant polling on the NP, the more expensive event mechanism is used instead of shared memory flags for handing off the message to the NP. On the CM-5, we observe that the sending and receiving overhead accounts for almost all of the round-trip latency, unlike the other implementations that involve a co-processor.

Comparing the store overhead to the other results helps reveal the underlying architecture. As expected, the CM-5 overhead for gets and puts is almost twice the overhead for stores, which do not require a reply.

Gap: The results for the gap expose the bottlenecks in the various systems. For the CM-5, the gap is the same as the overhead, which indicates that the sending processor is the bottleneck. The higher gap for Meiko-NP and the NOW-NP implementations show us that the slower NP is the bottleneck. On the Paragon, the get operation has a higher gap than puts due to an extra software address translation made while handling the reply. This behavior means that the NP on the sending side is the bottleneck. In the Inject implementation, the cost difference between gets and puts disappears implying that the NP on the remote node has become the bottleneck.

Two other observations can be made concerning the gap results for the Paragon. First, there is no substantial change in the gap between the Proc and NP implementations in spite of removing the compute processor from the critical path. Second, the store gap is lower due to an implementation optimization that bunches together acknowledgments. Note that even though the language does not require acknowledgments for stores, they are sent to ensure flow control in the network (for all versions) and availability of buffer space (for the Proc version).

It is also interesting to note that the NOW-NP implementation trades-off gap for latency by having the NP be responsible for both interfacing with the NI as well as handling the messages. While this approach lowers latency by eliminating messages transfers between the CP and the NP, it increases the load on the NP. If we view the various components of the system as different stages of a pipeline, the NP strategy eliminates some of the stages in the pipeline while increasing the time spent in the longest stage. Consequently, it improves latency at the expense of gap.

Gap-2: We can further isolate the system bottleneck by modifying the gap microbenchmark to issue gets, puts, and stores to two remote nodes. If an operation can be issued more frequently when issued to two different nodes, then the bottleneck for the operation is the processing power of the remote node; otherwise, it is send side limited. For the CM-5, we notice that the operations are send side limited. On the Paragon, the numbers support our earlier conjecture that the NP on the source node is the bottleneck in the NP version while the NP on the destination node is the bottleneck in the Inject and Receive versions. For the Meiko-NP implementation, the remote NP is the bottleneck for gets while the NP on the source node is the bottleneck for puts and stores. Similarly, for the NOW-NP implementation, we observe that the NP on the source node is the bottleneck for gets and puts while the NP on the destination node is the bottleneck for stores.

Gap for Exchange: Our final microbenchmark measures the gap when two processors issue requests to each other simultaneously. This test exposes the issues relating to how the CP and NP divide up the workload involved in communication. As expected, for the Proc implementations on the Paragon and the Meiko, the cost is roughly twice the regular gap since the compute processor experiences the overhead for its message as well as for the remote request. Since the NP and Inject versions allow for the overlapping of resources, the gap costs increase by less than a factor of two. The Receive version, where the compute processor receives and handles messages to share the communication workload with the NP, performs better than the Inject and NP versions in spite of incurring the cost of using locks. It is interesting to observe that the Receive version has higher overhead and gap when a single node issues fetches from a remote node, but it performs better for bulk-synchronous communication patterns, such as exchange.

Conclusion: The latency, overhead, and gap measurements of the Split-C primitives quantify the combined effects of the trade-offs discussed in the previous section. In particular, we can observe the utility of having direct access to the network, a fast NP, and of optimizing the global access primitives onto available architectural support in the target platform. We are also able to observe the effect of factors like software address translation and protection checking.

5.4.2 Bandwidth for bulk transfers

Figure 5.7 shows the bandwidth curves for the bulk store and bulk get operations for the different machines. With the exception of the CM-5, all our architectures have a DMA engine to support bulk transfers. The Meiko NP implementation out-performs Meiko Proc for long messages; for stores, it achieves 38 MB/s compared to 32 MB/s. This occurs because the Proc version trades off bandwidth for latency by having the NP continuously poll for new messages from the CP. While this reduces the latency, it constantly takes

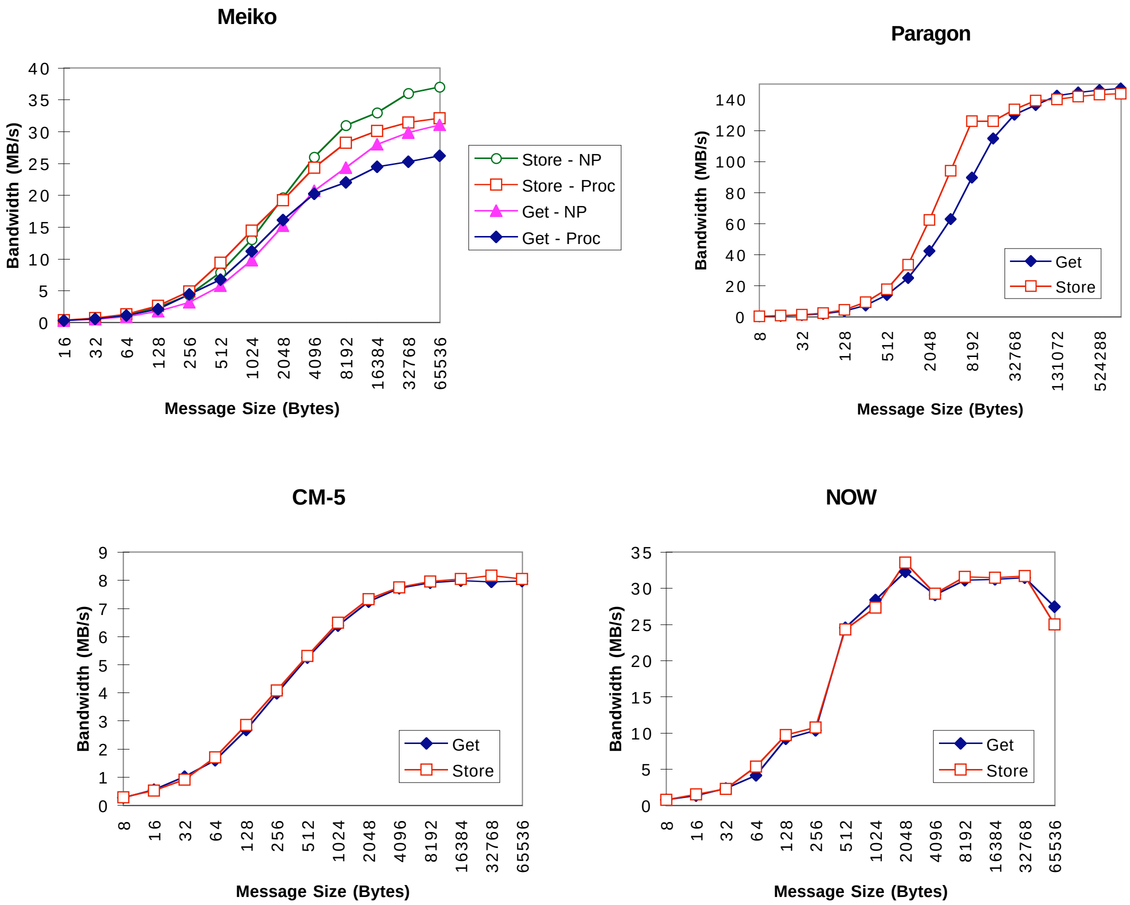


Figure 5.7: Bandwidth of Split-C bulk get and store operations for our study platforms.

resources from the NP and reduces the bandwidth for bulk transfers. The NP implementation, on the other hand, only schedules threads on the NP when needed. All the Paragon implementations use the same bulk transfer mechanism (since the device is complex to control and must be operated at kernel level) and achieve the same performance: a maximum bandwidth of 144 MB/s. The CM-5 achieves only 10 MB/s. The NOW throughput is limited by the SBus bandwidth.

5.4.3 Polling granularity

To study the impact of attentiveness on communication performance, we use a microbenchmark where each processor performs a simple compute/poll loop with the computation granularity varied based on an input parameter; after each computation the process

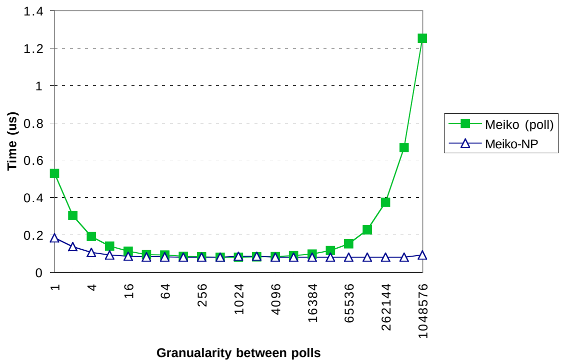


Figure 5.8: Run times per iteration for varying granularity between polls, on a 16 processor Meiko CS-2.

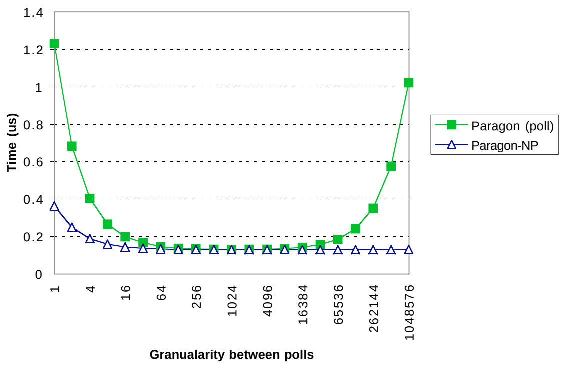


Figure 5.9: Run times per iteration for varying granularity between polls, on an 8 processor Paragon.

may poll for messages. All processors take turns computing while the remaining processors request a single data item from the busy processor. The requesting processors need this data item to make progress. If this request is not serviced immediately, the requesters idle, and only a single processor is busy computing at any given time. However, if the compute granularity is small or if a NP is used to service requests, the responses come back immediately, and all processors can work in parallel.

Figures 5.8 and 5.9 show the impact of varying the compute granularity on the overall run-time. As expected, not polling or polling infrequently results in poor performance. The NP implementation always performs well, because the NP immediately services requests. For a wide range of granularities, polling on the compute processor performs just as well as the NP implementation. Only when polling occurs very frequently does the polling overhead become noticeable.

5.4.4 Discussion

The original motivation for an NP was to enable user-level protected communication without the limitations of gang-scheduling found on machines such as the CM-5. Protection is realized by running part of the operating system on the NP, either in software as on the Paragon, or in hardware as on the Meiko. Having the NP helps decrease the overhead observed on the compute processor, and thus may enable overlapping of computation and communication to a greater degree. On some of the machines, the NP also helps reduce the gap between messages and improve bulk transfers. An important advantage of the NP is that it improves the responsiveness.

There are several factors that prevent us from realizing the full utility of the NP, including limited speed, required protection, functionality of the NP and address translation, as well as the synchronization and the shared memory access cost between main processor and NP. For example, on the Meiko, the protection check, which involves table lookups, is expensive, since the NP does not have an on-chip cache. On the Paragon, the address translation has to be performed in software. Having the NP do the sending increases the observed latency as one more step is involved. Just getting the information from the compute processor to the NP is already quite expensive, since it involves an elaborate shared memory protocol. Finally, if the NP is a specially designed chip, as in the case of the Meiko and the NOW, it is likely to be much slower than the compute processor.

5.5 Summary

There has been a clear trend in the designs of large scale parallel machines towards more sophisticated hardware support for communication, better user-level messaging capabilities, and a greater emphasis on global address-based communication. In many cases, this has led in the direction of dedicated network processors; however, there is a great deal of variation in how specialized these are to the communication task, how they interface to the processor and the network, the kind of synchronization support they provide, and the level of protection they offer.

The Split-C and Titanium communication layers provides a framework for experimentation with new communication hardware mechanisms by involving the compiler in the support for the global address operations. Global memory operations are statically typed, so the compilers can generate a short sequence of code for each potentially remote operation as required by the specific target architecture. In this chapter we studied the tradeoffs present in a large design space by implementing the communication layer on a range of these architectures and by pursuing a family of implementation strategies, each fully optimized for the capabilities of the hardware under that strategy. We see quite substantial differences in the latency, overhead, and gap exhibited on the individual global access primitives, and the differences are, in hindsight, readily explained.

The experience of the study and the measurements that it offers provide some clear design guidelines for the communication substructure of very large parallel machines. First, imposing a network processor between the application program and the network provides a very simple (although not necessarily inexpensive) means of addressing the complex requirements of protection, address translation, media arbitration, and flow-control for communication. However, it is important that the interface between the processor and the network processor be efficient. This is not necessarily achieved by traditional bus-based cache-coherency protocols, since the idea is to move information from producer to consumer quickly, rather than to hold data close to the processor that touches it. Second, there is a distinct advantage to having the network processor be responsive to the network and service memory access requests without waiting for the processor. However, if the network processor is going to do more than act as an intermediary and sanitize the network interface, it needs to be powerful enough to do this job with performance competitive with the processor. In particular, if the network processor is to provide a protection model powerful enough to allow its use on general purpose operations, it should be fast enough to be effective on those operations.

Finally, the study provides ample evidence that a global address space language like Split-C or Titanium can be implemented efficiently on a wide range of distributed memory machines, which were primarily designed for supporting message passing programs. To achieve the kind of low latency that enables access to remote memory through global pointer dereferences, we needed to take advantage of several architecture specific details to minimize the overheads associated with message transmission.

Chapter 6

Global Memory Operations using Direct Hardware Support

In this chapter, we continue our discussion of the issues involved in the implementation of global memory operations on distributed memory machines. While the previous chapter focussed on machines with hardware support for communication in the form of programmable network processors, we now consider the Cray T3D and T3E, which are machines with hardware support for a few specialized global memory operations. There are considerable performance benefits if the language primitives can be implemented using the specialized operations supported by the hardware. In addition to developing an efficient implementation of Split-C on these machines, we also provide a detailed empirical performance characterization of the two machines and identify some of the pitfalls in their design given the constraints imposed by current microprocessors.

The Cray machines were designed to provide transparent access to remote memory using the "shell" approach. In this approach the core of each node is realized by a state-of-the-art commercial microprocessor and its memory system, surrounded by a shell of additional logic to support global operations, such as communication and synchronization. The global memory operations supported by the machine are hardwired in the shell by having the memory controller intercept certain kinds of memory accesses and transform those operations into appropriate network transactions. For numerous reasons, the Cray machines provide a very interesting case study for implementing a global address space layer: the shell is complex and elaborate, the semantics of the hardware primitives for global operations are at essentially the same level as the language primitives, and many distinct mechanisms exist to perform the same function.

The difficulty this design presents for language implementation is two-fold. First, we need to map the language primitives onto the hardware primitives that correctly implement the desired semantics. In many cases, there are still several options for implementing a language primitive. The second step, therefore, is to choose the primitives with the best performance. We need to establish the performance of the machine primitives taking into account the complex interactions between the microprocessor and the shell. The performance of the memory system also needs to be carefully examined as the implementation of the global operations is closely tied to the memory system. To understand the performance issues, we follow a "gray-box" methodology, where design documents are used to establish the functional characteristics of the hardware and a set of micro-benchmarks are used to characterize its performance empirically.

The rest of the chapter is organized as follows. In Section 6.1 we outline the basic machine architectures. In Section 6.2 we explain our micro-benchmarking methodology and characterize the data access performance of an individual node. In Section 6.3 we explain how the language primitives are mapped to the analogous hardware concepts and identify performance concerns that arise. We extend our micro-benchmarking to characterize reads and writes to the global address space in Section 6.4, and identify semantic problems attributable either to the Alpha processor or the shell. In Section 6.5 we consider memory operations that overlap communication with computation, and in Section 6.6 we investigate mechanisms for bulk transfer. In each of the sections, we outline the requirements of the language model as well as the structure, constraints, or performance characteristics of the machine that dictate how the language is implemented.

6.1 Overview of Cray-T3D and Cray-T3E

The CRAY-T3D is a massively parallel processor, consisting of up to 2,048 Alpha 21064 nodes with 16 to 64 MB of memory each and a "shell" of support circuitry to provide global memory access and synchronization as part of the interface to the network. Here we discuss the key features of the design; see [36] or [37] for an overview and [21] for a complete functional description.

The DEC Alpha 21064[61] is a 64-bit, dual-issue, second generation RISC processor, clocked at 150 MHz (6.67 ns cycle), with 8 KB instruction and data caches, each with 32-byte lines. The Alpha operates on 64-bit data values, whether integer or floating point, and has only word (32-bit) and long word (64-bit) memory operations. Accesses to smaller data types use powerful byte manipulation instructions. Stores are non-blocking and loads and stores may be reordered, so a memory barrier instruction is required to serialize memory references. The 21064 supports a 43-bit virtual address space, but can only address 4 GB of physical memory.

The T3D shell provides several features to support global operation under a primarily shared-memory paradigm; the simplest of these lets a processor access any memory location in the machine through a standard load or store instruction. However, since the physical address space is small, the remote processor number is obtained from one of 32 external registers, called the DTB Annex, indexed by five bits of the physical address. Additional fields in each Annex entry control the mode of remote operation. The opcodes for the Alpha load-locked and store-conditional instructions are redefined on the T3D to allow read and write access to the Annex registers. The shell also supports an atomic-swap between a shell register and memory and two fetch&increment registers per processor, as well as global-OR and global-AND barriers. The Alpha has a fetch hint instruction, which is interpreted by the shell as a binding prefetch into a prefetch FIFO, which is popped by the processor through loads from a memory-mapped address. The Alpha memory barrier instruction ensures that writes and prefetches have been delivered to the shell; an additional status bit indicates whether any remote accesses are outstanding. The shell provides a system-level block transfer engine, which can DMA-transfer large blocks of contiguous or strided data to or from remote memories. Finally, the shell provides a user-level message send FIFO; arrival of a message at the receiver either places the message into a user-level message queue or invokes a specific user thread. Processors are grouped in pairs that share a network interface and block-transfer engine, and these 2-processor nodes are connected via a three-dimensional torus network with 300 MB/s links. The network uses dimension-order routing and incurs a very small latency per hop. These machine features will be discussed in more detail in later sections.

The Cray T3E [22, 57] was designed with a faster processor as the core and a simpler shell with less mechanisms than the T3D shell. The T3E uses the 300MHz quad-issue Alpha 21164 microprocessor [27], which has a second level on-chip cache and added support for memory accesses, namely a write memory barrier. (Like the T3D, the T3E has no board-level cache.) The T3E introduces extra hardware logic to support virtual to physical address translation at the destination node for remote accesses; this eliminates the difficulty of addresses that are legal on one node but not on others. The T3E's design is greatly simplified by having a single remote access mechanism supported by a large set of off-chip registers, called the E-registers, as opposed to a wide variety of remote access mechanisms on the T3D. A detailed functional description of the synchronization and communication mechanisms of the machine can be found in Scott's machine overview [57].

On initial inspection, the T3E is a significant improvement over the T3D: It has a faster clock, more on-chip cache (organized in two levels), and a higher bandwidth network. Figure 6.1 summarizes these differences.

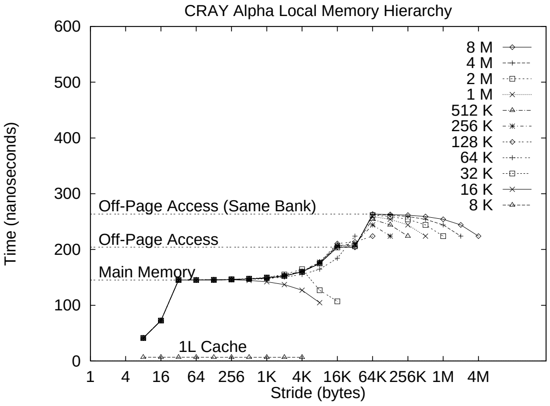
	T3D	T3E
processor	Alpha 21064	Alpha 21164
clock speed	150 MHz	300 MHz
instructions per cycle (maximum)	2	4
L1 cache size	8 KB	8 KB
L2 cache size	–	96 KB
network latency	.55 us	.5 us
peak link bandwidth	300 MB/s	600 MB/s

Figure 6.1: Peak performance of hardware components on the Cray T3D and T3E.

6.2 Local-Node Performance

In this section we introduce our micro-benchmarking strategy, which is used to decide what implementation approach to adopt. We use micro-benchmarking to expose the structure and costs of the local memory system. The results prove critical to understanding the costs and consistency issues of global operations because the memory system is the primary gateway to the shell.

Our experiments are inspired by Saavedra's micro-benchmarks [53, 52]. We treat the system as a gray box – admitting that we have some *a priori* knowledge of the system, but that it is both incomplete and unverified. Simple probes are used to determine the parameters and characteristics of the machine. We work from the bottom-up, analyzing the simplest primitives first so the results can be used to help understand increasingly more complex mechanisms. Our probes are written in assembly language so that measurements reflect actual hardware costs, not overhead inflicted by the compiler or message passing library. As the probes are simple, using assembly is not overly burdensome, and it leads to measurements that are physically meaningful and therefore comparable across machines.



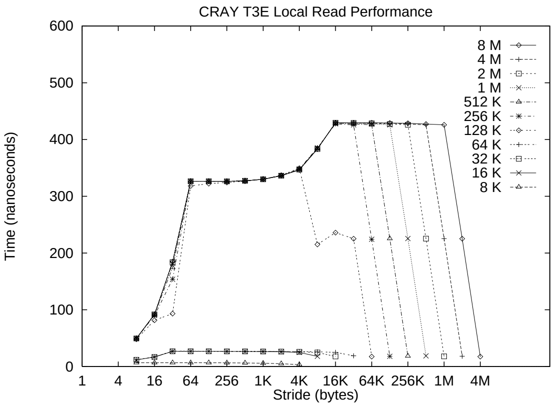


Figure 6.2: Local Memory Hierarchy Comparison: Average read latency for CRAY-T3D and CRAY-T3E.

6.2.1 Local Read Latency

Our first experiment characterizes the latency of a local read. The idea is to generate a controlled stream of addresses to the memory system, *i.e.*, a stimulus, and to observe the response in terms of the average latency per memory access. By varying the parameters that define the address stream, *i.e.*, by varying the stimulus, and observing the variations in the response we can infer specific properties of the memory system. The probe, derived from [53] with slight modifications, simply steps through an array of a given size with a given stride. By increasing the size of the array, we increase the range of addresses within the stream. By increasing the stride, we increase the frequency at which the address varies from low to high. The stimulus is a sawtooth wave, and can be described with the pseudo-code:

```
for (arraySize = 4 KB; arraySize < 8 MB; arraySize *= 2)
  for (stride = 1; stride <= arraySize/2; stride *= 2)
    for (i = 0; i < arraySize; i += stride)
      MEMORY OPERATION ON A[i];
```

We surround the innermost loop with an additional loop that repeats the experiment to mitigate timer granularity and obtain a suitable confidence level. All loop and address calculation overhead is subtracted out so that the reported time reflects only the time of the requisite memory operation. We use two separate probes: reads from memory location A[i], and writes to memory location A[i]. Note that A[i] is an 8-byte word. In each case we plot the average latency curve as a function of stride for a range of array sizes. We conduct the read experiment on the Cray T3D and the T3E. The results for the read experiment are shown Figure 6.2.

Analysis of T3D Data: The graph shows that reads take an average of 6.67 nanoseconds for array sizes up to 8 KB, matching the cycle time (150 MHz) of the microprocessor and the published size of the on-chip first-level data cache, respectively. (Note that while a read issues in one cycle, the result is not accessible for two more cycles.) When the size of the array exceeds the size of the data cache, the reads begin to generate misses. The average access time is now the weighted sum of the hit time and miss time, so an inflection point occurs when every read generates a miss. This point reveals the cache-line size of 32 bytes and the full memory access time of roughly 145 ns (22 cycles). We can tell that the cache is direct mapped because the access time does not drop to the cache-hit time for large strides. If the cache had an associativity of two, for example, there would have been a drop when the stride was half of the array size since the two addresses being accessed would fit in a single set.

As the array size and stride continue to increase we see another rise in latency. This effect is due to the internals of DRAM: strides of 16 KB or greater result in off-page DRAM accesses with each subsequent load. The net result is an additional 60 ns (9 cycles) of latency. At 64 KB strides, the effect is slightly worse, again due to the organization of the memory system. Since there are 4 memory banks, every access with a stride of 64 KB accesses the same bank, thus exposing the full memory-cycle time. This brings the total worst-case memory access to 264 ns (40 cycles).

In most systems, this secondary increase in latency is indicative of the translation look-aside buffer (TLB), which caches a limited number of virtual to physical translations. However, on the T3D the rise occurs at too small an array size (32 KB) for it to be caused by TLB costs, as this would imply a 2-entry TLB with the smallest possible page size of 8 KB. The absence of a rise in latency attributable to the TLB indicates that the T3D designers have chosen to use very large page sizes. This resolves a potentially difficult code-generation issue regarding global pointers, and will be discussed in Section 6.3. The graph also shows that there is no second-level (L2) cache on the T3D; an L2 cache would reveal itself as an intermediary latency between the L1 cache cost and the full memory access time. Eliminating the L2 cache allows for higher memory bandwidth when streaming through very large data sets, as is typical of array-based scientific codes[37]. The T3D can deliver roughly 220 MB/s from memory into the processor, while a workstation built around the DEC Alpha can deliver only about half that amount.

Analysis of T3E Data: The T3E data also reveals a first level cache that is 8KB in size with direct-mapped 32 byte cache lines. In addition, there is an on-chip second level cache, which operates with 64 byte cache lines. The second level cache size is between 64KB and 128KB with an associativity of three. We also observe that the DRAM is organized into 16KB pages.

From the experimental data, it appears that T3E's L2 cache is a mixed benefit. The read latencies are considerably lower on the T3E when the data fits into the second level cache. However, the main memory latencies are significantly higher, which would significantly slow down codes that access large data sets. We can also predict an increase in the cost of the shell operations, which are usually initiated and monitored through off-chip operations on memory-mapped locations. In summary, the T3E node is less "Cray-like" and more like a workstation. As with the T3D node, the T3E results do not show any increases in read latency that can be attributed to TLB misses.

6.2.2 Local Writes

Our second probe, which updates the array in the inner loop, produces the write latency profile shown in Figure 6.3. The difference between the read and write profiles is dramatic, but not surprising. Much of the difference is due to the write buffer hiding the latency of writes to memory.

The Alpha 21064 has a write-through, read-allocate data cache. We observe the presence of the write buffer by noting that every write to the cache does not incur the full memory latency. Note that the inflection point at a stride of 16 KB occurs because each successive store to the array causes an off-page DRAM access in the memory controller.

For very small strides, the average access time is about 20 nanoseconds (3 cycles), but at a stride of 32 bytes (the block size of the cache), the average access time goes up to 35 nanoseconds. We can draw two conclusions from these measurements. First, at very

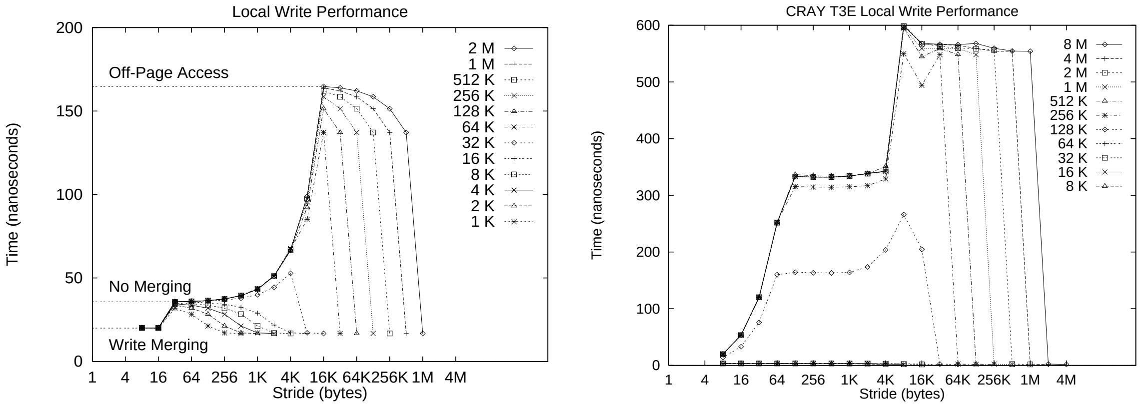


Figure 6.3: Local Write Cost.

small strides, successive writes to the same line are written to the same entry in the write buffer. This is a feature known as *write-merging* [61]. Second, since main memory access time is roughly 145 nanoseconds, dividing this by 35 nanoseconds gives an estimated write buffer size of 4. This is corroborated by the Alpha 21064 Reference Manual[26].

The L2 cache on the T3E is a write-back write-allocate cache, which results in very low latencies for small data sets as the system eliminates all of the main memory accesses. However, the write-back property of the second level cache results in relatively high latencies for large arrays where every access involves a write back of a dirty line in addition to fetching a new cache line. The other differences in the performance profiles of the two machines can again be attributed to the miss penalty of T3E's L2 cache.

This section has provided a detailed examination of the local memory system. We have determined the cost of an off-chip memory access, that the large page size essentially eliminates TLB costs, and that the write buffer contains four entries and supports write-merging. Later sections will discuss how these costs and mechanisms affect the language implementation.

6.3 Global Pointers

In this section we turn to the first of our code-generation challenges: the representation of global pointers and the operations on these pointers. We discuss the machine features relevant to the implementation. On the T3D, the implementation of global pointers involves a complex aspect of the T3D shell that provides a level of physical address translation and interacts with the TLB. The T3E implementation is much simpler as the machine hides the translation from the user and provides a simpler interface. We now discuss the issues involved in supporting the following operations on global pointers.

- *Extraction and construction:* The global pointer can be taken apart to obtain its processor number and local address components. A global pointer can also be constructed from these components.
- *Dereference:* The location referenced by the pointer is read or written.
- *Transfer:* The pointer may be passed as a parameter or stored in an object.
- *Arithmetic:* The arithmetic operations act on a global pointer as the corresponding operations would act on a standard pointer, *i.e.*, an incremented pointer refers to the next location on the same processor.

6.3.1 T3D Design Constraints

The 21064 implementation of the Alpha architecture supports 43-bit virtual addresses and 32-bit physical addresses (two additional high bits support memory mapped devices). This has a significant impact on how the machine supports shared memory. Many shared-memory machines map all memory accessible to a process into the virtual address space and a virtual address is translated into a *global* physical address [9, 42, 35, 38, 39]. The memory system extracts the node number and physical location from the global physical address and performs either a local memory access or a message transaction with a remote memory controller. For a fully configured T3D, this would require at least 37 bits of physical address. Since there not that many bits available, the T3D shell performs an additional level of address translation using a set of 32 segment registers known as the DTB annex (Figure 6.4). Each Annex register specifies a processor number and function code. Annex registers are updated at user level with the revised store-conditional instruction at a measured cost typical of off-chip access, 23 cycles. Annex register 0 always refers to the local processor.

The page tables are constructed to provide shared stack and heap segments containing 32 regions of 128 MB each, one per Annex register. The virtual-to-physical translation performed in the Alpha carries the Annex register index through and translates the

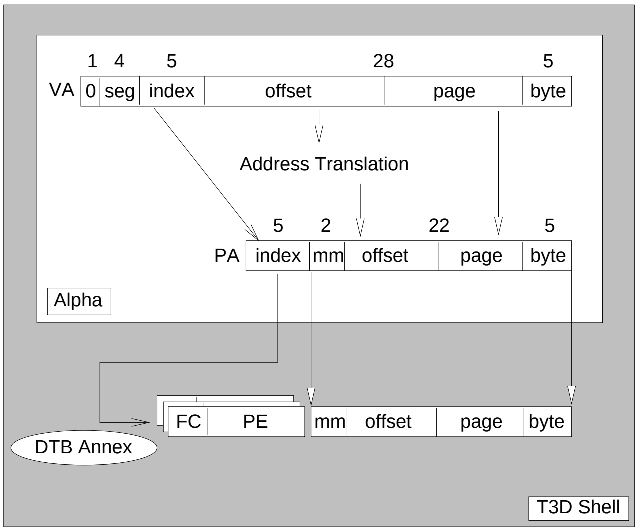


Figure 6.4: The DTB Annex: A 43-bit Virtual Address (VA) is translated to a 34-bit Physical Address (PA), followed by the index into the Annex. Each of 32 Annex entries specifies a function code (FC) and a remote processor (PE).

remainder of the address to a 27-bit physical address, which must be valid on all processors. Thus, one must view a virtual address as a *temporary* global address; its meaning is dependent on the configuration of the Annex. [1]

A remote access involves storing the destination node number into one of the Annex entries, and then accessing the desired offset in the corresponding segment of the virtual address space. Thus, the Annex presents a new set of registers that the compiler must manage and a short sequence of instructions is required to perform a remote access.

6.3.2 Global Pointer Representation on the T3D

A natural choice for the global pointer representation is a 64-bit entity with the local address stored in the lower 48 bits and the processor number stored in the upper 16 bits. This is the same size as a local address, so transfer is efficient. The Alpha has powerful byte manipulation instructions, so extracting the components of a global pointer and performing arithmetic are also fast. In fact, with the particular virtual memory mapping in the T3D, the 42nd bit of any virtual address is zero. Thus, local address arithmetic is performed on

[1] An alternative would have been to provide truly global virtual addresses and have the operating system manage the Annex transparently. The page tables would associate addresses for the currently mapped remote processors with the appropriate Annex indexes and a fault would occur on reference to an un-mapped remote processor.

global pointers exactly as it is on local pointers; the result should never overflow into the processor portion of the address. The meaning of a global pointer is independent of the processor that dereferences it, so they can be freely placed in shared data structures.

On dereference, the processor number is extracted and stored in an Annex register, and the annex entry number is inserted in the appropriate bit-field to generate a valid virtual address for use in a load or store instruction. Local arithmetic can be performed on this "internal" global pointer and multiple references can be made without incurring the cost of annex setup.

6.3.3 Annex Register Management on the T3D

A key question underlying the use of global pointers is how the compiler manages the Annex registers. The simplest approach is to use only one Annex register and update it on each global access, skipping the Annex update if the compiler can determine that successive accesses are to the same processor. An alternative is to use several Annex registers and keep a runtime table of their entries, with a table lookup added to each global access. This could be combined with compiler analysis to eliminate the table lookup when there is sufficient static information about the pointers.

On the surface, using multiple Annex registers appears to be the better alternative, but it leads to subtle semantic problems and in the end there is no clear performance advantage. The semantic problem occurs if two Annex registers specify the same processor. Because the Annex performs address translation on physical addresses, this allows synonyms: two physical addresses that differ only in their Annex index may map to the same location. Both the cache and write buffer use physical addresses to determine when two accesses are to the same location, so synonyms potentially lead to inconsistent copies. Inconsistencies from caching do not arise on the T3D because the annex entry appears in the high order bits of the address and the cache is direct-mapped, so two synonyms always map onto the same cache line. Unfortunately, the write buffer does admit inconsistencies, since reads can bypass writes. If a write is followed by a read to a synonym, the read may see a stale value from memory while the write is caught in the write buffer. We have produced probes that exhibit this unpleasant phenomenon. Thus, in order to use multiple Annex registers, the compiler must recognize aliases between global pointers or it must generate runtime checks to prevent synonyms.

If a runtime table of Annex entries is kept, the Annex register will need to be selected by hashing the processor portion of the global pointer. Even a simple table lookup requires a memory read and a branch, so the savings relative to a 23-cycle Annex update are small.

A final consideration in using multiple Annex registers is whether this interacts poorly with other aspects of the memory system. Since the Annex register number appears in the high order bits of the virtual address, remote accesses will consume TLB entries. Our concerns on this point where allayed when we determined that the processor was configured to use huge pages and therefore few TLB entries.

6.3.4 Global Pointer Operations on the T3E

The T3E has a markedly different design. There is no machine-defined notion of a global address in spite of the increased addressing capabilities (40 physical address bits) of the 21164 processor. The hardware does not overload the use of load and store instructions for accessing remote memory. Instead, a short sequence of loads and stores to memory mapped external registers (E-register) initiate and complete a remote access. More specifically, a remote load is implemented by writing the target processor number and address to a memory mapped location followed by a load from a E-register, which returns the fetched value. A remote store requires an E-register to contain the value to be written before the store operation is initiated by writing the processor number and the address on a memory mapped location. The virtual address of the target location is communicated in its untranslated form to the destination node where the shell provides logic to do address translation. As with the T3D, since the Alpha processor supports 64-bit addresses internally, and also has efficient byte manipulation instructions, we store the processor number and address within a single word at a byte boundary.

Code generation for the T3E is much simplified due to the absence of the Annex registers. One could view T3D's design as an attempt to expose to the compiler the different components of a global memory operation. The task of updating the Annex entries is separated from the actual memory operation allowing the compiler to schedule and possibly eliminate the Annex operations. However, as we saw in the previous section, the effectiveness of this design is undermined by the difficulties associated with eliminating the Annex updates at compile time. The T3E, on the other hand, does not separate the address translation mechanism from dereference and therefore provides a simpler interface to the compiler. The disadvantage, however, is that the address translation costs are incurred for every remote access, which could potentially result in higher remote access latencies. We will examine this performance issue in the next section.

6.3.5 Summary

Global pointers are easily represented on the Cray machines in a form almost identical to that of local pointers and consequently address arithmetic on global pointers is extremely fast. The implementation is aided by the 64-bit architecture of the Alpha as well as its powerful byte manipulation instructions. However, the small physical address space of the 21064 presents a serious problem for large shared memory multiprocessors, even when the compiler explicitly manages the global address space. The use of external segment registers on the T3D to extend the effective physical addressing interacts poorly with the memory system and complicates code generation. The provision of many external segment registers appears to be of small value. These problems are successfully addressed on the T3E by providing a simpler interface in the form of E-registers, and the apparent downside of not allowing load and store instructions on remote memory is minimized.

6.4 Remote Reads and Writes

This section describes the implementation of the simplest Split-C remote access primitives, *read* and *write*, which are blocking remote access operations. We characterize the performance of the hardware mechanisms and then describe our resulting implementation.

T3D's shell provides a large set of mechanisms for reading and writing to the global address space, each with its own semantics and costs. The compiler writer must determine which of these mechanisms can be used to implement a language primitive and then choose the fastest. T3E, on the other hand, offers a single powerful mechanism for remote access, thus easing the implementation task. However, on both machines there are semantic mismatches – discrepancies between the behavior defined by the language and that implemented by the machine – that required awkward workarounds.

6.4.1 T3D Remote Reads

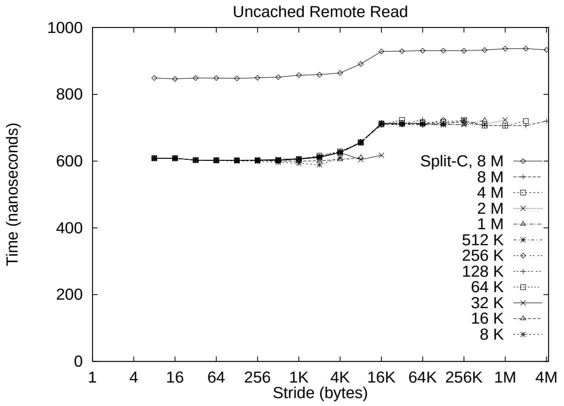
The T3D provides two types of remote reads: cached and uncached. Both use the standard Alpha load instruction on a global address, but a function code in the DTB Annex entry specifies the type of read. A cached read updates the requested line (32 bytes) in the the cache and places the requested word into the specified register. An uncached read fetches only the requested word and does not modify the cache.

We determine the latency of remote reads by modifying the local micro-benchmark so that it strides through remote memory. The results are shown in Figure 6.5. An uncached read costs roughly 610 ns (91 cycles) while a cached read costs 765 ns (114 cycles). As we saw with local-memory operations, strides of 16 KB or greater cause an off-page DRAM access (this time in the *remote* node's memory controller) which increases the average access time by about 100 ns (15 cycles). The cost of cached reads depends on the contents of the local cache. In our micro-benchmark, array sizes that fit in the cache result in local-cache memory-access times since the first time through the array brings the data local for all successive iterations. Cached reads also perform slightly differently for strides of 8 and 16 bytes. By bringing over entire cache lines, cached reads essentially prefetch the next 1 or 3 accesses. Note that all of our measurements are to an adjacent node, with only one processor active; measuring the additional latency through the network reveals roughly a 13 to 20 ns (2-3 cycle) cost per hop.

To put these costs in perspective, remember that a remote uncached read is only three to four times slower than a read from local memory. In fact, the latency to remote memory on the T3D is only 80 ns higher than a main memory access (including TLB miss) on the DEC workstation. This also is significantly faster than similar MPPs: the cost of a read to a remote node on the DASH multiprocessor is roughly 3 μs, and is about 7.5 μs on the KSR [59].

6.4.2 T3E Remote Writes

The remote read latencies are substantially higher for the T3E than the T3D. The increased latencies are primarily due to the extra logic that the T3E requires for address translation on the target node and higher main memory access costs. Recall that T3D required no additional address translation mechanisms on the destination node as the virtual address translation is handled by the source node's Alpha processor. The T3E shell



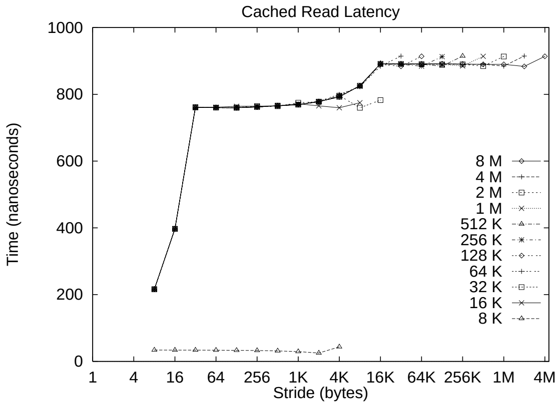


Figure 6.5: Remote Read Latency: Uncached and Cached Read Latency on the T3D.

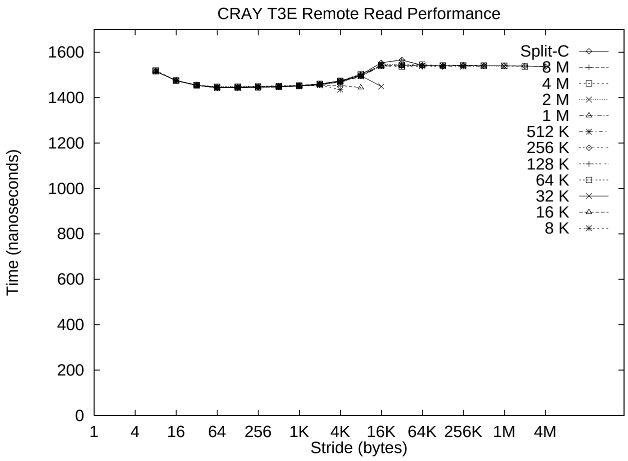


Figure 6.6: Remote Memory Read Performance on the Cray-T3E.

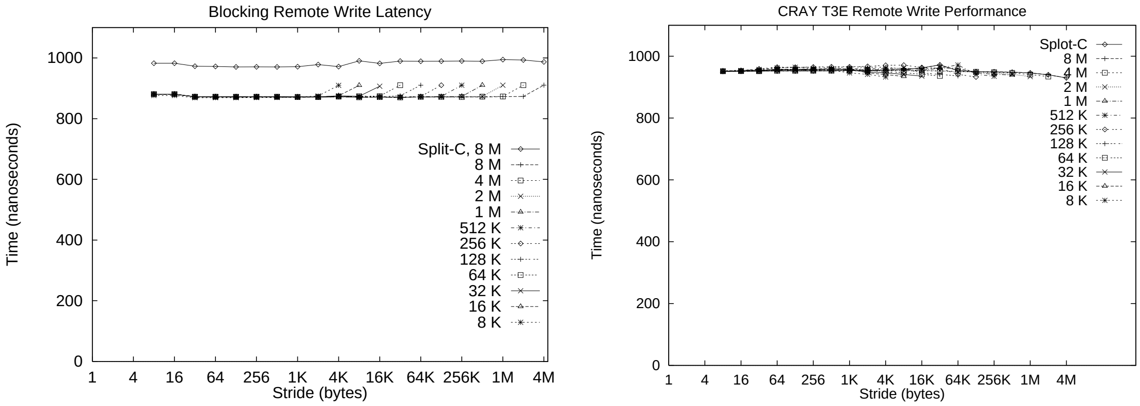


Figure 6.7: Remote Write Latency: The time to write a word into the network and get an acknowledgement.

incurs a performance cost from its attempt to duplicate Alpha's efficient on-chip address translation mechanism. We have devised experiments that show that the additional cost introduced by the address translation logic is about 450ns.

6.4.3 Remote Writes

The next primitive we examine is the remote write. On both machines, an off-chip flag is polled to check for completion of the write operation. Figure 6.7 shows the results of the experiment. There is very little difference between the raw hardware write latencies on the two machines. This behavior is very surprising given that the remote read latency on the T3E is substantially higher than that on the T3D. A possible explanation for this behavior is that writes are acknowledged as they are received by the remote node without waiting for the corresponding address translation and memory operation to complete

One subtlety arose while implementing the blocking writes that illustrates the unwanted interactions that can occur between the custom shell and the commodity core. The remote-write status bit is set when there are writes that have left the processor and not returned, but it is unclear if there are pending remote writes in the write buffer. Therefore, to poll on this bit, one must first guarantee that the corresponding write(s) have left the buffer. This can be done by issuing a memory barrier instruction or a sufficient number of writes, which effectively increases the cost of the operation.

6.4.4 Compiler Implications

On the T3D, it was not immediately obvious which mechanism should be used to implement the Split-C *read* primitive. Cached reads offer a higher bandwidth, but are difficult to use because the machine does not guarantee coherence of cached values. If the processor that owns the cache line updates it, the change is not reflected on remote processors that are caching the same line. Reads to the cache line would then obtain stale data. Therefore, if cached reads were used for reading remote values, the cache line would have to be flushed after the read. This restriction makes cached reads less advantageous since a cache-line flush costs an extra 23 cycles (equivalent to accessing main memory). Languages such as HPF might be able to perform some type of global analysis to determine when data-sharing occurs (and thus when to flush), but this is not possible in a C-like language. Therefore, the Split-C implementation uses uncached reads. Figure 6.5 shows the total latency as seen by the programmer, which is about 850 ns (128 cycles). The majority of this is attributed to the remote read and Annex set-up time.

The Split-C implementation of remote accesses on the T3D includes the cost of manipulating the Annex in addition to the raw hardware costs. As discussed earlier, eliminating Annex setup costs requires complex program analysis for the compiler to ensure that physical address synonyms are not generated. However, on the T3E, the Split-C implementation does not incur any additional cost beyond the raw hardware performance.

The Split-C *write* primitive uses the write mechanism followed by polling and requires a total latency of 981 ns (147 cycles), as seen in Figure 6.7. However, on the T3D, the operation has a subtle associated cost. When a processor initiates a remote write, the object being written to might be cached by the remote processor that owns the object. In order to maintain coherence, the T3D allows the caches to be operated in a *cache-invalidate* mode. When a remote write request is received, the corresponding cache line is flushed even if that particular cache line isn't currently cached. In other words, we are forced to operate in this mode in the absence of higher-level information, and doing so may generate spurious cache flushes in order to preserve coherence.

Finally, the design choices of using an Annex table and source-based address translation, in the absence of OS support, result in a rather tricky language implementation problem on the T3D. Since the Annex contents determine the actual translation of a global address, and since it is impractical to access the Annex during a virtual address translation, the designers decree that the address mapping must be uniform across all processors. To preserve this property, heap memory allocation for globally visible data requires a coordinated allocation strategy across processors. Note, however, that this issue could be resolved with OS support for managing the Annex (just like the OS handles TLB misses) without requiring specialized hardware for address translation at the destination node as found in the T3E.

6.4.5 Semantic Mismatches

On both machines, there are two idiosyncrasies in implementing reads and writes, caused by tension between the machine's shell and core, that led to serious semantic mismatches between the machine and the language:

Byte Writes: The 21064 processor does not support byte store operations. The processor however has a set of powerful byte manipulation instructions. A byte store could therefore be effected by a read-modify-write sequence. While this solution works on a local node, it breaks down in a multiprocessor context like the T3D. The T3E has the capacity to support remote byte store operations since a remote write operation is disguised as a store of a virtual address to a memory mapped location. Unfortunately, the shell does not support a byte store operation to a remote node. Byte writes have to be implemented on top of a more heavy-weight mechanism such as Active Messages [65]. A significant improvement though is that the handler code on the T3E can implement atomic operations using the load-locked/store-conditional instructions, which were not available on the T3D since the shell interprets these instructions as Annex operations.

Global-Local Consistency Issues: The Split-C implementation of remote reads and writes wait for the completion of the operation. Split-C also allows the owner of a globally accessible value to access the value through local pointers. However, since loads and stores through local pointers can be reordered due to the write buffer, the implementation could cause violations of sequential consistency. This problem is generic to any MPP system that allows a remote node to access memory locations without involving the local node.[2] Since we would like to avoid the rather unfeasible solution of flushing the write buffer after every store operation, this problem indicates that a private-shared distinction in the language's type system would allow the compiler to statically determine when a flush of the write-buffer

[2] On systems where a Split-C handler is executed by the local node, the handler would see the values languishing in the write buffer.

is necessary.

6.5 Split-Phase Accesses

We now discuss the issues involved in implementing Split-C's *get* and *put* primitives, which are split-phase, non-blocking operations that can be used to overlap communication and computation. A *get* operation initiates a prefetch from a remote address to a local address, while a *put* operation initiates a non-blocking write to a remote location. The *sync* operation waits for all outstanding split-phase accesses to complete. The *get* and *put* operations lead to weak consistency semantics because the accesses initiated between two *syncs* can no longer be ordered.

6.5.1 Prefetching

The T3D's prefetch mechanism uses the Alpha fetch hint instruction to provide non-blocking reads of remote addresses. The fetch instruction informs the support circuitry to fetch a word from a remote node while the processor continues computation. The fetch request is placed in the write buffer and is eventually sent to the specified remote processor. When the processor actually needs the requested data it must pop the word(s) from the 16-entry memory-mapped prefetch queue by issuing a load instruction. On the T3E, as we had discussed in the previous section, a remote read is implemented by issuing a store operation to a memory mapped shell location followed by a load operation from an E-register. The remote access latency can be masked by simply delaying the E-register load.

To examine the performance of the prefetch mechanism we used a simple benchmark that measures the average latency of prefetching one or more words. The benchmark groups n remote read requests into a series of prefetch instructions and then pops the results from the prefetch queue and stores them to local memory. The results indicate the extent to which the prefetch mechanism can overlap remote access costs (Figure 6.8).

On the T3D, one (prefetch, pop from queue, store) sequence is slower than a blocking read by about 15 cycles, but issuing four prefetches, popping the queue four times, and storing all the results to local memory is significantly faster than four blocking reads. The prefetch essentially allows the network to be pipelined, thereby masking most of its latency. An important conclusion we can draw from Figure 6.8 is that the remote latency is almost entirely hidden as the size of the group approaches 16. Therefore, the choice of

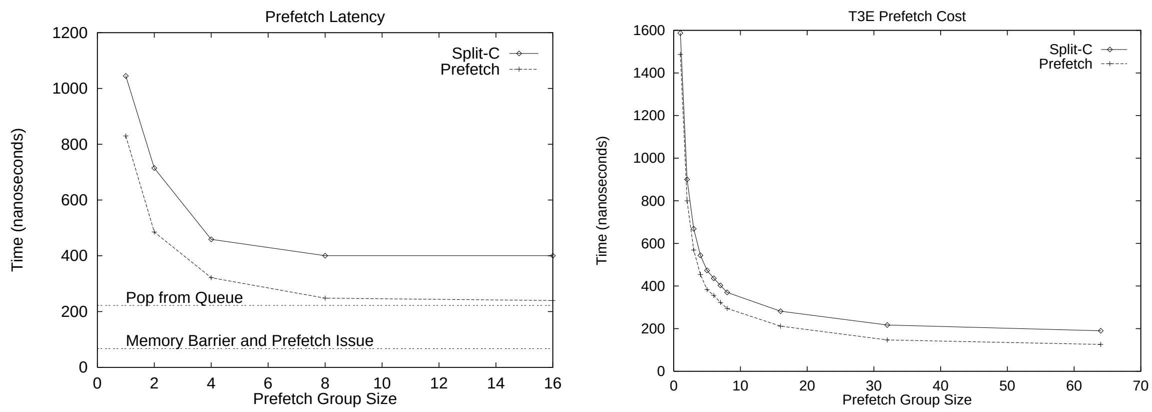


Figure 6.8: Prefetch Latency: Average latency for prefetches based on number issued.

16 for the size of the prefetch queue seems to be a reasonable one. Note that when less than 4 prefetches are issued, a memory barrier must be inserted before the pop from queue. This guarantees that the prefetch has left the processor. Most of the latency gets hidden by grouping prefetches with the cost stabilizing at about 250ns per prefetch. Since the memory access costs are less than 150ns (based on our local-node measurements), the performance limiting factor is the cost of crossing the boundary between the processor and the shell. The T3E attains a lower gap of about 130ns. Further instrumentation of the prefetch operation reveals that remote memory read accesses can be pipelined at the rate of one every 67ns, while the cost of doing a store followed by a load of an off-chip register is 120ns. We are now beginning to see the effect of having a second level cache, which increases the cost of shell operations.

6.5.2 Non-blocking Writes

Recall that on both machines, an off-chip flag is polled to check for completion of a remote write operation. A non-blocking write operation simply eliminates the check. The performance results for non-blocking writes are shown in Figure 6.9. We observe the write merging behavior even for remote writes on the T3D. However, since a remote write operation on the T3E is disguised as a store of an address value into a memory mapped register, the remote stores do not merge in the write buffer. Consequently, the benefits of spatial locality for updating remote data structures are lost. On both machines, the memory operation at the remote node is the performance limiting factor, which is in contrast to the

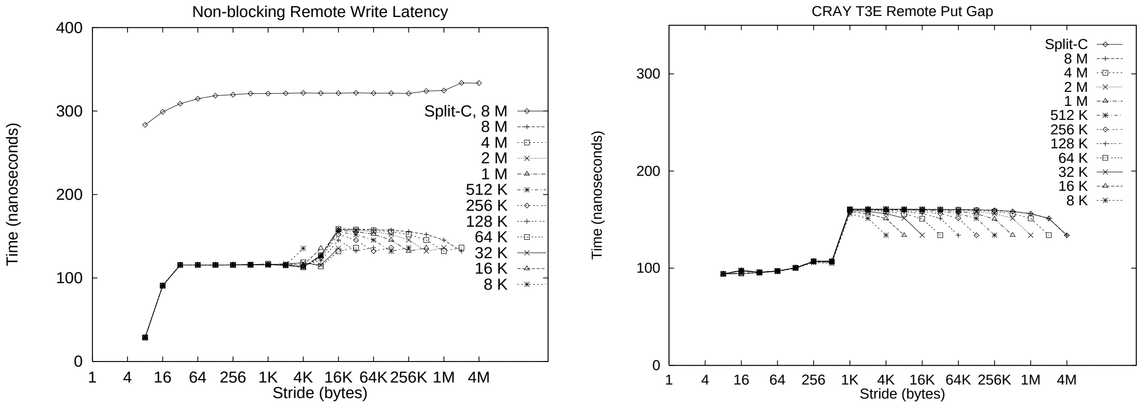


Figure 6.9: Non-blocking Write Latency: The average time to write to a remote processor (no acknowledgement).

performance characteristics of the prefetch mechanisms.

6.5.3 Compiler Implications

We can implement the Split-C *get* operation with the prefetch mechanism. However, some additional work is required since the *get* operation also specifies a local address that serves as the target for the fetched value. The target address needs to be stashed away when the fetch is issued. This can be accomplished if the compiler has sufficient information to match the prefetches and the corresponding *sync*s, which might not be possible in the presence of unstructured control constructs. In the general case, we maintain a table for storing these local addresses. The target address is stored in this table at prefetch initiation time. When a *sync* operation is encountered, or when the number of outstanding fetch operations reaches 16, a memory barrier is issued, and the values in the prefetch queue are dequeued and stored in the appropriate target addresses obtained from the table.

The Split-C implementation introduces two additional pieces of overhead over the raw hardware prefetch mechanism: the table update and the store into the local address when the prefetch is complete. Figure 6.8 reveals the final Split-C *get* cost, which includes table management, annex set-up time on the T3D, and all other overheads.

The implementation of *put* is straight-forward from the non-blocking write. The performance of the Split-C primitive is shown in Figure 6.9. Once again, we observe that the Split-C primitives, especially the *put* operation, can be implemented on top of the hardware primitives at a much lesser cost on the T3E due to the absence of the Annex registers. Though both machines have similar gap and overhead characteristics, the Split-C implementation on the T3E benefits from a closer match between the language and the machine.

6.6 Bulk Operations

Bulk transfers occur in Split-C in two different flavors: implicitly when an entire remote structure is accessed by a read, write, get, or put in a single assignment and explicitly through calls to *bulk_read*, *bulk_write*, *bulk_get*, and *bulk_put* for copying contiguous regions of global memory.

6.6.1 T3D's Bulk Transfer Mechanisms

The block-transfer engine (BLT) is essentially a DMA device: given a remote processor number, a pointer to a buffer, and a length argument, the engine initiates a transfer between the two processors' memories. The operation specified can be either a read or a write. The BLT is also capable of performing strided-array accesses. One would think that this would be the natural match for the Split-C bulk routines. However, unlike the hardware primitives we have tested so far, this mechanism for remote memory access is only available through an operating system invocation with high software overheads. As language implementors, we need to compare the BLT with the other remote access mechanisms that the T3D shell provides and answer the obvious question of which mechanism to use in implementing the bulk transfer routines.

To compare the different mechanisms for implementing bulk transfers, the appropriate metric is the bandwidth attained by the mechanisms for large reads. Using each of mechanisms described before (uncached reads, cached reads, the prefetch queue, and the bulk-transfer engine), we implemented and micro-benchmarked four bulk read operations. The left side of Figure 6.10 compares the resultant bandwidths of each implementation.

As the figure indicates, for very large reads (i.e. greater than 16 KB in size), the bulk-transfer engine achieves the highest transfer rate, peaking at roughly 140 MB/s. For reads between 128 bytes and 16 KB, the prefetch queue attains the maximum bandwidth. At this point, the overhead incurred by the BLT overcomes its superior maximum transfer rate. Though cached reads bring over four words at a time (whereas prefetches can only bring over one), the latency-hiding ability of the prefetch mechanism makes it strictly better than the cached reads, except in the case of 32 and 64 bytes[3]. At these points, the cached read can bring over either one whole cache line or two, and thereby achieve the best performance. Lastly, for very small transfers (8 bytes), the uncached read is optimal.

For bulk writes, we have two mechanisms available: the non-blocking store instruction and the BLT. Figure 6.10 shows the net difference between the two. Whether the local data to be written is in cache or not, the performance of non-blocking writes is superior to initiating the bulk-transfer engine. The bandwidth for writes from local memory to remote memory (i.e. not in the local cache) peaks at 90 MB/s, and is apparently bus limited.

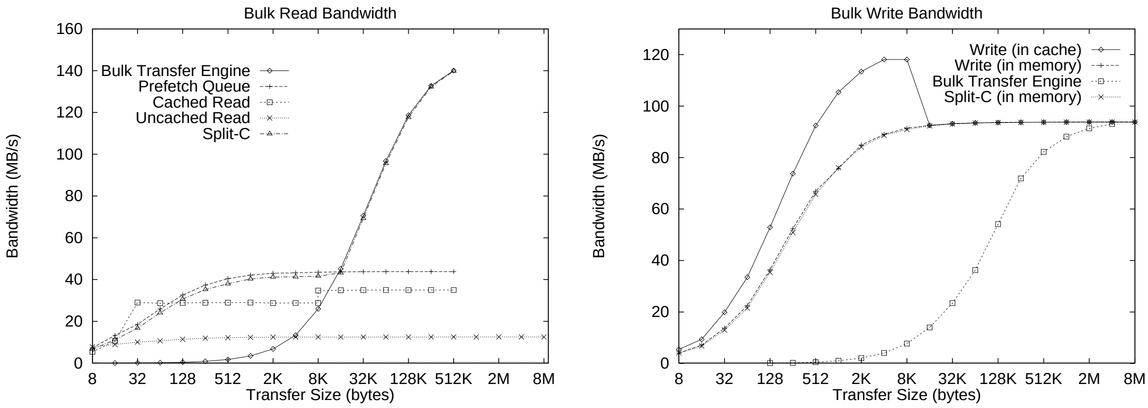


Figure 6.10: Bulk Read and Write Comparison: On the left, all the different remote access mechanisms are used to implement a bulk read. On the right, bulk writes are implemented using the BLT and the store instruction.

6.6.2 T3E's Bulk Transfer Mechanisms

The T3E design does not include the DMA engine. Instead, remote transfer using the E-registers is the only mechanism. However, one of the operations that the shell supports is a transfer of an entire cache line of data into and out of a set of E-registers, thus increasing the payload per operation. Furthermore, since E-registers can also be used for loading and storing into local memory, data can be transferred between two memory modules without going through a processor. Also, the E-register operations can be scheduled in a manner such that the cost of local memory operations are hidden by the latency of remote memory accesses. By providing more powerful operations and allowing the compiler to schedule

[3]The performance of bulk transfers using cached reads has an inflection point at 8K. This occurs because the cache line flushes (required to maintain coherence) can now be batched into an entire cache flush operation, which is less expensive.

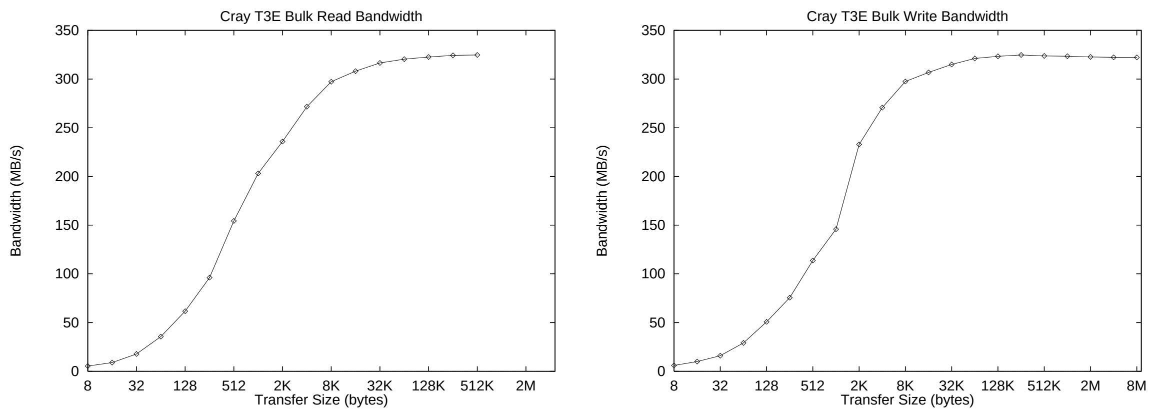


Figure 6.11: Bulk Transfers on T3E.

them, T3E allows for more efficient implementations as shown in Figure 6.11. [4]

6.6.3 Compiler Implications

The performance profiles of the T3D mechanisms suggest which mechanisms to use for code generation based on the transfer size. The key cross-over point occurs at a transfer size of about 16 KB. At this size, the bulk transfer engine should be used instead of the prefetch queue. For simplicity, the prefetch queue is used even for 32 and 64 byte transfers; switching this to cached reads would increase performance. Figure 6.10 shows the Split-C performance attained. Observe that the *bulk-read* operation achieves a significantly lower bandwidth than the *bulk-write* operation. An advanced compiler optimization, which is possible while compiling data parallel programs, is to transform a *bulk-read* executing on the source processor into a *bulk-write* from the remote processor.

There is a hidden cost associated with the *bulk-read* on the T3E. Since the data never enters the processor, the processor suffers cache misses when it actually accesses the data. Given that main memory latencies are high, the effective bandwidth could be much less than the raw performance. This behavior suggests that for smaller data transfers (less than 64K), the compiler could fetch the data into its cache in addition to writing the values into memory. Since loads from E-registers can be hidden by the latency of main memory operations, this strategy could be implemented without significantly altering the cost of the bulk transfer operation.

[4] The $N_{1/2}$ numbers for our *bulk_read* implementation are lower than that of the vendor provided *shmem* library [57].

6.7 Summary

In this chapter, we discussed the issues in obtaining an efficient implementation of a global address layer on two machines with hardware support for global memory operations. We used a systematic micro-benchmarking methodology to establish the performance of the hardware primitives. This approach is warranted because the machines provide an elaborate shell around a sophisticated microprocessor, the DEC Alpha, to support global operations which are critical to our language implementation. The performance characteristics of the many shell operations and the interactions between them are potentially non-trivial. The large menu of primitives supported by the two machines meant that there were potentially many different ways to realize the same language primitive on the machines. Careful assessment of the performance trade-offs was required to select between them. In making this selection, we have also provided a thorough empirical characterization of the performance of the many primitives supported in the machine.

The raw performance of the remote memory operations is very impressive and significantly faster in absolute terms than any previous large scale machines. Remote access is performed in about 1-1.5 μs, only four to five times the latency of a local cache miss. The prefetch support is very effective and capable of further reducing the effective remote latency. The shell provides additional operations to detect completion of remote operations. All these factors combine to facilitate a very efficient implementation of Split-C.

Chapter 7

Conclusions

We have presented compiler analyses and runtime systems for implementing an efficient shared address space abstraction for explicitly parallel programs. By providing a natural programming model based on shared memory, we allow programmers to focus on the higher-level problems of designing algorithms that are correct, have the right amount of parallelism, and are well load-balanced. The compiler is responsible for ensuring sequential consistency and performing communication optimizations. Our runtime system provides efficient mechanisms for communication by utilizing machine-specific features and resolving several crucial tradeoffs through systematic performance evaluation. Through this division of work, we have developed a system that is efficient and portable while providing a programming model with strong semantics.

Though the compiler analyses presented in this thesis enable many kinds of optimizations, we have focussed primarily interested on the communication optimizations. It is widely recognized that program transformations that mask latency of remote accesses (thereby having the program limited by the overhead of communication rather than its latency) and combine communication events (making the program bandwidth-limited) are extremely valuable in improving the performance of parallel programs. Our system is novel in its use of the compiler to automatically introduce such optimizations while preserving sequentially consistent semantics, and if such optimizations are not possible, our system is still efficient as the runtime system is optimized with respect to all three factors: latency, overhead, and bandwidth.

Our compiler analyses are independent of our runtime system efforts and can be used to target machines that provide blocking and non-blocking memory operations or communication primitives for moving data. The results from our work on runtime systems are also very general as our communication layer can be targeted by compilers for other high level languages that require efficient communication primitives. Consequently, the results from this thesis is not limited to the framework defined by our languages, Titanium and Split-C, but are applicable to other systems as well.

Our compiler relies on many different techniques that require substantial information about the program, such as control-flow information, inter-processor dependencies, local data-flow information, and synchronization behavior. Unlike parallelizing compilers, we use this information to optimize parallel code, since we believe that parallel algorithms should be designed by the user, not derived by a heroic compiler. The first analysis, cycle detection, improves on work by Shasha and Snir by making the analysis algorithm practical for SPMD programs. Whereas their framework leads to an exponential algorithm, our algorithm runs in polynomial time, and is more conservative only in rare instances that are unlikely to arise in practice. We implemented this analysis and showed that, using either formulation of the algorithm, cycle detection on its own is not sufficient to allow for any code motion or other optimizations in a set of simple application kernels. We therefore extended the programming language with a set of synchronization primitives that are known to the compiler and incorporated synchronization information into the analysis. By treating synchronization operations as special accesses, we significantly improve cycle detection and can therefore perform transformations on the example applications.

The analyses are useful for a variety of optimizations. On shared memory parallel machines with some form of weakly consistent memory, our analysis can be used to automatically introduce memory fences or synchronization instructions to ensure that the programmer observes a sequentially consistent execution. On machines with explicit prefetch or non-blocking writes, it can be used to safely convert traditional loads and stores into these non-blocking counterparts. We use Split-C with its split-phase versions of read and write as an abstract target language. Our transformations convert programs with only traditional read and write operations into optimized programs that make use of split-phase accesses.

The main optimizations described in this thesis are masking latency of remote accesses by message pipelining (split-phase writes) and prefetching (split-phase reads). We also make use of an unacknowledged write operation to reduce network traffic, bulk messages to combine several reads or writes to the same processor, and common subexpression elimination which leads to caching of remote values. We have a prototype compiler that implements the first two optimizations and quantified the payoff on a set of application kernels. The performance improvements are as high as 35% on the Berkeley-NOW. On machines with lower communication startup and longer relative latencies, the benefits for overlapping communication would be even higher. On machines with higher message startup, say from trapping to the kernel, the use of bulk operations or caching would be more important.

As part of our runtime system efforts, we developed highly optimized versions of communication primitives on important large-scale parallel platforms that have varying degrees of architectural support for communication: the Thinking Machines CM-5, Intel Paragon, Meiko CS-2, Cray T3D and T3E, and Berkeley Network of Workstations (NOW). Against these hardware alternatives, we considered a variety of implementation techniques for global memory operations, ranging from general purpose active message handlers to specialized handlers executing on a dedicated network processor or specialized memory controller. We were able to study the impact on performance of factors such as the computational power of the network processor, the nature of the interface between the compute processor and the network, and the network processor's ability to mimic the compute processor. Our work demonstrates that a small set of predefined handlers can provide higher performance without sacrificing flexibility. It also argues for a balanced system with the communications support without the investment of too much hardware. As a result of these architectural studies, we have developed implementations that attain the best round trip latencies on each one of these machines, and have also been able to expose the performance and semantic gaps between the language primitives and the machine features.

Most computing systems are developed using a layered approach with the traditional elements of the hierarchy being the machine architecture, runtime systems (including the operating system), compilers, and user programs. It is desirable to support an abstraction at the lowest level as long as the performance of the entire system is not compromised. There has been sufficient evidence to indicate that an efficient shared address space abstraction with sequentially consistent semantics cannot be provided at the hardware level or through runtime systems. In this thesis, we provide a solution that uses a combination of compiler techniques and runtime strategies for developing such an abstraction. Through our efforts, we protect the programmer from having to reason about complicated ordering semantics at the user-program level while providing an abstraction that promotes the use of powerful distributed data structures.

Appendix A

Complexity Results

A.1 Back-path Recognition is NP Hard

In this section, we show that detecting back-paths is NP hard. We prove that any instance of the Hamiltonian Path recognition problem (HPR) can be reduced to a corresponding instance of the back-path recognition problem (BPR). Given a general graph (V, E), we 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.

Let the vertices in V be $v_1, \ldots, v_n$. The HPR problem is to determine whether there exists a simple path of length $n-1$ from v_1 to v_n. We construct a parallel program with $n-1$ threads that access a set of shared variables of the form v_j^i where $1 \le i \le n$ and $1 \le j \le n$. For every vertex in V, we construct a program thread that initiates accesses to the shared variables in a particular order. The code for thread j is constructed based on the set of neighbors of vertex v_j in G. Let $w_1, \ldots, w_d$ be the neighbors of vertex v_j. We now define a macro called $CodeSeq$, which we will later use for fully specifying the code for thread j.

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

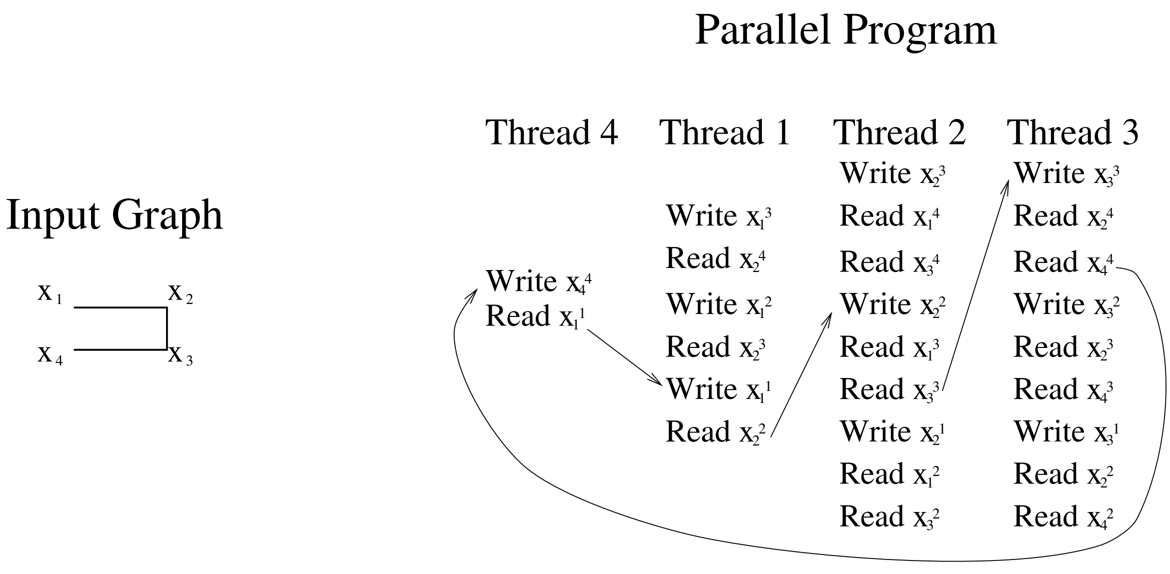


Figure A.1: Constructing a parallel program for a given graph.

The code for thread j, P_j, for $1 \leq j < n$ is defined as:

CodeSeq(j, n-1);

CodeSeq(j, n-2);

...

CodeSeq(j, 1);

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

We can show that a Hamiltonian path exists from v_1 to v_n if and only if there is a back-path between the two accesses initiated by thread P_n. Figure A.1 illustrates this construction for a simple graph consisting of four vertices. The figure shows a Hamiltonian path from vertex x_1 to vertex x_4 in the graph and a corresponding back-path between the two accesses initiated by P_4.

Theorem 5 *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, then a back-path exists. 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, ..., *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 path is a valid back-path from access *Read* v_1^1 to *Write* v_n^n.

We can also show that if a back-path exists between the two accesses, there is a Hamiltonian path in G that begins at vertex v_1 and ends at vertex v_n. To prove this assertion, we make use of the following properties of such a back-path.

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 a 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 between accesses initiated by thread n has the above mentioned properties. These properties also imply that there is 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.

To solve the HPR problem on a graph, we can use the reduction specified in this section and formulate an equivalent BPR problem. Hence, BPR is NP hard. □

A.2 Computing Smallest D_{synch} is NP Hard

We will now show that finding the smallest delay set for a parallel program with precedence information is NP hard. In particular, we will show than any instance of the Dominating Set problem (DS) can be reduced to an instance of computing a minimum cardinality delay set. The Dominating Set problem requires identifying the smallest subset of vertices of an undirected graph (V, E) such that every vertex in the graph is adjacent to

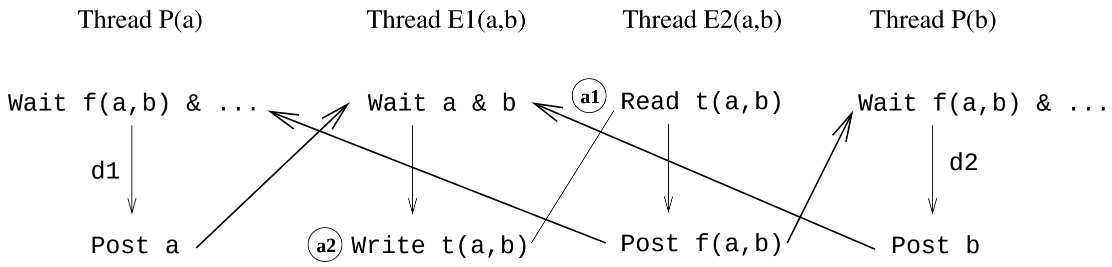


Figure A.2: Code fragments corresponding to two adjacent vertices.

some vertex in the dominating set. Given an undirected graph G, we construct a parallel program that requires a delay set D_{synch} of cardinality k only if a dominating set of size k can be found for G.

Let there be n vertices in V and m edges in E. We construct a parallel program with $n + 2 * m$ threads where each thread corresponds to a vertex or an edge in the input graph. These threads access shared variables $v_1, \ldots, v_n$ (named after the vertices in G) as well as $t(e_1), \ldots, t(e_m), f(e_1), \ldots, f(e_m)$ where $e_1, \ldots, e_m$ are edges in G. To simplify presentation, we assume that the language allows a $Post - Wait$ synchronization where the $Wait$ operation can wait for a set of condition variables simultaneously.

The thread $P(v)$ issues only two accesses: $Waitf(e_1), f(e_2), \ldots, f(e_k)$ and $Postv$, where $e_1, \ldots, e_k$ are the edges incident on v. The threads $E1(u,v)$ and $E2(u,v)$ correspond to an edge (a,b). $E1(u,v)$ issues a $Waitu, v$ operation followed by a $Writet(u,v)$. $E2(u,v)$ issues a $Readt(u,v)$ followed by a $Postf(u,v)$ operation. Figure A.2 illustrates this construction.

Theorem 6 *If there exists a delay set of size $k + 2 * m$ that is sufficient to avoid violations of sequential consistency, then G has a dominating set of size k.*

Proof: Consider the threads $P(a), P(b), E1(a,b)$, and $E2(a,b)$ for two vertices a, b that are adjacent in G (see Figure A.2). We clearly need to impose delay restrictions on threads $E1(a,b)$, and $E2(a,b)$ to prevent cycles from manifesting in the execution order. However, we observe that the threads $P(a)$ and $P(b)$ do not both require delay restrictions. If $d1$ is included in the delay set, it sets up a precedence relationship ($a1$ precedes $a2$) that makes it unnecessary to include $d2$ in the delay set. Similarly, if $d2$ is included in the delay set, $d1$ does not have to be included. In general, there are no delay restrictions imposed on thread $P(u)$ if and only if there is a vertex v adjacent to u and thread $P(v)$'s program edges are included in the delay set. Therefore, given a delay set that is sufficient for correct execution and includes the program edges from threads $P(u_1), \ldots, P(u_k)$, the set $(u_1, \ldots, u_k)$ is a dominating set in G. $\square$

Bibliography

[1] S. V. Adve and M. D. Hill. Weak Ordering–A New Definition. In *17th International Symposium on Computer Architecture*, April 1990.

[2] A. Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The mit alewife machine: Architecture and performance. In *International Symposium on Computer Architecture*, 1995.

[3] A. Aiken and D. Gay. Barrier Inference. In *Principles of Programming Languages*, 1998.

[4] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2), 1996.

[5] T. E. Anderson, D. E. Culler, and D. A. Patterson. A Case for NOW(Network of Workstations). *IEEE Micro*, 15(1), February 1995.

[6] B. S. Ang, Arvind, and D. Chiou. StarT the Next Generation: Integrating Global Caches and Dataflow Architecture. In *ISCA 1992 Dataflow Workshop*, 1992.

[7] R. Arpaci, D. Culler, A. Krishnamurthy, S. Steinberg, and K. Yelick. Empirical Evaluation of the CRAY-T3D: A Compiler Perspective. In *International Symposium on Computer Architecture*, June 1995.

[8] E. Barton, J. Cownie, and M. McLaren. Message passing on the Meiko CS-2. *Parallel Computing*, 20(4), Apr. 1994.

[9] BBN Advanced Computers Inc. *TC2000 Technical Product Summary*, 1989.

[10] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Fifteenth ACM Symposium on Operating System Principles*, 1995.

[11] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. In *IEEE CompCon Conference*, 1993.

[12] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. Dag-Consistent Distributed Shared Memory. In *10th International Parallel Processing Symposium*, 1996.

[13] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1), February 1995.

[14] F. Bodin, P. Beckman, D. B. Gannon, S. Narayana, and S. Yang. Distributed pC++: Basic ideas for an object parallel language. In *Proc. Supercomputing '91*, pages 273–282, 1991.

[15] E. Brooks. PCP: A Parallel Extension of C that is 99% Fat Free. Technical Report UCRL-99673, LLNL, 1988.

[16] D. Callahan, K. Kennedy, and J. Subhlok. Analysis of event synchronization in a parallel programming tool. volume 2, pages 151–169, Oct. 1990.

[17] M. C. Carlisle, A. Rogers, J. H. Reppy, and L. J. Hendren. Early experiences with Olden (parallel programming). In *Languages and Compilers for Parallel Computing. 6th International Workshop Proceedings*. Springer-Verlag, 1994.

[18] W. W. Carlson and J. M. Draper. Distributed Data Access in AC. In *ACM Symposium on Principles and Practice of Parallel Programming*, July 1995.

[19] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, 7(4), November 1989.

[20] K. L. Clarke and S. Gregory. PARLOG: Parallel programming in logic. *ACM Trans. Prog. Lang. Syst.*, 8(1):1–49, Jan. 1985.

[21] Cray Research Incorporated. *The CRAY T3D Hardware Reference Manual*, 1993.

[22] Cray Research Incorporated. *CRAY T3E Hardware Reference Manual*, 1996.

[23] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Sumbramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of the 1993 Conference on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.

[24] D. Culler, L. T. Liu, R. P. Martin, and C. Yoshikawa. LogP Performance Assessment of Fast Network Interfaces. *IEEE Micro*, February 1996.

[25] 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*, Portland, Oregon, Nov. 1993.

[26] Digital Equipment Corporation. *DECchip 21064-AA Microprocessor Hardware Reference Manual*, 1992.

[27] Digital Equipment Corporation. *Alpha 21164 Microprocessor Hardware Reference Manual*, 1996.

[28] M. P. I. Forum. Document for a standard message-passing interface. Technical report, Nov. 1993.

[29] 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*, 1990.

[30] W. Groscup. The Intel Paragon XP/S Supercomputer. In *Proceedings of the Fifth ECMWF Workshop on the Use of Parallel Processors in Meteorology.*, Nov 1992.

[31] High Performance Fortran Forum. High Performance Fortran language specification version 1.0. Draft, Jan. 1993. Also available as technical report CRPC-TR 92225, Center for Research on Parallel Computation, Rice University.

[32] 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.

[33] L. Iftode, J. Singh, and K. Li. Scope Consistency: A Bridge Between Release Consistency and Entry Consistency. In *Symposium on Parallel Algorithms and Architectures*, 1996.

[34] T. E. Jeremiassen and S. J. Eggers. Reducing False Sharing on Shared Memory Multiprocessors through Compile Time Data Transformations. In *ACM Symposium on Principles and Practice of Parallel Programming*, July 1995.

[35] Kendall Square Research. *KSR1 Technical Summary*, 1992.

[36] R. E. Kessler and J. L. Schwarzmeier. Cray T3D: a New Dimension for Cray Research. In *Digest of Papers. COMPCON Spring '93*, San Francisco, CA, Feb. 1993.

[37] R. K. Koeninger, M. Furtney, and M. Walker. A Shared-Memory MPP from Cray Research. *Digital Technical Journal*, 6(2), 1994.

[38] J. Kubiatowicz and A. Agarwal. Anatomy of a Message in the Alewife Multiprocessor. In *7th ACM International Conference on Supercomputing*, July 1993.

[39] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford Flash Multiprocessor. In *21st International Symposium on Computer Architecture*, April 1994.

[40] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9), September 1979.

[41] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. S. Pierre, D. S. Wells, M. C. Wong, S. Yang, and R. Zak. The Network Architecture of the CM-5. In *Symposium on Parallel and Distributed Algorithms '92*, June 1992.

[42] D. Lenoski, J. Laundon, K. Gharachorloo, A. Gupta, and J. L. Hennessy. The Directory Based Cache Coherance Protocol for the DASH Multiprocessor. In *Proceedings of the 17th International Symposium on Computer Architecture*, 1990.

[43] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, November 1989.

[44] L. T. Liu and D. E. Culler. Evaluation of the Intel Paragon on Active Message Communication. In *Intel Supercomputer Users Group Conference*, 1995.

[45] N. K. Madsen. Divergence Preserving Discrete Surface Integral Methods for Maxwell's Curl Equations Using Non-Orthogonal Unstructured Grids. Technical Report 92.04, RIACS, February 1992.

[46] S. P. Midkiff, D. Padua, and R. G. Cytron. Compiling Programs with User Parallelism. In *Languages and Compilers for Parallel Computing*, 1990.

[47] S. P. Midkiff and D. A. Padua. Issues in the optimization of parallel programs. In D. Padua, editor, *International Conference on Parallel Processing*, volume 2, pages 105–113. Penn. State Press, Aug. 1990.

[48] M. D. Noakes, D. A. Wallach, and W. J. Dally. The J-Machine Multicomputer: An Architectural Evaluation. In *20th International Symposium on Computer Architecture*, 1993.

[49] S. Pakin, V. Karamcheti, and A. Chien. Fast Messages (FM): Efficient, Portable Communication for Workstation Clusters with Supercomputing APIs and Performance. *IEEE Concurrency*, 5(2), 1997.

[50] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Typhoon and Tempest: User-Level Shared Memory. In *International Symposium on Computer Architecture*, April 1994.

[51] J. Rose and G. Steele Jr. C*: An Extended C Language for Data Parallel Programming. In *Proceedings of the Second International Conference on Supercomputing, Vol. 2*, pages 2–16, May 1987.

[52] R. H. Saavedra, R. S. Gaines, and M. J. Carlton. Micro Benchmark Analysis of the KSR1. In *Supercomputing '93*, Nov. 1993.

[53] R. H. Saavedra-Barrera. *CPU Performance Evaluation and Execution Time Prediction Using Narrow Spectrum Benchmarking*. PhD thesis, U.C. Berkeley, Computer Science Division, Feb. 1992.

[54] S. Savage, M. Burrows, G. Nelson, P. Solbavarro, and T. Anderson. Eraser: A Dynamic Race Detector for Multi-Threaded Programs. In *Sixteenth ACM Symposium on Operating System Principles*, 1997.

[55] K. E. Schauser and C. J. Scheiman. Experience with Active Messages on the Meiko CS-2. In *9th International Parallel Processing Symposium*, Apr. 1995.

[56] K. E. Schauser, C. J. Scheiman, J. M. Ferguson, and P. Z. Kolano. Exploiting the Capabilities of Communications Co-processors. In *10th International Parallel Processing Symposium*, Apr. 1996.

[57] S. L. Scott. Synchronization and Communication in the T3E Multiprocessor. In *Architectural Support for Programming Languages and Operating Systems*, 1996.

[58] 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.

[59] J. P. Singh, T. Joe, J. L. Hennessy, and A. Gupta. An Empirical Comparison of the Kendall Square Research KSR-1 and Stanford DASH Multiprocessors. In *Supercomputing '93*, Portland, Oregon, Nov. 1993.

[60] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. *Computer Architecture News*, 20(1), Mar. 1992.

[61] R. L. Sites. *Alpha Architecture Reference Manual.* Digital Equipment Corporation, 1992.

[62] S. K. Skedzelewski. Sisal. In B. K. Szymanski, editor, *Parallel Functional Languages and Compilers*, pages 105–157. 1991.

[63] T. von Eicken. *Active Messages: an Efficient Communication Architecture for Multiprocessors.* PhD thesis, U.C. Berkeley, Computer Science Division, Nov. 1993.

[64] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Fifteenth ACM Symposium on Operating System Principles*, Dec. 1995.

[65] 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.

[66] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient Software-Based Fault Isolation. In *Fourteenth ACM Symposium on Operating System Principles*, 1993.