LANGUAGE AND COMPILER MECHANISMS FOR PARALLEL PROGRAMMING WITH CUSTOMIZABLE PROTOCOLS

MUKUND RAGHAVACHARI

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

JANUARY 1998

Abstract

Customizing the protocols used to manage accesses to different data structures within an application can improve the performance of software shared-memory programs substantially. Existing systems for using customizable protocols, however, are hard to use because they force programmers to rely on low-level mechanisms for manipulating these protocols. Since several shared-memory systems implement protocols in software and can support customizable protocols, the development of intuitive abstractions for the use of application-specific protocols is important.

This dissertation is an in-depth study of the issues involved with providing language support for customizable protocols. We describe the design and implementation of a new language for parallel programming, $\mathcal{A}ce$, that integrates support for customizable protocols with minimal extensions to C. $\mathcal{A}ce$ applications are developed using a shared-memory model with a default sequentially consistent protocol. Performance can then be optimized, with minor modifications to the application, by experimenting with different protocol libraries.

The design of $\mathcal{A}ce$ was driven by a detailed examination of the use of customizable protocols. We delineate the issues that must be considered when programming with customizable protocols and present novel abstractions that allow for the easy use of customizable protocols. We describe the design and implementation of a runtime system and compiler for $\mathcal{A}ce$ and discuss compiler optimizations that improve the performance of such software shared-memory systems. We study the communication patterns of a set of benchmark applications, and consider the use of customizable protocols to optimize their performance. We evaluate the performance of our system using experimentation on a Thinking Machine CM-5 and a Cray T3E. We also present measurements that demonstrate that $\mathcal{A}ce$ has good performance compared to a modern software distributed shared-memory system.

Acknowledgements

One consequence of spending many moons in Princeton is that there are many friends, family, and colleagues whose support has been invaluable . . .

I thank my advisor, Anne Rogers, for her guidance during my graduate career at Princeton. Her attention to detail and keen perception of the key issues in a problem have had a lasting influence on my approach to research. I appreciate her encouragement of independent thought and her deep commitment to her students.

I thank my readers, Andrew Appel and Jaswinder Pal Singh, not only for their efforts in ensuring the clarity of my dissertation, but also for their comments and advice on many research issues over the years. Ed Felten's and Andrea LaPaugh's participation on my committee is appreciated. I learned much from my collaborations with Daniel Yellin at IBM and James Abello at AT & T Research. Kai Li and David Hanson gave valuable advice readily during my graduate career. I also thank the staff in the Computer Science department for answering my questions, and for their efforts in ensuring a smooth experience for the graduate students at Princeton.

I would like to thank the Wisconsin Wind Tunnel (WWT) project, led by Mark Hill, Jim Larus, and David Wood, for providing access to the CM-5 at the University of Wisconsin-Madison. I would also like to thank the National Center for Supercomputing Applications at the University of Illinois (Urbana-Champaign) for the use of their CM-5, the Pittsburgh Supercomputing Center for the use of their Cray T3E, and the Maui High Performance Computing Center for their SP-2. I was supported, in part, by ARPA grant N00014-91-J-4039 and NSF grant MIP-9420653, and a Princeton University Research Board Tuition Award.

While the professional support of many individuals were invaluable in my graduate career, my primary influence and motivating force has been my family. My father has always served as my role model — his love of research and scientific discovery were instilled in me at an early age. My mother's caring and warmth has refreshed me in many ways. My sister, Vindya, has offered constant encouragement and been a good confidant, as has my brother-in-law, Sanjay. My darling niece, Nitya, has always had a hug for her uncle. My grandparents, aunts, uncles, cousins — though India may be far away, they are never distant.

And finally, I must thank the friends that made my stay at Princeton so pleasant. Karthik has been a valuable friend in many a circumstance for many years. His fiancee, Vidya, is always warm, open, and cheerful. I wish them happiness in their future together. Rakesh and Barna provided me with food, shelter, and friendship, for which I am extremely grateful. And I thank the other members of the Indian mafia at Princeton for their friendship — Amar, Anant, Anil, Aparna, Elora, Karthik, Jayashree, Sandeep, Sanjeev, Subash, Shalu, Tipu, and Vishy. Robert Capaldi, Robert Nekrich and I have been to many homecomings and reunions, rooting for Penn at Penn-Princeton games (and the Flyers, of course, *Stanley Cup Champions '98*), and losing money in Atlantic City. Thanks should also go to my other Penn buddies, Suresh and Varun.

Chris "Boo-Boo" Dunworth was responsible for enlivening not only my spirits, but that of the entire department. With "Office-Ball" contests, coffee and ice-cream trips, Infinite Hoops basketball, Cache Hitters championships, screensavers, the department was never a dull place. His sense of humor and *joie de vivre* (that means stench, Chris) made coming to "work" enjoyable. Stef "Yogi Bear" Damianakis took charge of organizing the many activities that made graduate students feel like they had a life, and was always the epitome of Greek hospitality. Pritpal "YMCA" Ahuja won too many office-ball and card games, but it was fun wasting time with him and Jeff "Jasmine" Korn. I especially remember playing cards with them and Jovanna on our Montreal and Chicago trips. Others whose friendship I cherish are my officemates over the years: Norman, Norbert, Cheng, Bin, Wagner, Ben.

A few short paragraphs can never be adequate in describing the influence that each family member, each friend, each colleague has had on me over the years. I am deeply indebted to them for their support, and hope to continue our friendships in the years to come.

On to the dissertation …

To my parents

Contents

Chapter 1

Introduction

The expectations held out for parallel computing at the onset of this decade have remained largely unfulfilled. It has become evident that without suitable environments for application development, parallel computing shall remain the domain of the few programmers that have mastered the arcane art of parallel programming. Parallel machines, unlike their sequential counterparts, offer neither an accepted execution model for algorithm development nor intuitive programming languages, debuggers, or profilers. Investment in parallel computing requires the justification of the high monetary cost of parallel machines, as well as, the high cost of application development. The lack of a unified parallel programming model and the rapid rate of advance in parallel architectures render programmers wary of expending effort on applications for machines that soon become obsolete.

Yet the prospects for a portable, efficient programming language for parallel computing appear bright. This optimism is a result of the convergence of parallel architectures towards a set of workstation-like nodes, featuring commodity processors, and connected by a commodity interconnect [24]. The similarities in these architectures has made it possible to design clean and efficient interfaces that are portable across several systems. Examples include environments such as the Parallel Virtual Machine (PVM) [37] and the Message Passing Interface (MPI) [30], and data-parallel languages such as High Performance Fortran (HPF) [41] and Split-C [25]. These systems, however, have drawbacks which preclude their use for general purpose parallel programming. PVM and MPI support message passing, a difficult communication model for programmers to use, especially for applications with irregular communication patterns, such as, EM3D [25] and Barnes-Hut [11]. HPF and Split-C support a shared-memory abstraction, but are designed for a specific class of problems — data parallel applications.

Shared memory, supported on architectures such as FLASH [53], SHRIMP [14], Typhoon [76], and the SGI Origin 2000 [57], is popular as a programming abstraction because the underlying system assumes all responsibility for communication. While languages based on shared memory are convenient to use, the lack of control over the communication mechanisms can affect the performance of applications significantly. Tailoring the communication mechanisms to the access pattern of an application can improve its performance [12, 19, 34, 67]. The lack of control over communication plays an especially significant role in systems implemented on platforms without hardware support for shared memory, such as shared virtual memory systems [62].

The design of a general, intuitive and efficient programming environment for parallel application development is essential for parallel computing to achieve widespread acceptance. Since the mechanisms required for easy parallel programming are not yet well understood, experience with parallel applications and architectures must be evaluated systematically to develop an understanding of the design space for parallel languages. A methodical division of the design space into independent components can help designers understand the relative strengths of current languages and highlight constructs that require improvement. We believe that an incremental approach, where each component of the design space is attacked and understood independently, is an effective approach to attaining the goal of a general, easy-to-use, and efficient parallel language.

In this dissertation, we describe the design and implementation of $\mathcal{A}ce$, a new language for parallel computing. The constructs introduced in this language address one component of the design space of parallel languages — the interface to communication provided to the application programmer. Ensuring that communication is implemented correctly and efficiently is one of the most complex tasks of a parallel programmer. In modern architectures such as the Cray T3E [86], FLASH [53], and SHRIMP [14], the uniformity in processors and networks implies that the distinguishing characteristic of these architectures is usually their processor/network interface; this component often determines the communication primitives and parameters of the machine. This factor as well as the increasing discrepancy between processor speeds and communication times imply that communication plays an important role in determining the efficiency and portability of parallel programs. Current parallel systems either force programmers to manage all communication explicitly [30, 37], or provide them with little control over its implementation [22, 41, 78]. The strength of *Ace* is that it allows programmers to choose, using high-level abstractions, the level of control they require over the management of communication.

Ace allows programmers to develop applications using a shared-memory abstraction, and then optimize their performance easily by customizing the communication protocols used for accesses to shared data. While the use of specialized protocols to improve the efficiency of applications has been studied previously [12, 34, 67], current systems do not provide any language support for their use. Programmers are forced to use *ad hoc* and low-level mechanisms to integrate customized protocols into applications. The main contribution of this dissertation is an in-depth study of the issues involved in the design and implementation of a system that supports easy use of customizable protocols through language support.

In the rest of this chapter, we introduce the issues related to the design and implementation of a language that provides support for customizable protocols. Readers unfamiliar with certain concepts of parallel computing, such as consistency models and protocols, are encouraged to read Section 1.1 for background and pointers to references. In Section 1.2, we motivate the design of *Ace*, and discuss its relevance in the context of modern parallel languages. Section 1.3 lists the contributions of this dissertation. Finally, in Section 1.4, we summarize the chapter and describe the layout of the dissertation. This dissertation expands on our earlier work on programming abstractions for parallel programming [72, 74]

1.1 Background

In this section, we review certain issues that are referred to often in this document. First, we describe briefly the communication models found in parallel systems. We then discuss two key concepts in parallel programming — protocols and consistency models. For a primer on consistency models and protocols, please consult Adve *et al.* [1]. Readers interested in more detail on parallel architectures and systems are referred to Culler *et al.* [26].

Parallel systems can be classified on the basis of their communication model as supporting shared memory or message passing. In shared-memory systems, programmers visualize all memory in the system as part of a single, unified address space. Though, physically, the memory may be distributed across the processors of the machine, the underlying system maps the global shared address space viewed by the programmer to the actual physical memory locations transparently. When a processor accesses data that resides in a remote processor's memory, the transportation of data is performed automatically by the system without intervention from the programmer.

The mechanisms necessary for supporting a globally shared address space may be implemented in either hardware or software. Architectures such as FLASH [53], SHRIMP [14], Typhoon [76], and the SGI Origin 2000 [57] are examples of machines with hardware support for shared memory. Virtual memory management support can also be used to implement shared memory, as in systems such as Ivy [62], Munin [15], and Treadmarks [6]. Alternatively, shared memory may be implemented completely in software, as in systems like Blizzard [85], CRL [47], and Shasta [83]. A system that supports a global shared address space on a physically distributed-memory machine, is called a *distributed shared-memory system*, or DSM.

In message-passing systems, the distributed nature of the underlying architecture is explicit to the programmer. Each processor may access only data in its memory. Communication between processors is accomplished by passing messages between processors. Examples of message-passing architectures include the CM-5 [59] and the IBM SP-2 [44]. Though message-passing is usually a more difficult communication model to program, it is popular because message-passing machines can be built quickly and cheaply by connecting a set of workstations with a network. In the past, each message-passing machine typically supplied a proprietary library of message-passing routines. Recently, however, there has been a drive to standardize the set of message-passing calls offered by systems, which has resulted in interfaces such as MPI [30] and PVM [37].

Active Messages [95], a message-passing model that has been implemented on systems such as the CM-5 and Cray T3D [51], offers a different flavor of message passing. Communication between processors in message-passing systems typically requires that both the sending and receiving processors call message-passing routines explicitly. The sender invokes a routine that informs the system the data that are to be communicated, and the receiver calls a routine to assert that it is ready to receive data. In contrast, Active Messages is a one-way message-passing scheme. The sender calls a routine that, along with the data, specifies the function on the remote processor that is to handle the receipt of the data. When the message reaches its destination, the specified function handler is executed independently of the main thread of the computation on that processor. In general, either a processor is interrupted when a message is received, or the main thread of computation at a processor polls the network explicitly for messages. It is the handler's responsibility to integrate the data received into the computation at that processor. Active Messages are useful in circumstances where the communication is irregular and it is difficult to determine, *a priori*, where to insert matching send and receive calls in a program.

Protocols and Consistency Models

In shared-memory systems, when a processor accesses an address that is physically resident at another processor, the underlying system performs the requisite communication operations automatically. The component of a shared-memory system that handles accesses to global addresses, communicating when necessary, is known as the *protocol*. Conceptually, shared-memory systems provide a unified address space similar to that in sequential architectures. Unlike sequential programming, however, multiple processors may attempt to access a data object at the same instant. In the presence of simultaneous reads and writes to an object, the results of execution depend on the algorithm used by the protocol to mediate among these accesses. For example, a protocol may employ a central algorithm where one processor receives all requests for data objects and permits only a single access to execute at a time. Alternatively, a protocol may replicate copies of data on processors and propagate updates of data lazily to processors. To ensure that an implementation of an algorithm has its intended purpose, a parallel programmer must understand the memory behavior, that is, the *consistency model*, of the underlying protocol.

Considerable effort has been expended in the past in developing consistency models that are intuitive to programmers and allow for efficient protocol implementation [13, 38, 40, 50, 55]. We now consider two common consistency models — sequential consistency [55] and

Processor 0		Processor 1	
	$\{a = 0, b = 0\}$		$\{a = 0, b = 0\}$
a = 1	$\{a = 1, b = 0\}$	b = 1	$\{a = 0, b = 1\}$
lock = b	$\{a = 1, b = 0\}$	lock = a	$\{a = 0, b = 1\}$
if (lock == 0)	$\{a = 1, b = 0, lock = 0\}$	**if** (lock == 0)	$\{a = 0, b = 1, lock = 0\}$
Enter critical section		Enter critical section	

Figure 1.1: Example of an execution of a program that is not sequentially consistent. Italicized values in braces show the current state of memory at each processor.

release consistency [38] — in greater detail. Sequential consistency is generally considered the most intuitive protocol for parallel programming because the behavior of memory under this model is closely related to that in sequential programming. Under this consistency model, the behavior of memory is as if the accesses executed on processors were interleaved in some order and executed on a sequential machine.

The utility of sequential consistency is understood best by considering the execution of a program using a protocol that is not sequentially consistent. For example, consider the execution in Figure 1.1. Assume that in the example, the system's protocol propagates updates to variables to all processors, but that an update may take an unspecified amount of time to reach any processor. In the execution in the example, when Processor 0 reads b and Processor 1 reads a, the updates from Processor 1 and Processor 0 respectively have not arrived. Both processors obtain the value zero from the read, and therefore, enter the critical section, which is not the intent of the program.

Programmers, conditioned by experience on sequential machines, expect that on a write to memory, the next access temporally to that location would view the new value. For the program in the example, a programmer would assume that if Processor 0 saw a value of zero in b, then Processor 1 had not executed its write operation yet. Continuing this train of thought, the write to a on Processor 0 executed before the write on Processor 1, and therefore, before the read on Processor 1. Therefore, the read on Processor 1 should view the result of the write to a on Processor 0. An assertion that follows is that if Processor 0 read a value of zero in b, then Processor 1 would read one in a. A similar reasoning could be used to determine that if Processor 1 read zero in a, then Processor 0 would read one in b. Finally, the programmer would assert that either the read to a would return zero or the read to b, but not both, and therefore, only one of the processors could enter the critical section.

Sequential consistency supports the natural intuition of programmers that accesses to memory are performed "one at a time." The consistency model employed by the non-sequentially consistent system in the example requires more complex reasoning. The system offers no guarantee to a programmer as to when any update will be visible at a processor. A programmer must, therefore, ensure that an application handles cases in which the data will have arrived and when they will have not. The process of making applications conform to non-intuitive consistency models is time-consuming and error prone. Errors in consistency are often subtle and hard to detect. As a result, most systems support either sequential consistency or release consistency, discussed below.

The semantics of sequential consistency place severe limitations on the implementation of protocols that support it. To provide the illusion that each access to memory is visible instantaneously at each processor, generally, a protocol must block on each write to ensure that the next access to the location written will receive the new value. Release consistency relaxes the semantics of sequential consistency to allow for optimizations that provide better performance.

The release consistency model is motivated by the observation that in parallel applications, most *conflicting accesses*, that is, reads and writes to the same data location, are separated by synchronization. If a programmer could assert that an application exhibits this behavior, then a protocol need not ensure that a write is visible instantaneously; there will be a synchronization event on the processor that updates the data before the next access to that data from any processor. The protocol, therefore, must only guarantee that the effect of an access at a processor is made visible globally before the next synchronization point at that processor. This additional flexibility is instrumental in obtaining better performance on memory accesses. While release consistent systems are capable of higher efficiency, they place a larger burden on programmers. To obtain a sequentially consistent execution, programmers must place synchronization appropriately; otherwise, the programmer must reason about the execution of the program using the more complicated semantics of release consistency [38].

In sequentially consistent and release consistent systems, protocols generally cache copies of data at processors to avoid excessive communication. When data is updated, the protocols require some mechanism for informing processors that their copy of the data is stale. *Invalidation-based* protocols send messages to processors asking them to invalidate their copies. When a processor, subsequently, wishes to read the data, it requests a new copy from the protocol. In contrast, *update-based* protocols forward the newly updated data to all processors that currently have copies so that they may have the latest version. Neither of the two mechanisms is consistently better than the other; each is suited to certain communication patterns. An example in the next section examines this issue more thoroughly. Central to many implementations of invalidation and update-based protocols is the concept of *home*. In many protocols, the home processor of a data object, that is, the processor that allocates it, is responsible for ensuring the consistency of that data.

1.2 Motivation

A systematic framework for understanding parallel language abstractions and constructs is invaluable in evaluating the relative strengths of modern parallel languages. Our framework, described in greater detail in the related work subsection below, was introduced in an earlier study [72]. In that study, we examined the implementations of a set of benchmarks to understand the language abstractions necessary to implement them easily and efficiently. The result of our experience was a general philosophy for parallel languages — it is important that in the design of constructs, parallel languages provide good default behavior, and yet, allow enough flexibility that programmers may alter this behavior easily.[1] The meaning of this statement is illustrated best through an example that will also serve to motivate the design of $\mathcal{A}ce$.

Our example (Figure 1.2) is a skeletal version of a parallel application EM3D [25] (this benchmark is described in greater detail in Section 6.5.3).[2] The access pattern of this algorithm follows a *producer-consumer* relationship, where one processor is responsible for creating new values of a data object that is consumed by another. In the example, there

[1] Variations on this philosophy have been expressed by other researchers [16, 20].
[2] The example is derived from a description in Falsafi *et al.* [34].

are two globally shared integers, a and b. Two processors, x and y, are randomly chosen from the set of processors, and x and y are assigned the duty of updating the locations a and b, respectively. In each iteration of the algorithm, x first updates a using b, after which, a new value for b is calculated using a. These updates are synchronized using barriers.

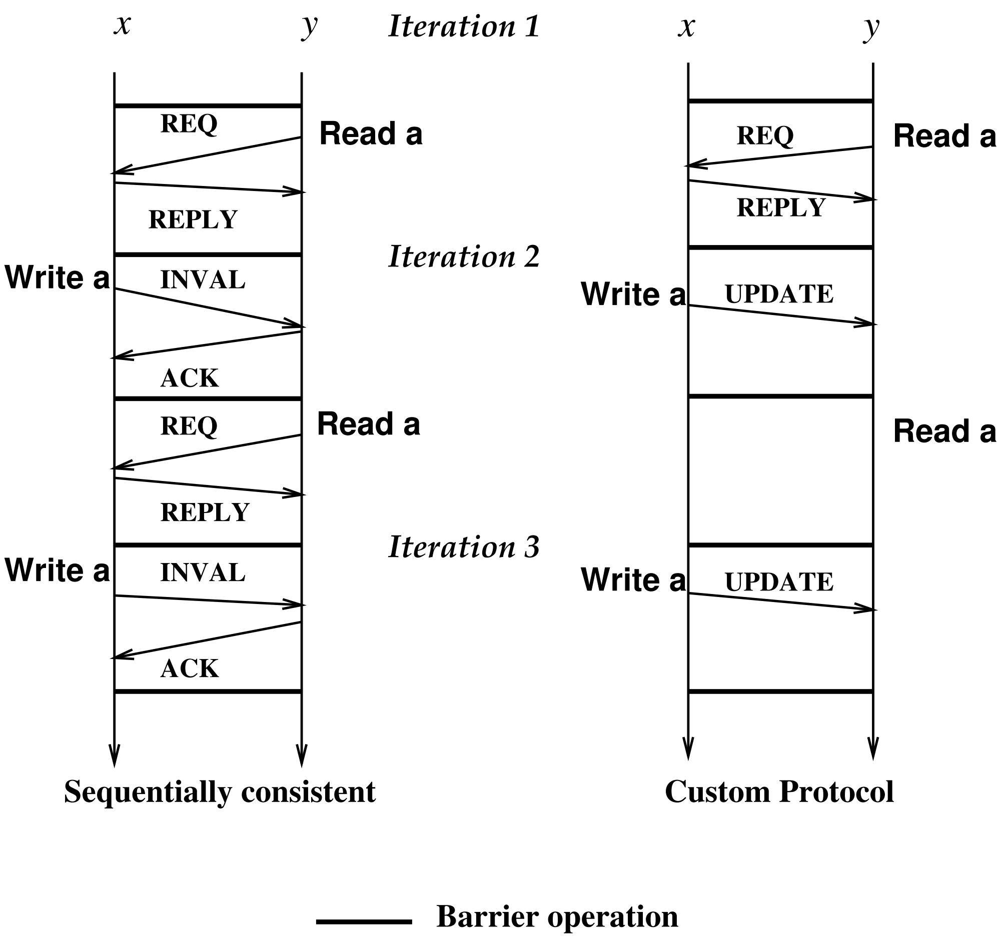
Consider the execution of this algorithm on a shared-memory platform that supports a sequentially consistent invalidation-based protocol. We focus on the protocol actions necessary to handle accesses to a. A graphical representation of the protocol events is shown in Figure 1.2b. We assume that at the onset of the algorithm, Processor x has exclusive write permission to a. When, in the second part of the first iteration, Processor y requires the updated value of a, it must request a copy from x, and block until the reply arrives. In the subsequent iteration, when Processor x updates a again, according to the rules of the protocol, it must first send invalidation messages to all processors holding outstanding copies, and wait for acknowledgments before proceeding. This process results in four messages being sent between the processors for accesses to a in each iteration.

Semantic information about the structure of the algorithm can be used to reduce the cost of this communication. While the identities of processors x and y are not known during the development of the program, it is evident that the access pattern is constant during the application. A more efficient protocol for this application is as follows. The first iteration is handled as before, Processor y requests a copy of a when it updates b. When Processor x receives this request, however, it records the information that Processor y made this request. On subsequent writes to a, Processor x does not invalidate the outstanding copy, but instead, sends an updated copy of the new value of a to y. To ensure that Processor y accesses an up-to-date version of a, the protocol guarantees that Processor y receives the new value of a before the barrier between the two operations is completed. With this protocol, after the first iteration, only one message is required per iteration for accesses to a.

Previous studies have shown that on software shared-memory systems, protocol optimizations such as the reduction in the number of messages sent above, can improve the performance of applications substantially [12, 34, 67, 74]. In addition to reducing the burden on the communication network, these optimizations reduce the amount of time that a processor spends blocked waiting for messages to arrive. Furthermore, protocols with relaxed consistency semantics allow compilers more flexibility in optimizing programs (see

```
x and y are distinct random processors
for i = 1... to number of time steps
    if I am x, a = b + 4;
    barrier();
    if I am y, b = a + 4;
    barrier();
endfor
```

(a)



(b)

Figure 1.2: (a) Pseudocode for example. (b) Graphical representation of protocol actions during the execution of (a).

Section 5.3). The advantage of the update protocol in the example above is that, in general, the *producer-consumer* access pattern is not well matched to invalidation-based protocols; if one knows the consumer of a data object, it is more efficient to send an update than to invalidate the consumer and force it to request a new copy explicitly. Due to the diverse nature of applications, however, it is difficult to conceive of a protocol that is well suited to all situations. While, in our example, an update protocol is more appropriate than an invalidation-based protocol, there are several cases, for example, migratory protocols, where the converse is true [27].

Consider now our example in the context of our statement that parallel languages should provide programmers with good defaults, but permit easy modification of the default behavior. With respect to protocols, during application development, it is likely that programmers would prefer a shared-memory model where a default protocol manages communication transparently using a well understood consistency model. Once a program is built and tested, a programmer may wish to modify the behavior of the default protocol so that accesses to data structures are handled by the most appropriate protocol for that data structure. The mechanism for changing the protocol of a data structure should be intuitive and easy to use.

In such a programming model, protocols for common access patterns, such as the producer-consumer mentioned above, would be available as libraries, and programmers could specify their use with high-level language constructs. Developers unwilling to reason about complex protocols could be content with the default protocol, while advanced programmers would have the flexibility of making their applications more efficient by choosing appropriate protocols from existing libraries, or by developing new protocols customized to the programs's access patterns. The subject of this dissertation is the design and implementation of a language, $\mathcal{A}ce$, that supports this model of application development, that is, *programming with customizable protocols*, efficiently and intuitively.

For example, in the producer-consumer application above, a programmer would first write the program assuming a default shared-memory model. Once the application is developed and tested, the programmer would use high-level language constructs to choose a protocol from a set of libraries that is most appropriate to the access pattern of the algorithm. In more complex applications, different data structures would probably require different protocols. Furthermore, as the execution passes through various phases, the protocol associated with a data structure might change as well [12]. If a desired protocol is not available in a library, the system should allow the programmer to add new protocols easily.

Related Work

It is instructive to consider the position that *Ace* occupies in the spectrum of modern parallel languages. We use a systematic framework, summarized below, for evaluating the strengths of parallel languages (for details on the framework, please refer to our earlier study [72]). Our framework divides the design space of parallel languages into the data model and the task model. The data model includes all issues associated with the access of data and can be further subdivided into the following: address space, data distribution and replication, and communication. The task model corresponds to constructs used by programmers to assign work to processors and includes the following components: thread model, task creation, task allocation, and synchronization. The issue of synchronization, though placed in the task model above, straddles both components. While synchronization is often used to control dependences between tasks, this control can be, and often is, implemented by synchronizing accesses to data. In this sense, synchronization can be considered part of the data model.

The *Ace* programming model is concerned with issues relevant to the data model component of the framework. Specifically, it is aimed at understanding the constructs and abstractions necessary for supporting our philosophy of good defaults with flexible control over the address space, data replication and locality, and communication. As such, we do not consider constructs associated with the task model. *Ace* is explicitly parallel, that is, we assume that there is a single thread per processor, and that task creation and allocation are handled explicitly by the programmer. In this model, synchronization should actually be considered part of the data model because task ordering is implemented by inserting appropriate synchronization between accesses to data structures. *Ace*, therefore, incorporates appropriate abstractions for synchronization as well.

The issue of address space revolves around whether a system provides a globally shared address space, or whether each processor may only access addresses in its own private memory. A globally shared address space is generally accepted to be more intuitive for application development because, though the underlying machine's memory may be distributed, the programmer can abstractly view the memory system as a single unified memory. This feature is especially useful in irregular applications, where, in its absence, programmers often simulate it by hand [87]. The utility of a shared address space is evident from the fact that most parallel languages except for message-passing systems support it.

On most platforms, however, accesses to local memory can be implemented more efficiently than accesses to remote memory. For example, on software shared-memory systems, local accesses can be handled without performing access checks. In these situations, it is useful to be able to partition data into private data and globally shared data so that the most efficient mechanisms are used in each case. For this component, therefore, a good default mechanism is a globally shared address space. In addition, the flexibility to partition data appropriately is helpful. Among parallel languages, those that are designed to support a shared-memory abstraction on message-passing architectures typically have both features, while languages built on shared-memory hardware generally only provide the default mechanism.

Data distribution is the ability to specify the layout of data structures across processors. Considerable effort has been expended in developing constructs for specifying the data distributions of arrays in languages such as HPF [41] and Fortran D [35]. For irregular applications, however, the appropriate partitioning of data is not yet well understood, much less the primitives required to specify these partitionings easily. We believe that the issue of data distribution is an important problem, the solution of which is orthogonal to the issue studied in this dissertation. In *Ace*, as in most explicitly parallel languages, we require that programmers manage the distribution of data manually.

The replication of data is necessitated by a reality of current technology — accesses to the memory of remote processors are more expensive than local accesses, often by more than an order of magnitude [51, 59]. As a result, data must be cached locally when possible to avoid the cost of remote accesses. In shared-memory systems, in the presence of replication, a protocol is necessary to manage the consistency of the various copies of data. In message-passing systems, programmers manage the consistency manually in an *ad hoc* fashion.

The main thrust of *Ace*'s mechanism is in attacking this issue. A drawback of modern shared-memory languages is the fact that they typically support only a single or a limited number of fixed protocols [13, 25, 62, 70, 47, 83, 84]. On certain platforms, the inability to customize the underlying protocols to the access patterns of applications can affect the application's performance substantially. Munin [16] provides a fixed set of protocols that programmers can associate with data structures, but provides no facility for the customization of protocols. There are many access patterns for which this set is insufficient. For example, several applications, such as EM3D [25], parallel multigrid applications [4], and CHARMM [43], have access patterns similar to that of the producer-consumer application in the previous section. Munin's set of protocols, however, does not include the update protocol that is most appropriate for this communication pattern, nor does it allow programmers to extend the system with new customized protocols.

At the other end of the spectrum are message-passing systems such as MPI [30] and PVM [37]. In these systems, the programmer assumes complete control of the management of communication. Programmers must be conscious of the role of communication at all stages in program development, and improving the implementation of communication incrementally is an error-prone and complex chore. There is no default communication protocol to facilitate the early stages of application development.

Tempest [76], a hybrid of shared-memory and message-passing systems, has some support for using application-specific protocols. Tempest provides *access fault control*, the ability for programmers to specify handlers that are invoked when an access to a cache block causes a fault in memory. This mechanism, along with message-passing routines, can be used to develop some customized protocols for applications. Tempest, however, does not provide any language abstractions for manipulating these protocols. It forces programmers to use low-level mechanisms such as explicit virtual memory management. Tempest's hybrid shared-memory/message-passing model, along with *access fault control*, hinder the creation of protocol libraries and complicate application development. Unlike Tempest, *Ace* allows programmers to manipulate and add protocols in a modular fashion with few changes to an application's code (Section 2.1 examines this issue in greater detail).

Parallel architectures, such as FLASH [53], Typhoon [76], and Alewife [3] offer some support for using both message passing and shared memory in an application. FLASH has hardware support for multiple protocols, where the system provides a fixed set of protocols. Typhoon has hardware support for user-level customizable protocols, but may suffer from protection problems since a user-level protocol may unfairly grab control of a machine's resources. No high-level language support for using these mechanisms is available as of yet. The problems and solutions discussed in this dissertation can be applied to these architectures as well. We discuss how *Ace* could be used on these architectures in Section 4.6.

When parallel architectures offer a range of communication mechanisms, such as bulk transfer or Active Messages [95], a programmer may possess information that is helpful in determining the appropriate mechanism to use for the transport of a data structure. In message-passing systems, since programmers insert communication calls explicitly, programmers have this ability. In most shared-memory systems, however, the communication mechanisms used are hard-wired into the protocol. Split-C and CRL, however, do offer some ability to control the mechanisms used by an application. In *Ace*, programmers can design their own protocols to add to the system. A conscious goal in the design of *Ace* was to allow for the use of architecture-specific mechanisms in protocols. *Ace* programmers have the flexibility of choosing the communication mechanisms best suited to a protocol or access pattern.

1.3 Contributions

The contributions of this dissertation are as follows:

Programming with customizable protocols: We perform a thorough examination of the issues related to supporting the programming with customizable protocols model efficiently (Chapter 2). We consider the form that a language for this model should take — how can protocols be introduced into the standard programming model so that application developers may manipulate them easily and efficiently? We also delve into the semantics of consistency models and programming with customizable protocols, and discuss issues related to profiling and debugging such programs.

The *Ace* language: We introduce a new language, *Ace*, that integrates support for customizable protocols with minimal extensions to C (Chapter 3). *Ace* programmers use high-level abstractions to select the protocols associated with different data structures. Typically, a programmer will develop an application using the standard sequentially consistent shared-memory model, with locks and barriers for synchronization. Then, using knowledge about the access patterns and consistency semantics of data structures, programmers can optimize performance by associating appropriate protocols with data structures. *Ace* has several novel features:

- *Complete separation of application and protocol code:* Current systems for customizable protocols present a hybrid shared-memory/message-passing programming model [76]. Since the communication code is often part of the application code, programmers must reason explicitly about the complicated interactions between the two portions. This error-prone process retards application development because the application and the protocols must be developed and tested as a monolithic unit. In addition, the lack of separation between application and communication code inhibits the creation of protocol libraries. In our experience, encapsulation of protocol code eases the use of customizable protocols considerably.

- *Language abstractions for associating protocols with data structures:* Current systems require programmers to associate protocols with pages and to manage virtual memory explicitly. We provide an intuitive abstraction, called *spaces*, for associating protocols with data structures.

- *User-specified granularity:* Current systems for customizable protocols set the granularity of coherence of data implicitly, based on multiples of a fixed cache-line size. Often, assertions made by a programmer about data are incorrect due to the presence of *false sharing of protocols* (Section 2.3). User-specified granularity makes the granularity of coherence intuitive to the programmer, and is also a means of taking advantage of a bulk transfer mechanism. User-specified granularity is similar to the coherence granularity in region-based systems such as CRL [47], but these systems do not have any support for customizable protocols.

- *Extensibility: Ace* offers a clean interface for adding new protocols to the system.

Implementing *Ace* on distributed-memory machines: We discuss the problems and solutions with implementing *Ace* on distributed-memory machines. We study the implementation of *Ace* on two different architectures: a message-passing machine, the Thinking Machines CM-5 [59], and a machine with hardware support for implementing a shared address space, but no hardware support for maintaining global coherence of data, the Cray T3E [86]. We describe, in detail, the design and implementation of the *Ace* runtime system (Chapter 4), and the *Ace* compiler (Chapter 5). Most of the responsibility for supporting the *Ace* programming model rests with the runtime system. The runtime system also provides routines that support the development and addition of protocols.

The compiler transforms *Ace* programs into code that can be linked with the runtime library to create executables for our platforms. The compiler represents a new method of implementing fine-grained shared memory on distributed-memory machines. Previous research has concentrated either on using virtual memory [13, 16, 62] or link-time object-code rewriting [83, 85] to implement shared memory. In our description of the compiler, we examine issues in compiling shared-memory programs and describe the design and implementation of optimizations that are essential for good performance.

A study of protocols: In Chapter 4, we describe the implementation of the protocols that constitute the *Ace* standard protocol library. These represent common access patterns that are found in many parallel applications. In Chapter 6, we study extensively, for a set of eight standard benchmarks, the protocols that are most appropriate for these applications. In several cases, we describe custom protocols that improve the performance of our benchmarks significantly.

Experimental validation: We perform a thorough evaluation of the *Ace* system to understand its performance (Chapter 6). Through a set of microbenchmarks, we reveal the costs of important *Ace* operations. We then study in detail, the performance of our set of benchmarks and custom protocols on a Thinking Machines CM-5 [59] and a Cray T3E [86]. We also compare *Ace*'s performance with that of CRL [47], an efficient software distributed shared-memory system. The speedups obtained with custom protocols for *Ace* applications range from 1.0 to 6.6 on the CM-5 and 1.0 to 8.2 on the T3E. Finally, we examine various aspects of the compiler's performance and isolate the effects of optimizations.

1.4 Summary

Shared memory's popularity as a parallel programming model is due, in part, to the fact that the complexity of communication is concealed from the programmer. This abstraction can simplify programming significantly, especially for problems with irregular access patterns. Many shared-memory systems suffer from decreased performance because they use a single or a small number of fixed communication mechanisms to share data; inefficiencies occur when there is a mismatch between a program's access pattern and the system's communication protocol. For example, producer-consumer style sharing is not well suited to an invalidation protocol.

This dissertation describes the design and implementation of the *Ace* language. *Ace* programmers develop applications easily by assuming a shared-memory model. Programmers can then use high-level language abstractions to optimize their applications by controlling the communication protocols used for accesses to data structures. Typically, programmers will associate with a data structure, a protocol from a standard library that is most appropriate for the data's access pattern. When the standard protocol library is insufficient, programmers have the ability to extend the system with new protocols easily.

We describe, in the following chapters, the design of *Ace* and the implementation of its compiler and runtime system. We also present experimental evidence to validate the performance of our system. We isolate the issues involved in providing language support and introduce novel language abstractions for the easy use of customizable protocols in Chapter 2. Chapter 3 presents the constructs of the *Ace* language. The structure of the implementation of *Ace* and the design of the runtime system is examined in detail in Chapter 4. This chapter also describes the implementation of a standard library of protocols that handles certain common access patterns. Chapter 5 analyzes the *Ace* compiler and its optimization phases. In Chapter 6, we provide detailed descriptions of the implementation of a set of eight benchmarks on *Ace* and present experimental results. Finally, in Chapter 7, we summarize our findings and discuss future avenues of research.

Chapter 2

Issues in the design of the *Ace* language

Language design is a process of balancing many, often contradictory, goals to arrive at a set of constructs that is both intuitive and efficient. For ease of programming, a language ought to minimize the amount of information that must be provided by a programmer that is extraneous to the algorithm implemented. The efficiency of a program, however, is often dependent on the level of detail to which a compiler can analyze a program. The fewer implementation-specific details available in a program, the harder it is for a compiler to obtain good performance. To this end, it is often necessary for a programmer to expend extra effort in augmenting a program with annotations such as the data distribution annotations in HPF [41]. Yet, at times, the use of implementation-specific techniques such as address arithmetic can hamper a compiler's ability to optimize a program.

The goal of the *Ace* language is to provide a set of constructs that allows programmers to manipulate customizable protocols easily and efficiently. The form of these abstractions was molded by extensive experience with programming on message-passing machines and on Tempest [76], a system with some support for application-specific protocols. The design philosophy underlying these constructs is that of allowing programmers maximum flexibility in controlling the implementation of communication through high-level abstractions, while providing efficient default-case behavior.

The intent of the chapter is to introduce some of the issues specific to the use of customizable protocols that arose during the design and implementation of *Ace* and to describe the rationale behind our solutions. In the next section, we begin with a discussion of the advantages of $\mathcal{A}ce$'s model of programming with customizable protocols, specifically, the benefits of separating the algorithmic and communication components of a program. Section 2.2 introduces a new abstraction called *space*, which forms the basis for the mechanism for associating protocols with data structures.

The problem of granularity of coherence is then studied in Section 2.3. We proceed to analyze the semantics of programming with customizable protocols (Section 2.4). Section 2.5 discusses the development of $\mathcal{A}ce$ programs — how they may be debugged and profiled. Finally, Section 2.6 summarizes the chapter, discusses the shortcomings of the language, and offers possible improvements.

2.1 Encapsulation of Communication Complexity

The aim of the $\mathcal{A}ce$ system is to allow the development of parallel applications using a default, well understood protocol, and then allow programmers the ability to optimize communication by selecting appropriate protocols for different data structures. The viability of this programming model hinges on the ability to select among different protocols with minimal changes to the structure of a program. The modularity of protocols implies the existence of a well defined interface that mediates the interactions between the algorithmic details of an application and the component that handles the communication.

In the presence of modular protocols, the process of communication is as depicted in Figure 2.1. For communication, the algorithmic portion of the code uses the interface's routines, which in turn, select the appropriate protocol code to execute. If a programmer deems that a customized protocol is appropriate, changing the protocol of a data structure involves no change to the structure of the algorithm. All that is required is a mechanism by which the algorithm informs the interface of the new protocol that is to be invoked when the algorithmic code calls interface routines on that data structure (a description of the construct for informing the interface of associations between data structures and protocols is provided in the next section).

If, on the other hand, the communication code were embedded in the algorithmic component of a program, changing the protocol of a data structure would be more difficult. A programmer would have to sift through the entire application, identify each segment that

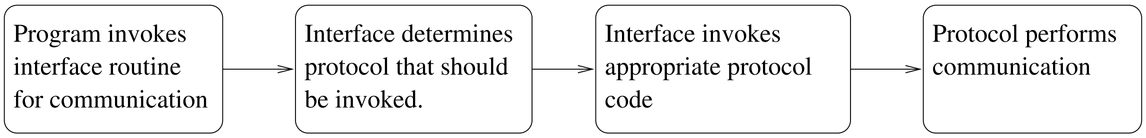


Figure 2.1: Communication process for systems with customizable protocols.

is associated with communication of that data structure, and then, rewrite it to fulfill the new protocol's semantics. This approach is error-prone and does not lend itself well to large-scale applications. The process is rendered more complex by the fact that the communication routines may use or modify the program's local context and thus influence the behavior of the program.

The advantages of separation of algorithm and communication code are many. An application can be developed and tested independently of communication code. Similarly, the communication code can be built and exercised separately using small test programs, and then, be united with the algorithmic code to obtain an efficient program. In addition, the encapsulation of communication code facilitates the creation of protocol libraries to handle common communication patterns, such as update protocols and migratory protocols, that can be reused by many applications.

The separation of protocol and application code enhances the portability of an application as well. Since algorithms are implemented in terms of an interface, once the interface and the underlying protocols are ported to a new architecture, the application should execute correctly as well. Embedding communication code into the application complicates the porting process for the same reason as that described above for the changing of protocols — treating applications as a single monolithic unit makes it harder to isolate and manage communication.

The encapsulation of communication code into protocol libraries does not restrict protocols from using architecture-specific communication mechanisms for optimal performance. For example, a previous study by Chandra *et al.* [19] has shown that on an architecture with some support for customizable protocols, FLASH [53], using two fixed protocols, hardware shared memory and shared virtual memory, appropriately can improve the performance of applications. A similar effect can be obtained in *Ace* by having one protocol use underlying hardware shared-memory support to implement communication, while another implements shared virtual memory. Unlike Chandra's system, however, such protocols are not built in, but are modular extensions to *Ace*.

An obvious choice for an interface between the algorithmic and communication code is one based on the conventional shared-memory interface. The program reads and writes remote data through load and store interface operations, which in turn, are dispatched to appropriate protocol routines that implement custom semantics for these operations. Care must be taken, however, when such an interface is designed. In order to maintain consistency according to its custom semantics, a protocol may require control at various points during the execution of a program. An interface must provide sufficient "hooks" to allow the implementation of most common protocols.

The diverse nature of communication within applications makes it difficult to predict when protocols may require control. For example, an invalidate protocol requires invalidations to be performed *before* writes, while a dynamic update protocol propagates writes automatically to sharers *after* the write. Other protocols, such as the data-race checking protocol proposed by Larus *et al.* [56], can be executed either before or after accesses. It is sometimes necessary to execute protocol actions at synchronization points. For example, at barriers, a protocol for split-phase memory operations, as offered by Split-C [25], must check that all outstanding memory operations have completed. Similarly, shared virtual memory systems based on relaxed consistency, for example, Treadmarks [6], perform protocol actions at synchronization points.

In our experience, providing control to protocols before and after accesses and at synchronization points is sufficient for most common protocols (we examine some limitations of this model in Section 2.4.3). *Ace*, therefore, supports *full access control*, that is, programs invoke interface routines at each of these points and protocols can specify actions that are to execute at these events. Previous systems, such as Tempest, support *access-fault control*, the ability to specify protocol actions that are executed when an access to data faults in memory. Several commonly used protocols cannot be implemented using access-fault control without either violating the encapsulation of protocols or circumventing the system in a complex and *ad hoc* manner.

For example, consider the implementation of a dynamic update protocol that requires writes to be propagated *after* each write. Using access-fault control, one might have the protocol declare all data to be invalid and fault on every write. The problem with this approach is that the protocol handlers are executed before the write occurs. After the write is performed, there is no way to return control to the protocol so that the write can be propagated. While a protocol designer may be able to simulate full access control by augmenting access fault control with hardware or software mechanisms explicitly, for example, by inserting message-passing calls at appropriate points in a program, the goal of an abstraction is to conceal implementation details. Full access control, unlike access-fault control, allows protocol designers to associate handlers at appropriate points. The actual *method* of invocation is transparent to the protocol designer.

It may appear that it is sufficient to support a small set of few fixed protocols in the system, from which programmers may select efficient ones for their data structures. We have found the contrary. Even if one could classify the entire set of access patterns into a small number of categories, it is not necessarily true that a single protocol could handle all access patterns in a category efficiently.

For example, in parallel applications, producer-consumer access patterns are common. Update protocols, where the producer updates the consumer's values of data, are generally the most appropriate for this access pattern. While systems such as Midway [13] and Munin handle this access pattern with a single fixed protocol, the best implementation (and semantics) of update protocols differs for each application. For example, for certain applications, the access patterns do not change after the first iteration of a loop. In such cases, a static update protocol is most efficient. In other cases, one may choose to implement updates using dynamic updates, pipelining, etc.

In general, we believe that it will be necessary at times for a programmer to modify existing protocols to suit an application's access patterns. A conscious goal in the design of 𝒜*ce*, therefore, is to provide a clean mechanism for adding new protocols to the system. While the difficult task of writing protocols must be borne by the protocol designer, systems such as Teapot [21], can be modified to be used with 𝒜*ce* to alleviate this burden.

2.2 Spaces

With the separation of communication code and algorithmic code, the utility of the *Ace* system becomes contingent on the ease with which protocols may be associated with data structures. There are several conceivable solutions to this problem. For example, a programmer could specify the protocol that is to manage each coherence unit, or, as in Tempest, protocols could be associated with data on a per page basis. The first approach requires a great deal of effort from the programmer and may have significant memory overhead as well, since the space of each coherence unit must be stored in the system. The second approach has less overhead, but requires the programmer to manage the low-level details of allocating data to pages explicitly.

To understand the appropriate abstraction for associating data and protocols, it is instructive to examine how protocols manipulate data. As the execution of a program passes through different phases, the access patterns of its data structures may change. It is sometimes necessary, for efficiency or correctness, to change the protocol associated with a data structure. For example, in Water [88], the program alternates between phases where intraprocessor and interprocessor calculations are made. We have found that shifting between a protocol that specifies null handlers for all events for the intraprocessor phase and an update protocol tailored to the communication pattern of the interprocessor phase has a speedup of 1.4 over a sequentially consistent execution (using *Ace* on the CM-5). Bennett *et al.* [12] have also found that data structure access patterns can change over the course of an application. As parallel applications become larger and more complex, we believe that the changing of access patterns will be more evident.

Protocols do not always operate on each logical datum independently, but often handle a collection of data. For example, a static update protocol (Section 4.5.3 describes this protocol in greater detail) builds a list of sharers for all data in a data structure, collects all writes to that data, and then, uses the sharer list to propagate updates at barriers. One may also require separate instances of the same protocol to operate on different data structures. For example, two different data structures may use the static update protocol. Though the access patterns of the data structures are abstractly similar, the sharer lists of each data structure may be different. In this case, one would require two separate copies of the static update protocol, one managing each data structure. As such, it is not sufficient to associate a space with a data type, since different instantiations of a type may have different access patterns.

In *Ace*, we introduce an intuitive abstraction, called *space*, that is designed to meet the requirements outlined above. Protocols are not associated with data directly, but indirectly through spaces. A space manages a subset of the address space, and handles all accesses and synchronization to data allocated from it. To associate a protocol with a data structure, a programmer would take the following steps:

1. Invoke an interface call that creates a space and associates a user-specified protocol with it.

2. Allocate all nodes of the data structure from the space. The system provides a default invalidation-based sequentially consistent space from which data may be allocated.

Figure 2.2 is a graphical example of the use of spaces. In the example, the application has two data structures, a tree and a linked list. The program creates a space for each data structure, tree_space and list_space, and associates the default protocol with them. The program then allocates all nodes of the tree from tree_space and all nodes of the linked list from list_space. The underlying system will route all subsequent accesses to these data to the default protocol automatically.

The space abstraction is an intuitive and powerful abstraction for associating data with protocols. In Figure 2.2, suppose the program wishes to change the protocol associated with the linked list data structure to a custom protocol. This can be implemented simply by having the program change the protocol associated with the space (depicted by the dotted line in Figure 2.2). If the protocols were associated with coherence units or with pages, a programmer would have to have some means of determining all the coherence units or pages that contain relevant data, and changing their protocols individually.

In addition, a space collects together all the nodes of a data structure so that a protocol may optimize accesses to them. Finally, spaces allow protocols to store information in them (depicted by the shaded box in the space structure in Figure 2.2). A protocol may operate on several spaces independently by storing space-specific state information in the

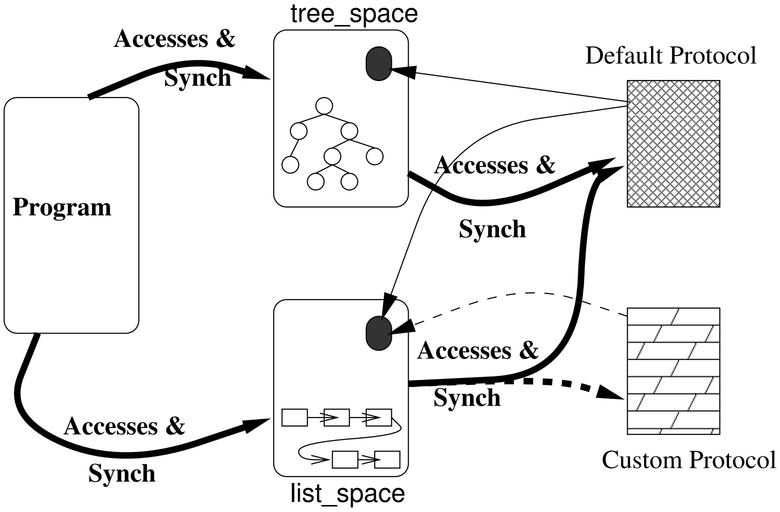


Figure 2.2: Graphical representation of the use of spaces. Thick lines show the path taken by accesses and synchronization through the system. Thin lines from protocol to spaces depict the storage of space-specific information in spaces. Dashed lines show the result of changing the protocol of list_space to the custom protocol.

respective spaces (as in the static update protocol mentioned above). From a programmer's point-of-view, spaces are intuitive abstractions; their use is similar to that of the memory arenas that form the basis of several allocators such as vmalloc [94], except that spaces possess greater semantics.

2.3 Granularity

The issue of granularity of coherence in a distributed shared-memory system is, in a sense, orthogonal to that of using customizable protocols. The problems and solutions mentioned above apply equally to systems of page or fine-grained granularity. Fixed-size granularity, however, complicates the use of customizable protocols. Assigning data to fixed-size coherence units implicitly forces programmers to worry about false sharing of data, and creates the problem of *false sharing of protocols*.

The problem of false sharing of data is common to several distributed shared-memory systems and has been studied extensively in the past [31, 33, 45, 93]. Essentially, the problem arises from a system allocating two or more logically distinct objects to the same

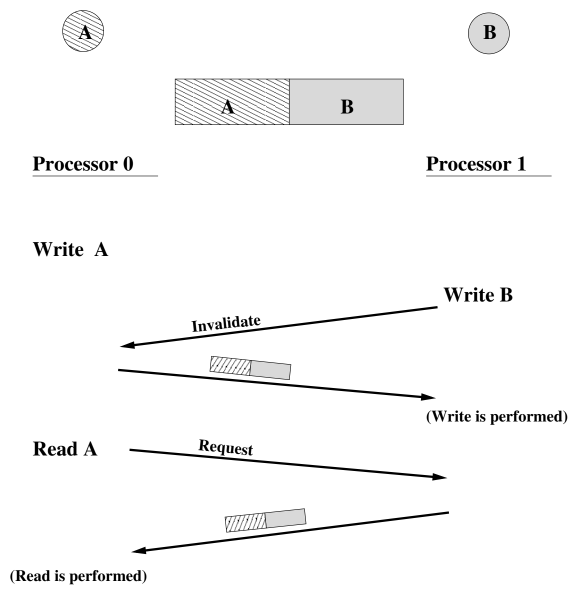


Figure 2.3: False sharing example. A and B are logically distinct objects, but placed in the same coherence unit. The underlying protocol is sequentially consistent, invalidation-based.

coherence unit. For example, consider the situation depicted in Figure 2.3. In the program, there are two logically distinct variables, A and B, which are assigned to the same coherence unit by the underlying system. Assume that a sequentially consistent invalidation-based protocol manages accesses to these variables, and that initially, Processor 0 has exclusive write permission for the coherence unit.

When Processor 1 attempts to write to B, according to the rules of the protocol, it must obtain write permission for the coherence unit. Processor 0's local copy is invalidated and exclusive write permission is granted to Processor 1. Subsequently, when Processor 0 attempts to read A, it must request an up-to-date copy of the coherence unit from Processor 1. Notice that the two processors do not share any data; the invalidation of Processor 0's copy and the subsequent request are artifacts of the implicit allocation by the underlying system. When such sharing is induced in a distributed shared-memory system, it is known as *false sharing*.

While false sharing of data is common to several systems, a more insidious problem, *false sharing of protocols*, asserts itself in the presence of customizable protocols. This problem can appear in two forms. First, data with different access patterns and managed by different protocols may reside in the same coherence unit. In this case, each protocol managing an object in the coherence unit, unaware of the others, might manipulate the coherence unit into a state that is inconsistent with the other protocols's semantics.

The space abstraction provides a solution to this problem. Since a space controls the allocation of objects within it, the runtime system can ensure that data from different spaces are not allocated in the same coherence unit. When a space requests memory from the underlying system, a block of memory rounded up to the nearest coherence unit boundary is returned. This mechanism guarantees that data managed by different protocols will not be located in the same coherence unit.

The second, more subtle, variation of false sharing of protocols can arise when a programmer selects a protocol for a data structure. A programmer may choose a particular protocol based upon knowledge about the access patterns of each individual object. This knowledge may be rendered false if multiple objects reside in a coherence unit. For example, a programmer may make the assertion that each datum is written only by a single processor. When many data (written by different processors) are located in a single coherence unit, a statement that is true of each of the constituent data, may be false for the coherence unit as a whole. Consider the example in Figure 2.3. Abstractly, A and B are each written only by a single processor. If, however, they are put into the same coherence unit, then the coherence unit, as a whole, is written by many processors.

Since false sharing of protocols may cause subtle inconsistencies during the execution of programs, *Ace* operates on variable-sized, user-specified coherence units. Specifically, when a programmer allocates memory for a data structure node, the entire node is considered an atomic coherence unit. In addition to removing the false sharing of protocols, user-specified granularity diminishes the problem of false sharing of data. Furthermore, having the ability to control the size of coherence units provides the programmer with a means of specifying when bulk transfer may be useful. For example, in certain applications, when the program accesses a part of a structure or array, the entire structure or array is typically accessed as well. In this case, a programmer can create a coherence unit consisting of the entire structure or array, and the underlying system will transport the coherence unit as a whole.

2.4 Semantics of Customizable Protocols

In practice, the association of efficient protocols with data structures in an application is intuitive to a programmer. It is, however, helpful to have a theoretical foundation for reasoning about the use of customizable protocols in an application. In this section, we examine this issue in detail, with a focus on the following two questions that a programmer may wish to have answered:

- What are the semantics, in a formal sense, of using more than one protocol in an application? That is, what is the overall consistency model of the application, and when can two data objects be associated with different protocols?

- What is the set of protocols and communication optimizations that can be used in the *Ace* programming model? In other words, what are the limitations, if any, on the protocols and communication optimizations that can be implemented in *Ace*?

We begin by presenting an alternate view of the *Ace* model of computation with an analysis of the manner in which we expect programmers to use this system. We then attack the problem of the semantics of using multiple protocols and conclude with an examination of the limitations of *Ace* protocols.

2.4.1 Using Customizable Protocols

On most platforms, the semantics of sequential consistency are too strict to permit efficient protocol implementation (see Section 1.1). Relaxing the restrictions of this consistency model allows protocols greater freedom in the means they use to handle communication. For example, with release consistency, the effect of an access at a processor need not be made visible globally until a synchronization routine is invoked by the processor. Writes to a data object, therefore, need not block as long as synchronization events check that all previous writes have been performed globally before completing.

If a release consistent protocol is to be used with a program developed under the assumption of sequential consistency, the programmer must guarantee that *all* conflicting accesses are separated by synchronization to ensure that the original program's semantics

Processor 1	Processor 2	Processor 3
Read x		
Unlock	Write y	
Read y		**Lock**
Write y		Write x
	Read y	**Unlock**
Barrier	**Barrier**	**Barrier**
	Write x	

Figure 2.4: Execution trace of a program.

are met [38]. This requirement, in essence, reveals the drawback of a system supporting only a single fixed protocol. Consider the example in Figure 2.4. The figure depicts a possible trace of a sequentially consistent execution of a program. While programmers cannot visualize the execution of programs in such detail, they may be able to make certain assertions about the order in which accesses will be performed. For example, one possible assertion, necessary for release consistency, is that all conflicting accesses are separated by synchronization.

In most applications, however, broad assertions such as the one necessary for release consistency are not always true for all accesses. While synchronization may separate most conflicting accesses, it may not separate all accesses. For example, in the figure, accesses to x satisfy the release consistency contract, but accesses to y do not. If the underlying system supports a fixed release consistent protocol, a programmer must insert unnecessary synchronization to ensure that the required assertion is satisfied. Fixed protocols are designed to satisfy a particular access pattern and programmers must ensure that their applications confirm to the requirements of the protocol.

With customizable protocols, a programmer has the ability to partition the data into disjoint sets, each with its own access pattern. For each set, programmers check whether the access pattern satisfies the requirements of any protocol in the standard library, or whether the communication costs of accesses to data in that set are significant enough to warrant a custom protocol. This ability to divide data into independent sets and choose protocols appropriate for each set allows programmers to optimize an application's performance with little change to the original program. In the example, a programmer could partition the

Processor 1		Processor 2	
Write a, *1*	(1)	Write b, *1*	(2)
Read b, *0*	(3)	Read a, *0*	(4)

Figure 2.5: Example of the non-locality of sequential consistency. Write a, x denotes that the value x is stored in a. Read a, x implies that the read to a returns a value x.

data into two sets, one containing x, and the other, y. x would be associated with a release consistent protocol and y with a sequentially consistent protocol.

2.4.2 Interactions of Multiple Protocols

An advantage of single-protocol systems is that the consistency model is clear — it is that of the protocol. In the presence of the multiple protocols with different consistency models, it is not obvious how a programmer can reason about the overall consistency model of the application. In the following discussion, we assume that a program is developed initially using a sequentially consistent protocol and is modified subsequently to use custom protocols.

It has been shown previously that sequential consistency is not a *local* property [40]. To quote Herlihy and Wing, "a property P of a concurrent system is said to be *local* if the system as a whole satisfies P whenever each individual object satisfies P." In other words, if a programmer were to partition data into two sets and associate a sequentially consistent protocol with each set, the execution of the program as whole may not be sequentially consistent.

Consider the execution trace in Figure 2.5. In the example, if one were to examine the accesses of a, one could derive a sequential order — instruction 4 executed before instruction 1. Similarly, for b, instruction 3 executed before instruction 2. The accesses as a whole, however, are not sequentially consistent. The instructions cannot be interleaved in any manner that represents a legal sequential execution.

The implication of the non-locality of sequential consistency is that programmers cannot reason that the execution of an application is sequentially consistent by considering each space independently. Herlihy and Wing [40] have introduced a stricter form of se-

Processor 1		Processor 2	
Write a, *0*	(1)		
		Write a, *2*	(2)
Read a, *0*	(3)		

Figure 2.6: Example of an execution that is sequentially consistent, but not linearizable. The numbers in parentheses denote global clock timestamps.

quential consistency, *linearizability*, that is a local property of programs. In sequential consistency, abstractly, each processor has its own local clock. An execution trace is sequentially consistent if there exists an order of accesses that is equivalent to a sequential execution which preserves the order in which accesses occur at any processor, as given by the processor's local clock. Linearizability, in contrast, maintains a global clock for the system. To be linearizable, an execution trace must possess an order equivalent to a sequential execution which preserves the order in which accesses occur at *all* processors, as given by the global clock.

For example, Figure 2.6 gives an execution trace that is sequentially consistent, but not linearizable. Since sequential consistency must only respect intraprocessor ordering, a possible sequentially consistent execution is $1 \rightarrow 3 \rightarrow 2$. For linearizability, the order of accesses equivalent to a sequential execution must respect the time stamps of the global clock. In the example, according to the global clock, instruction 2 was executed before instruction 3. To be linearizable, therefore, any ordering equivalent to a sequential execution must have instruction 2 executing before instruction 3, which is impossible. This example is based on one presented in Herlihy and Wing.

One advantage of the linearizability property is in situations where there is more than one means of probing the state of a program. For example, consider the situation in which a distributed program executes on computers at two physically separated locations and two people collaborate over the telephone over the execution of the program. Under sequential consistency, it is possible that the two collaborators would temporarily see different views of the state of the program. In the example above, at time 2, the person at Processor 1 would see a value of zero in a, while the person at Processor 2 would see a value of two in a. The global clock in the linearizability property ensures that the view of memory is consistent throughout the system across time. We are, however, not as concerned with this characteristic of linearizability as with its local property, as discussed below.

Practical implementations of sequential consistency in software shared-memory systems, such as those in $\mathcal{A}ce$, CRL [47], and Blizzard [85] are, in effect, linearizable. A read on any processor that completes temporally later than a write will view the result of the write. As a result, one possible way of reasoning about the use of multiple protocols is to verify that each protocol is linearizable. The local property of linearizability ensures that if a program meets the requirements of each protocol, then the program, as a whole, is linearizable as well. The practical implication of this discussion is that if a programmer wishes to ensure that the result of a program using customizable protocols is equivalent to an execution on a sequential machine, the programmer must ensure that all protocols used in the program are linearizable (Herlihy and Wing discuss mechanisms for showing the linearizability of protocols [40]). If some of the protocols are not linearizable, the programmer must reason explicitly about the interactions of these protocols with the others in the program.

Note that in the discussion above, we have assumed that in using multiple protocols, programmers wish to retain the execution semantics of a sequentially consistent execution. It is, however, sometimes the case, that an application's semantics allow for chaotic accesses to data, for example, Gröbner basis calculation [18]. In these applications, accesses to stale data do not affect the correctness of the program. The execution semantics of these programs depend heavily on domain-specific knowledge held by the programmer. In these situations, therefore, it is the programmer's responsibility to reason about the consistency of the program explicitly.

2.4.3 Limitations of $\mathcal{A}ce$ Protocols

The use of customizable protocols is instrumental in the optimization of the access patterns of an application. Our interface for protocol development, full access control, allows events such as the beginning and end of reads and writes and synchronization points to be recognized. It is interesting to consider, in a formal automata sense, where, in the space of languages for access patterns, does full access control lie. While the answer to this question is open, we discuss some limitations on the set of protocols that can be implemented in *Ace*.

In message-passing applications, sometimes, the control flow of the program is based on the current state of memory. For example, if a particular data object is not resident in local memory, then a program may defer processing it until it is. As another example, in Olden [79], when a thread accesses data that is not local, a thread may be dispatched to the remote processor. In *Ace*, the execution model of the program is fixed (single program-multiple data) and there is no mechanism for dynamically creating threads or for modifying control flow based on the state of data. Such protocols, therefore, cannot be implemented.

It is sometimes useful for protocols to have type information so that transport of data may be optimized. Currently, *Ace* protocols are divorced from the language on the other side of the application/protocol interface, and therefore, do not receive any type information. For example, given the information that an access is to a node in a linked list and the type of a linked list, a protocol could prefetch the entire linked list if desired. In *Ace*, a protocol cannot determine which field holds the address of the next node in the linked list, and therefore, cannot implement this prefetch optimization. Finally, it is sometimes the case that while two data structures may have different access patterns, an action in one is triggered by an event is another. For example a read in one always follows a write in another. There is no mechanism in *Ace* by which protocols may interact with each other. Each protocol only views the data it manages.

2.5 Profiling and Debugging

In developing *Ace*, a mechanism that warns programmers of faulty association of protocols with data, that is, when the data's access pattern does not match that of the protocol, would be of great utility. The effects of these mismatches are typically unintended race conditions during the execution of the program. Netzer [68], however, has shown that finding all feasible races in a program is NP-hard.

The detection of a weaker category of race conditions is, however, sometimes tractable. *Apparent* race conditions [69] are those detected in an execution that is permitted by the synchronization in the algorithm. In other words, data dependences are ignored in determining the possible control flow. For example, a programmer may have semantic information about a program that states that a statement `B` will always execute after another statement `A` and not concurrently with `A`. In determining apparent race conditions, however, one would assume that `B` and `A` could execute concurrently unless *explicit* synchronization prevented them from doing so [69]. The set of apparent race conditions is a superset of the set of feasible races since some of the executions in the apparent case may not be realizable. The use of protocols to implement on-the-fly, apparent race detection schemes has been studied extensively by Richards in his dissertation [77]. A possible drawback of such schemes is that since apparent races are not guaranteed to be true races, these protocols may force programmers to sift through many *false-negative* errors to determine which of the races detected are actually feasible.

With respect to the development of protocols, in *Ace*, the burden of verifying a protocol's correctness rests on the protocol writer. Systems such as Teapot [21] can be used with *Ace* to facilitate this chore. Teapot is a programming language designed to allow easy specification of protocols. An optional output of Teapot is code that can be passed through the Murϕ system [29] (pronounced "Murphy") to verify the correctness of the protocol. The Murϕ system uses a brute-force technique to check a protocol's assertions, and therefore, can verify protocols for only small configurations of memory. In practice, however, it has proved useful in detecting bugs in protocol implementations [21].

The incremental approach to the development of *Ace* programs presents another opportunity for the debugging of programs. If the execution of a program with the default protocol is deemed to be correct, the results of its execution could be used to verify the execution of the program with various custom protocols. While this process does not offer any guarantees, it is a useful check for the correctness of a program.

A logical extension of the *Ace* programming model is to have the runtime system or compiler select protocols automatically based on profiling data. The nature of access patterns, however, are not understood well enough to permit this feature. At present, programmers may avail themselves of profiling information such as the number of reads and writes executed and the number of cache misses to help them determine when communication optimization is necessary. It is hoped that continued experience with the system will lead to a better understanding of the profiling mechanisms necessary for easy program development.

2.6 Discussion

The easy use of customizable protocols relies on the presence of mechanisms that allow protocols to be treated as interchangeable modules. For this purpose, it is necessary to define an efficient interface between the algorithmic and communication components of an application so that the protocol code may be extracted from applications and stored in modular libraries. Furthermore, this interface must be powerful enough to allow the implementation of protocols for common access patterns. The *Ace* interface, called full access control, allows protocols the ability to specify event handlers that are invoked at the beginning of reads and writes and at synchronization points.

While benefits in efficiency from using customized protocols for programs are expected, customizable protocols can play the role of a software engineering tool as well. Programs are first developed and tested easily by using a standard well understood protocol. Subsequently, the communication performance can be improved, independent of the application code, by experimenting with modular protocol libraries. The key to this feature is the separation of algorithmic and communication code. There are, however, certain disadvantages to defining an interface between protocols and algorithmic code. For one, a protocol may require knowledge of the types and local state of the program to optimize communication. Complete encapsulation of communication code implies that this information does not pass through the interface.

Consider *Ace*'s programming model in the context of object-oriented languages. Abstractly, a customizable protocol overloads the semantics of loads and stores to an object. Extending *Ace*'s programming model to this language domain may suggest a mechanism by which a programmer could associate different protocols with data structures cleanly by overriding the default semantics of loads and stores, but still provide the protocol with type information that could be used to optimize communication. This is one possible area of future research.

The introduction of a new abstraction, *space*, allows programmers to associate data structures with protocols intuitively. Currently, spaces, which dispatch all accesses and synchronization to data allocated from it to the appropriate protocol, are "flat" abstractions; spaces do not interact with each other. It is possible that under certain circumstances, hierarchical spaces or cooperating spaces are desirable. Again, in an object-oriented model, these issues may have a natural solution in the inheritance and subtyping mechanisms.

To remove problems associated with false sharing of protocols in *Ace*, user control over coherence granularity is essential. There are other systems, such as CRL [47] and Shasta [83], that are based on user-specified granularity as well. One limitation of user-specified granularity is that the granularity of an object may change across time, that is, in one phase of an application, an object is accessed as a whole, but in parts in other phases. In this case, user-specified granularity forces programmers either to set the granularity to the whole data object or to break the data structure into its constituent parts explicitly. The first solution entails extraneous communication during phases where only a small part of the data is accessed, the second requires additional programming effort. The issue of coherence granularity — given an application, how should one assign data to coherence units — and the abstractions required for the easy specification of coherence granularity are not yet well understood and present another avenue of further research.

If each protocol in a system is shown to be linearizable,[1] programmers can reason about the semantics of programs that use multiple protocols by considering each protocol independently. In practice, it is generally straightforward to determine that a protocol is well suited to a data structure's access pattern. A deeper theoretical understanding of the interactions of protocol, however, is essential in developing debugging and verification tools, powerful compiler optimizations, and profilers that can use runtime information to recommend appropriate protocols,

Several of the data race protocols developed by Richards [77] can be used with *Ace* to debug programs. One characteristic of these protocols is that while they detect all true race conditions, they may report several spurious races as well. It is up to the programmer to determine, based on the information provided by the protocols, whether the execution of the program is correct. With respect to the correctness of protocols, systems such as Teapot [21] allow protocols to be built and verified more easily.

[1] As mentioned earlier, Herlihy and Wing discuss mechanisms for proving the linearizability of protocols [40].

Chapter 3

The *Ace* Language

Modern parallel programming languages are often designed as extensions of popular sequential programming languages such as C or Fortran. Designing these languages as extensions rather than as completely new languages has several benefits. A familiar base environment eases the learning process for programmers. It also allows for greater reuse of existing code for the sequential portions of a program. Furthermore, it is easier to leverage existing compilers and tools for the new language. For these reasons, *Ace* was designed as an extension to C.

It is necessary to introduce certain new constructs in order to allow programmers to develop shared-memory programs with customizable protocols. In the previous chapter, we discussed issues relevant to the design of a language for customizable protocols. In this section, we describe the specific constructs in *Ace* that address these issues. This chapter focuses on the programming model provided to *Ace* application developers. The next chapter includes a discussion of how protocols are developed and added to the system (Section 4.4). We first discuss the programming model of the language in Section 3.1, with an example provided in Section 3.1.3. We conclude with a discussion in Section 3.2.

3.1 Programming Model

The execution model is single program-multiple data (SPMD), that is, it supports a single user thread per processor, where each processor runs the same executable. Each processor is assigned a unique numeric identifier at startup, which is stored in the variable, _ace_self_address. SPMD programs often use this identifier to assign work to processors. The _ace_num_nodes variable is set automatically by the system to the number of processors executing the program.

Since *Ace* supports a shared-memory model, most communication is accomplished using locations in shared memory. Collective communication, such as routines to broadcast a value to all processors, or to calculate the minimum value of a variable across processors, is supported as well. Synchronization operations allow programs to manage the overall control flow of the program. All these operations are implemented as runtime library calls. The language itself is essentially C with minor modifications. In this section, we describe the components of the *Ace* programming model. We begin by outlining the features of the language, and then, the runtime library routines that are essential to the programming model. We conclude with an example of an *Ace* program.

3.1.1 Language

As mentioned above, the *Ace* language is essentially C, with a few additional constructs. Unlike C, *Ace* programs do not declare a main routine, but an ace_main routine instead. Execution of a program commences with this routine on all processors. *Ace* requires that all globally shared data be allocated dynamically in the heap. Furthermore, to inform the system that a declaration refers to globally shared data, programmers must annotate it with the keyword **shared**. Since globally shared data is always allocated in the heap, one cannot declare variables that are shared, only pointers to shared data. Table 3.1 gives a few examples of *Ace* declarations. All data not declared to be shared are assumed to be local and are allocated on the stack or from a processor's private memory. Finally, the compiler assumes (see Section 5.2.2) that a program will not use pointer arithmetic to access arrays beyond bounds, or generate new addresses by adding arbitrary constants to *Ace* addresses. The correctness of such irregular pointer arithmetic is, in any case, highly implementation-specific, and is not guaranteed under *Ace*.

A programmer may be conservative in marking declarations of pointers to shared data. Annotating data that is not globally shared with **shared** does not change the correctness

Table 3.1: Valid and invalid *Ace* declarations.

shared int $*$x	Pointer to shared integer.
shared int A []	Array of shared integers.
shared int $*$ **shared** A [10]	Array of ten shared pointers to shared integers.
shared int x	*Invalid Declaration*
shared int A [10]	*Invalid Declaration*

```
void Process (shared int * ptr) {
    ⋮
}
void main () {
    int stack;
    shared int *shared_ptr;
    Process (shared_ptr);
    Process (&stack);
}
```

Figure 3.1: Example of conservative annotation of declarations.

of a program. For example, in Figure 3.1, the function Process is called twice, once with an argument that is a pointer to shared data, and once with a pointer to an integer on the stack. Declaring the formal parameter as a pointer to a shared integer will handle both cases appropriately. Annotating local data as **shared** results in some *Ace* runtime system overhead for each access and is not as efficient as annotating such data more precisely.

The requirement that all declarations of globally shared data be marked with the keyword **shared** is not intrinsic to the *Ace* programming model, but is a characteristic of the present implementation. We shall describe in Chapter 5 how the protocol associated with an access can be inferred. One can treat variables that access strictly local data as being associated with a special protocol and infer conservatively whether an access will touch global data or not. An enhanced compiler could also remove the restriction that all global data be allocated in the heap.

3.1.2 Runtime Library Functions

In this discussion, we outline the various runtime library system routines that are essential to the development of *𝒜ce* programs. Table 3.2 lists the *𝒜ce* library routines.

Using Spaces and Protocols

Space and Protocol are predefined types in *𝒜ce*. All the protocols available to an application programmer are predefined by the system. In general, a programmer allocates a space for each data structure in the program using the Ace_NewSpace command, which takes a Protocol as an argument and returns a Space. A default space, _ace_sc_space, that is always associated with a sequentially consistent protocol is predefined by the system.

During the execution of a program, the protocol associated with a space can be changed with the Ace_ChangeProtocol routine. The semantics of a protocol change involves manipulating objects into a "base" state, that is, ensuring that the home processor has exclusive write permission on the latest copy of the data. In addition, changing the protocol of a space is similar to a barrier operation — each processor must execute the change operation for it to complete.

Memory Management

Globally shared data is allocated using the Ace_GMalloc call, which takes as an argument the space from which the data is to be allocated. All subsequent accesses to the data will be handled by the protocol associated with the specified space. Using the default space, _ace_sc_space, ensures that the data will be managed by a sequentially consistent invalidation-based protocol. The granularity of coherence in *𝒜ce* is user controlled. Each invocation of Ace_GMalloc for size bytes creates an atomic coherence unit of size bytes and returns its address. To be consistent with terminology in the field, these atomic coherence units will be referred to as *regions*. Memory for data that is private to a processor is allocated with the Ace_Malloc routine. Ace_GFree is used to free globally shared data, while the Ace_Free routine is used to deallocate private data.[1]

[1]In the current implementation, global data cannot be freed. The problem of freeing shared data efficiently while maintaining memory arena consistency requires further study. Most of the benchmarks we studied do

Table 3.2: 𝒜*ce* runtime library routines.

𝒜*ce* Library Routines	Description
	Space Related Routines
Ace_NewSpace(*protocol*)	Create a new space.
Ace_ChangeProtocol(*space*, *protocol*)	Change the protocol of a space.
	Memory Management
Ace_GMalloc(*space*, *size*)	Allocate a new shared object from *space*.
Ace_GFree(*global address*)	Free global data.
Ace_Malloc(*size*)	Allocate local data.
Ace_Free(*local address*)	Free local data.
	Collective Communication
Ace_SendBcast(*size*, *local address*)	Broadcast size bytes, starting at the address in local address.
Ace_RecvBcast(*size*, *local address*)	Receive broadcast data.
Ace_ReduceAddD(*double **)	Perform an add reduction operation.
Ace_ReduceMaxD(*double **)	Perform a maximum reduction operation.
Ace_ReduceMinD(*double **)	Perform a minimum reduction operation.
	Synchronization
Ace_Barrier(*space*)	Execute a barrier operation with the semantics specified by *space*.
ACE_LOCK(*global address*)	Lock a region.
ACE_UNLOCK(*global address*)	Unlock a region.
	Miscellaneous
Ace_GetElapsedTime()	Return the number of seconds since program started.

Collective Communication

In certain models of computation, execution of a program begins in a single master processor. The master processor initializes the global state of the program and then forks off slave threads, which inherit pointers to the global state from their parent. In *Ace*, however, all processors commence execution at the same instant in the ace_main routine. There must be, therefore, some way for the processor to bootstrap the shared memory, that is, a mechanism by which a processor "knows" the address of an object on another processor. Once a processor has the means of accessing one object, through it, it can access other objects.

For the purpose of bootstrapping shared memory, *Ace* supports broadcast primitives that allow processor to broadcast data to other processors. The processor performing the broadcast invokes Ace_SendBcast, and the others, Ace_RecvBcast. These routines take the address and size of the data to be broadcast as parameters. Currently, only local data may be broadcast.

In addition to bootstrapping shared memory, collective communication routines such as broadcasts and reductions have other uses. In several parallel applications, a reduction to find the maximum value of a variable on all processors is a common operation. *Ace* provides reduction operations that can be used to calculate the sum (Ace_ReduceAddD), the minimum (Ace_ReduceMinD), and the maximum value of a variable across processors (Ace_ReduceMaxD). These routines operate on variables of type double.

Synchronization

Ace programmers use barriers and locks for synchronization. Ace_Barrier takes a space as an optional argument. If present, it determines the protocol that will perform the barrier operation. If no space is provided, a default barrier operation is executed. The default barrier has conventional semantics, that is, no processor can leave a barrier until all processors have reached the barrier.

In *Ace*, locks are implicitly associated with all *regions*. The ACE_LOCK (ACE_UNLOCK) routine takes as argument, the address of the region on which a lock (unlock) is desired. The semantics of these operations depend on the protocol currently associated with the

not allocate and free memory often, and as a result, do not require this call.

```
#include "ace.h"
typedef struct taskq_t {
     shared work_t *the_work;
     shared taskq_t *next;
} taskq_t;
shared struct queue_t {
     shared taskq_t *head;
     shared taskq_t *tail;
} *the_queue;
Space work_space;
```

Figure 3.2: Declarations for sample *Ace* program

space from which the region was allocated. While protocols may define these routines as they see fit, default routines are provided that implement conventional locks.

Miscellaneous

The Ace_GetElapsedTime routine can be used by programmers to measure the time spent in various portions of the program. It returns the number of seconds that have elapsed since the program began execution. By calling this routine once before the execution of a phase of the program, and once following the completion of the phase, and subtracting the first result from the second, a programmer can measure the time spent in the phase.

3.1.3 Sample *Ace* Program

The use of *Ace* constructs is illustrated best through an example. The program that we shall examine in this section is an example of a parallel algorithm that uses a central work queue. Figure 3.2 shows the declarations used by the sample program. The program is listed in Figure 3.3. The variable, the_queue, points to a structure that refers to the head and the tail of the central task queue in the program. Each task contains a reference to the data in shared memory (of type work_t) that is required for the completion of the task. A space variable, work_space, is declared from which the work data is allocated.

In the program in Figure 3.3, at the onset of execution, control resides in the ace_main

```
/* Fills up queue with work allocated from work space */
extern void AllocWork();
extern void DoWork(shared work_t *);

void ace_main() {
    shared taskq_t *a_job;
    double start, stop;
    work_space = Ace_NewSpace(SC);        /* Sequentially consistent */

    if (_ace_self_addr == 0) {
        the_queue = Ace_GMalloc(_ace_sc_space, sizeof(struct queue_t));
        the_queue→head = the_queue→tail = NULL;
        Ace_SendBcast(sizeof(struct queue_t *), &the_queue);
    }
    else Ace_RecvBcast(sizeof(struct queue_t *), &the_queue);
    /* Now all processors have the address of the queue variable */
    AllocWork(); /* Fill the_queue with work */
    Ace_ChangeProtocol(work_space, UPDATE);

    start = Ace_GetElapsedTime();
    while (1) {
        ACE_LOCK(the_queue);
        a_job = the_queue→head;
        if (a_job == NULL) {
            ACE_UNLOCK(the_queue);
            break;
        }
        the_queue→head = a_job→next;
        ACE_UNLOCK(the_queue);

        DoWork(a_job→the_work);
    }
    Ace_Barrier();
    stop = Ace_GetElapsedTime();
    if (_ace_self_addr == 0)
        printf("Time taken: %5.3lf\n", stop - start);
}
```

Figure 3.3: Sample 𝒜ce program

procedure. Processor 0 allocates space for the queue from shared memory and initializes the structure. It then broadcasts the address of the queue to the other processors (Lines 10–15). Once the processors have received the address of the queue, they create work for the queue (allocating data for each task from work_space), and add the tasks into the queue in the AllocWork procedure (not shown in the figure). Subsequently, they enter a loop (Lines 21–32), where they attempt to lock the queue, extract the head of the queue, and then, perform the work associated with the task. The program ends when the queue is empty. The Ace_Barrier call on Line 33 ensures that all processors finish their tasks before the program exits. The time taken to execute the tasks is measured with the Ace_GetElapsedTime routines on Lines 20 and 34.

Typically, the program would be developed without the Ace_ChangeProtocol invocation on Line 18. Once the program is tested and found to run correctly, the programmer can experiment with different protocols to determine the one that achieves the best performance. Assume that the access pattern of the work data in the routine DoWork (not shown in the figure) is well suited to an update protocol. Then, as shown in the example, an Ace_ChangeProtocol routine would be inserted to change the protocol of the work data to UPDATE. The programmer may also choose to allocate a new space for the task data structure and experiment with protocols that access taskq_t structures more efficiently. These experiments can be performed with few changes to the program.

3.2 Discussion

The *Ace* programming language is similar to the programming models of other explicitly parallel systems such as Split-C [25], Treadmarks [6], etc. The few new constructs introduced for the manipulation of spaces and protocols are relatively simple. We have found it straightforward to port programs written for other explicitly parallel systems to *Ace*.

In Figure 3.4, we list versions of a dot product routine written for *Ace*, the conventional shared-memory model, and CRL. When compared to runtime systems such as CRL [47], we find *Ace* considerably easier to program. In region-based runtime systems such as CRL, programmers must insert annotations around accesses to shared data. In addition, global addresses do not have types and must be cast explicitly to appropriate types. For example,

```
double dotprod(shared double *x, shared double *y, int n) {
    int i;
    double rslt;
    rslt = 0;
    for (i = 0; i < n; i++)
        rslt += x[i] * y[i];
    return rslt;
}
```

(a)

```
double dotprod(double *x, double *y, int n) {
    int i;
    double rslt;
    rslt = 0;
    for (i = 0; i < n; i++)
        rslt += x[i] * y[i];
    return rslt;
}
```

(b)

```
double dotprod(rid_t x, rid_t y, int n) {
    int i;
    double *u, *v;
    double rslt;
    u = (double *) rgn_map(x);
    v = (double *) rgn_map(y);
    rgn_start_read(u);
    rgn_start_read(v);
    rslt = 0;
    for (i = 0; i < n; i++)
        rslt += u[i] * v[i];
    rgn_end_read(u);
    rgn_end_read(v);
    rgn_unmap(u);
    rgn_unmap(v);
    return rslt;
}
```

(c)

Figure 3.4: Dot product routine written in (a) *Ace* (b) standard shared-memory model (c) CRL.

in Figure 3.4c, rid_t is the CRL system type for a global address. The programmer must cast uses of these addresses (see the rgn_map calls) to the appropriate types explicitly. In $\mathcal{A}ce$, the presence of a compiler for type-checking and implementing shared memory is essential to reducing the burden on programmers. The primary difference between $\mathcal{A}ce$ and standard shared-memory programs are the **shared** annotations.

Chapter 4

Implementing *Ace*

We now turn to the implementation of the *Ace* programming model, with a focus on implementations on distributed-memory architectures such as the Thinking Machines CM-5 [59] and the Cray T3E [86]. On pure message-passing architectures such as the CM-5, the responsibility for maintaining a shared-memory abstraction lies wholly on the programming system. The system simulates a shared address space and the effects of shared-memory accesses using the message-passing routines provided by the platform. The Cray T3E has some hardware support for simulating a shared address space. The Cray T3E allows a processor to read any memory location in the entire machine without the intervention of remote processors. It does not, however, maintain global coherency of the data in the machine's memory — coherence must be maintained by software. The mechanisms developed for implementing *Ace* on these platforms are described in the next two chapters.

Figure 4.1 depicts the organization of the *Ace* implementation. *Ace* programs are developed using the customizable protocol programming model and *Ace* runtime library calls (Table 3.2). The compiler uses an interface defined by the runtime system to ensure that the appropriate sequence of actions are performed for each access to shared data. The compiler transforms an *Ace* program into C code annotated with runtime-system interface calls that has the same execution semantics as the original *Ace* program. The output from the *Ace* compiler is then compiled using a native C compiler, such as gcc, and linked with the runtime system and the appropriate protocol libraries to form an executable for the platform machine.

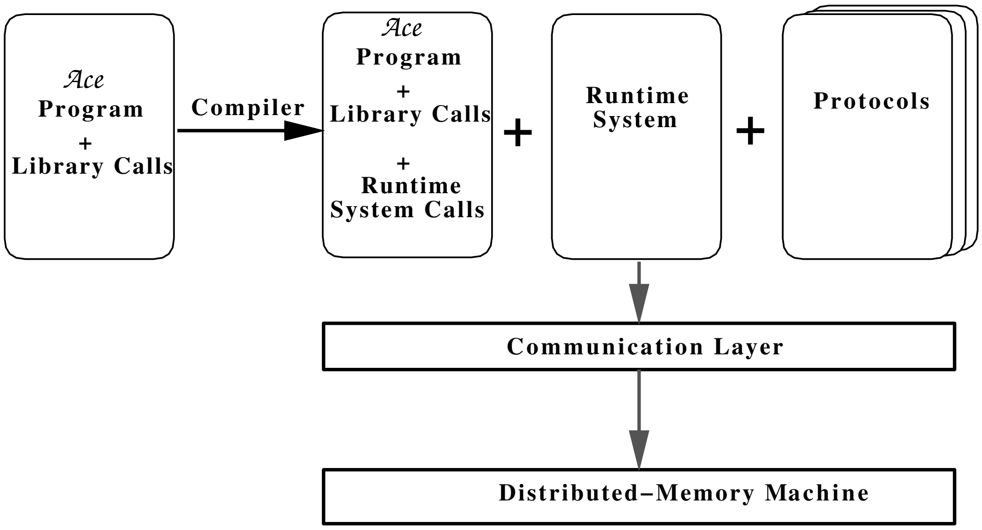


Figure 4.1: Organization of *Ace* implementation.

The responsibility for supporting *Ace*'s programming model lies with the runtime system. The runtime system uses carefully designed interfaces to bind together the various components of the *Ace* system — *Ace* programs, the compiler, and the protocols. The runtime system also mediates interactions between these components and the underlying architecture. On the CM-5, the runtime system is built on top of an Active Message library supplied by the platform machine. Since shared-memory communication is intrinsically asynchronous, a one-way message-passing model such as Active Messages [95] (see Section 1.1 for background), is most appropriate. On the Cray T3E, as mentioned earlier, the system provides support for addressing remote processors's memory. We use the shmem library on the T3E [7], which provides routines for reading and writing blocks from a remote processor's memory without interrupting the remote processor.

The two platforms on which we have implemented *Ace*, the CM-5 and the T3E, have different communication primitives. On the CM-5, a processor cannot operate on data (or the state of a data object) that resides remotely without interrupting the remote processor. As such, the most efficient means of modifying remote data is to send an active message to the remote processor that causes a handler to execute at the remote processor that will modify the data. On the T3E, in contrast, the hardware permits the modification of remote memory without the intervention of any remote processors. The implementation of the runtime system and the protocols on the two platforms are similar in an abstract sense, but the different communication abstractions on these machines influence their actual implementation. Whereas on the CM-5, an active message may be sent from one processor to another to modify data, on the T3E, the processor that wishes to modify data would do so directly. The descriptions in this chapter focus on the Active Message model of the CM-5. Where appropriate, we highlight differences in the T3E implementation.

In this chapter, we focus on the design and implementation of the *Ace* runtime system. We begin by describing the implementation of the runtime library calls available to *Ace* programmers in Section 4.1. We then discuss the interface used by the compiler to convert *Ace* programs into code annotated with runtime system calls (Section 4.2). The runtime system handles accesses according to the semantics of the *Ace* language by acting as a bridge between the compiled code and the shared-memory protocols. For every access to global data, it invokes the appropriate protocol routine to handle the access. We discuss the mechanisms for this dispatch in Section 4.3.

The programming environment presented by the runtime system for the development and addition of protocols is examined in Section 4.4. The *Ace* runtime system provides a set of primitives that form a base on which protocol writers can develop protocols easily. This layer implements certain shared-memory operations common to many protocols and permits the use of high-level operations in developing protocols. For example, the runtime system provides primitives for common operations such as requesting an object from a remote processor.

In Section 4.5, we describe the protocols in *Ace* that are considered part of the "standard" protocol library. The wide variance in access patterns of applications implies that there will be situations where, for optimal performance, a programmer will need to develop custom protocols. It stands to reason, however, that as the *Ace* system evolves, a set of protocols will emerge that is sufficient for the most common access patterns. The *Ace* standard protocol library is an attempt to provide such a set of protocols for certain communication patterns that are encountered often in parallel applications. We summarize the runtime system component of the *Ace* implementation in Section 4.6. The design and implementation of the compiler are described in Chapter 5.

```
struct space {
    char *protocol_name;
    unsigned spacenum;
    void ( *methods[NUM_METHODS] ) (address_t);
    Arena malloc;
    Cache *objects;
    void *aux;
}
```

Figure 4.2: Representation of a space in *Ace*.

4.1 Runtime Library Functions

The implementations of the runtime library routines provided to programmers for protocol management, memory management, collective communication, synchronization, and timers are discussed in this section. The semantics and use of these routines are listed in Table 3.2 (page 42).

4.1.1 Using Spaces and Protocols

The structure of a space is as shown in Figure 4.2. The aux field is a pointer which can be used by a protocol to store space-specific state information, such as sharer lists for objects in a space (as in the STATIC protocol discussed in Section 4.5.3). The other fields are described below. When a new space is created using Ace_NewSpace(*space*, *protocol*), a space structure is allocated by the system on each processor. The space is assigned a system-wide unique identifier, which is stored in the spacenum field. The space identifier is an index into a *space table* array that is allocated on each processor. Given a space identifier, a processor can determine the address of the corresponding space structure by indexing the space table array.

Once the space identifier is assigned, an initialization handler for the specified protocol is invoked. The initialization handler invokes an internal routine, _Init_Space, with the addresses of the handlers for that protocol as arguments (Section 4.4 describes protocols in greater detail). The internal routine stores these addresses in the methods field of the space

```
for (all objects in sp→objects)
    if (owner is remote processor)
        if (object is in ReadWrite state)
            Flush object back home.
        endif
        Set object state to Invalid
    else /* I am the owner */
        Set object state to ReadWrite
    endif
endfor
Wait until all processors have finished flushing data, and all
    updates are received.
```

Figure 4.3: Pseudocode executed on each processor for flushing data to a "base" state.

structure. This mechanism provides the runtime system with pointers to the event handlers associated with a space.

The semantics of the dynamic changing of protocols states that all data is returned to a "base" state (Section 3.1.2), where all cache copies are flushed to their home nodes. In the *Ace* runtime system, when remote data is created or cached, a pointer to its state is appended to the objects field in its space structure. When a program changes the protocol of a space with Ace_ChangeProtocol, the runtime system traverses this list and flushes each cached object back to its home node (Figure 4.3 provides the pseudocode for this process). Once data is manipulated into the "base" state, the new protocol's initialization routine is executed. This sets the methods field of the space to the event handlers associated with the new protocol.

4.1.2 Memory Management

The memory allocation mechanism in *Ace* centers around an *arena manager* component that creates new memory arenas and fulfills requests from arenas for memory. When a space is created using Ace_NewSpace, it is assigned an independent memory arena. At system startup, an independent arena is also allocated for handling requests for private memory, that is, memory that is not shared.

The implementation of the Ace_Malloc call, which handles allocations of private memory, is straightforward. A memory request is forwarded to the private memory arena. The private arena satisfies the request, obtaining additional memory from the arena manager if necessary. When an arena requests memory from the arena manager, the request size is rounded up to the nearest page size. This step ensures that data in a page are either all local, or allocated from the same space, as is required by the protocol dispatch mechanism described in Section 4.3.

The Ace_GMalloc routine works in a similar fashion, but dispatches memory requests to the specified space's arena. The arena allocates memory for the data object from the processor's local memory. A unique global address is then created for the object. On the CM-5, the internal type for a shared address, address_t (shown in Figure 4.4), uses a format derived from that used in Olden [79]. The processor identifier of the allocating processor is prepended to the physical address of the data to form a global address. The most significant bit is used to distinguish local and shared addresses — all addresses above 0x80000000 are from global shared memory. Since on the CM-5, physical addresses use only 25 bits of an address, the upper 7 bits of an address can be used to store this information. The identity of the home processor of an address can be isolated using bit arithmetic.

The Cray T3E has 64-bit addresses, of which only 43 bits are used by the system. As in the CM-5 implementation, we store the home of an object in its address. In addition, since there are many free bits available, the address of an object also contains the identifier of the space to which the object belongs. Storing the space in the address simplies the protocol dispatch process as described below in Section 4.3.

As discussed previously in Section 3.1.2, user-specified granularity implies that each invocation of Ace_GMalloc creates an atomic coherence unit. Once a new global address has been created, the Ace_GMalloc routine registers the atomic coherence unit with the runtime system. Section 4.4.2 discusses what registering a coherence unit entails. Pseudocode for the Ace_GMalloc routine on the CM-5 is provided in Figure 4.5.

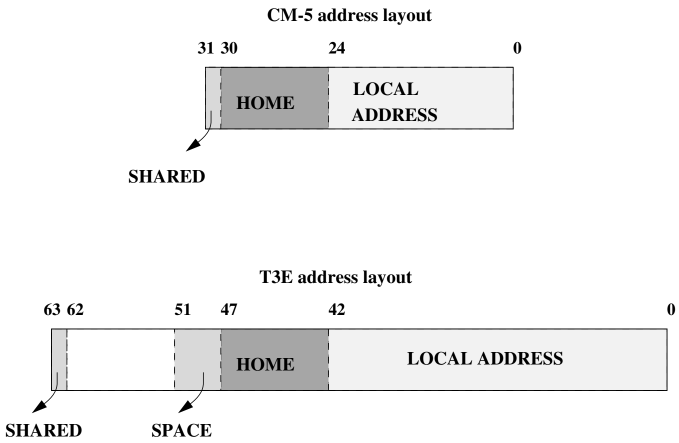


Figure 4.4: Format of *Ace* addresses (type address_t) on the Thinking Machines CM-5 and the Cray T3E.

proc Ace_GMalloc (*space*, *size*)
    addr = space→arena (size) /∗ *Allocates memory* ∗/
    addr $| =$ 0x80000000
    addr $| =$ _ace_self_addr $<<$ 25
    Register object with the runtime system.
    return addr
endproc

Figure 4.5: Pseudocode for allocating globally shared data on the CM-5.

4.1.3 Collective Communication

On the CM-5, hardware support is present for the implementation of collective communication primitives. For example, the CM-5 has a separate synchronization network that implements reductions and broadcasts efficiently. The *Ace* runtime library routines for reductions and broadcasts are essentially wrapper routines for the CM-5 hardware mechanisms. The Cray T3E provides library routines that implement the common collective communication primitives. Again, the *Ace* calls for collective communication are wrappers around the provided library routines. A large body of efficient algorithms for collective communication exists [39, 48, 49, 65] that can be used to implement these routines when system support is unavailable.

4.1.4 Synchronization

While each protocol may provide its own implementation of synchronization routines, the runtime system provides default routines that may be used by protocols. On the CM-5, the default barrier routine is implemented with the aid of the underlying hardware synchronization network. A subtle problem arose during the implementation of this routine. The CM-5's barrier routine, CMMD_sync_with_nodes, polls only the synchronization network when waiting for all processors to reach a barrier.

Consider the case where Processor a has reached a barrier, while Processor b has sent a request message to a. If the barrier operation is implemented with CMMD_sync_with_nodes, and Processor a reaches the barrier before the request from b arrives, then deadlock will occur. Processor a will poll the synchronization network waiting for all processors to execute the barrier. Processor b will poll the data networks waiting for a reply from a and never reach the barrier. *Ace*'s solution is to use a modified version of the CM-5 barrier routine that checks both data and synchronization networks while a barrier is being performed. On the Cray T3E, the system provides a barrier routine that is guaranteed to check the message queues while processors are waiting at the barrier. The *Ace* barrier is a wrapper around this routine.

In *Ace*, as mentioned in Section 3.1.2, a locking variable is associated implicitly with each region. The default implementation of ACE_LOCK and ACE_UNLOCK provides a general framework for specific implementations of these routines by protocols. When a processor requests a lock on a region, the runtime system checks a *lock hash table* on that processor to determine if the processor has locking permission on the coherence unit. If the region is not found in the hash table, a new entry is created for it, and the locking permission set to FALSE.

On the CM-5, to obtain locking permission, a processor sends a message to the home processor requesting the lock. If the lock entry for that region is not present in the home processor's lock hash table, it creates a new entry for it and replies to the requesting processor granting it lock permission. Otherwise, it forwards the message to the current owner of the lock. The home processor then considers the requester to be the current owner of the lock; that processor will be forwarded the next request that arrives at the home processor.

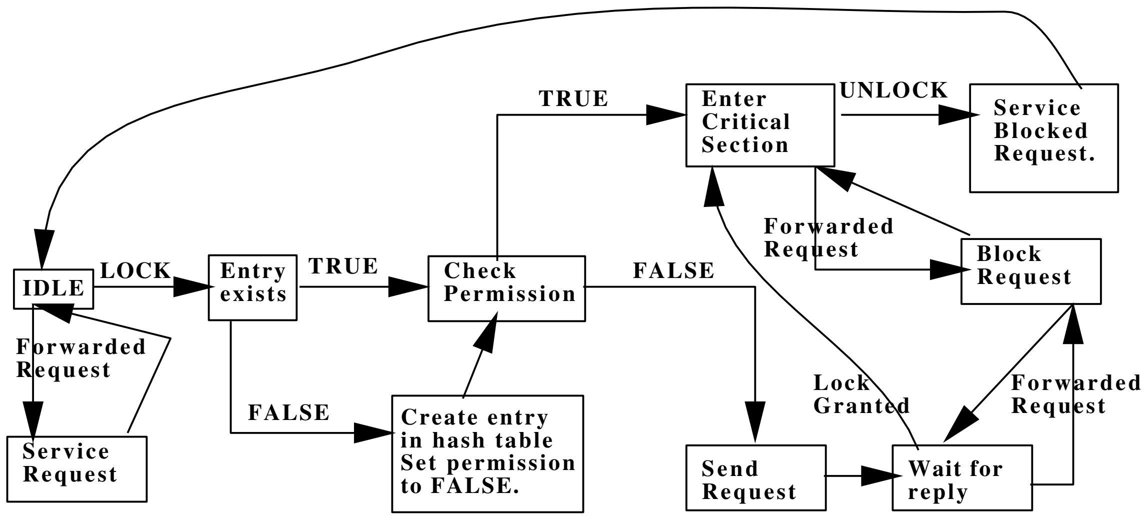
When a processor receives a forwarded request for a lock, if it has the lock and is not currently using it, it relinquishes the lock to the requester. If the processor has not yet received the lock, or is currently in the middle of the critical section regulated by the lock, it waits until the corresponding unlock operation is executed at that processor before satisfying the request. Using a lock hash table to allocate lock variables lazily ensures that memory is conserved by allocating space only if a processor uses a lock. The state machines for this algorithm are provided in Figure 4.6. The Cray T3E implementations are similar except that modifications of remote lock hash table entries are performed using the system-provided atomic swap instructions.

4.1.5 Miscellaneous Functions

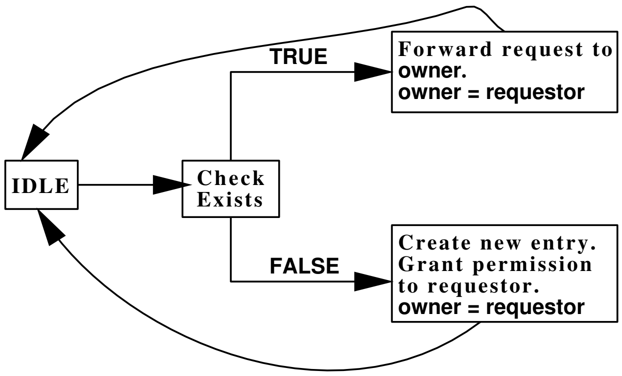
Most parallel machines provide facilities that allow for timing the execution of programs. On the CM-5 and the T3E, the *Ace* timing routine is implemented as a simple wrapper around the native timer library calls.

4.2 Compiler/Runtime-System Interface

In a sense, the *Ace* runtime-system primitives serve as an intermediate language for the compiler; the functionality necessary to implement the *Ace* programming model is en-



(a)



(b)

Figure 4.6: (a) State machine for lock actions at processors requesting locks on an object (b) State machine for lock actions at the home processor of the object. The boxes represent states, and the arcs represent events at a processor. LOCK = Processor executes a lock operation. Forwarded Request = A forwarded request active message arrives at a processor. Lock Granted = An active message arrives at a processor granting a lock. UNLOCK = Processor executes an unlock operation. Unlabeled arcs imply that the state transition occurs the instant the action associated with the state is performed.

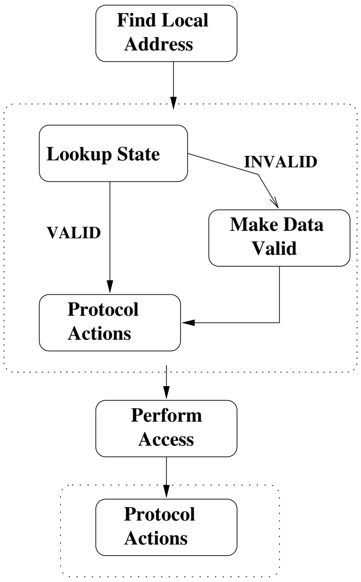


Figure 4.7: Anatomy of an access in *Ace*.

capsulated into a few key operations. The interface operations are designed to be simple enough to allow a compiler to transform *Ace* programs easily by instrumenting them at appropriate points with interface primitives, but exposes enough functionality to the compiler to allow it to optimize costly operations.

The design of the compiler/runtime-system interface can be understood best by considering the anatomy of an access in the *Ace* system (Figure 4.7). Distributed shared-memory systems use varying degrees of hardware and software techniques for handling accesses to global data, but in an abstract sense, the mechanisms they use are similar. As shown in the figure, the first step is to perform *address translation*, that is, determine from the global address, the location in local memory where the data or a cached copy can be found. For example, in hardware shared-memory systems, the hardware implements a shared address space. When data are cached, their addresses determine the set of locations in the hardware cache where they may be placed. When cached data is accessed, the hardware checks this set of locations to retrieve the data.

The system must then determine if a valid copy of the desired data exists locally, that is, the system performs a *cache state lookup*. What constitutes a valid state depends, of course, on the protocol being used. Since a protocol may assert that an access is always

Table 4.1: Compiler/Runtime-System Interface

Routine	Description
void *ACE_MAP(*address*)	Return location of copy of data in local memory.
void ACE_START_READ(*address*), void ACE_END_READ(*address*), void ACE_START_WRITE(*address*), void ACE_END_WRITE(*address*)	Dispatch to corresponding protocol handler.

valid, this phase may not be necessary, and hence, it is shown in a dotted box in the figure. For most protocols, however, a cache state lookup is essential. If the cache state lookup fails, protocol actions are necessary to manipulate the data into a valid state. After the state is checked to valid, the access can be performed. Once the access is completed, the protocol may execute any actions necessary to maintain coherence.

In *Ace*, it is the compiler's responsibility to ensure that accesses to globally shared data are handled in the manner as outlined in Figure 4.7. The interface between the compiler and the runtime system reflects this responsibility. The runtime system must be invoked before and after accesses to perform appropriate protocol actions. The runtime system must also be called to perform address translation on data. The *Ace* compiler/runtime system-interface is similar to that used by programmers of region-based runtime systems such as CRL [47]. The interface calls are listed in Table 4.1.

The compiler inserts ACE_START_READ and ACE_END_READ calls around loads to shared memory in an *Ace* program (ACE_START_WRITE and ACE_END_WRITE around stores). These routines call the protocol dispatch routine, which determines the appropriate protocol handler to invoke as described in Section 4.3. The compiler inserts ACE_MAP calls before accesses to shared data to perform address translation. Figure 4.8 gives an example of this process. The compiler optimizes the generated code by reducing, where possible, the number of runtime-system calls inserted. By limiting the compiler/runtime-system interface to these few operations, programs can be transformed easily, while allowing the compiler sufficient opportunities for optimization. The compilation process and optimizations are described in detail in the next chapter.

```
shared int *x;
int y;
y = *x;
```

```
int x, y;
x_local_addr = ACE_MAP (x);
ACE_START_READ(x);
y = *x_local_addr;
ACE_END_READ(x);
```

Figure 4.8: Compiler instrumented code for y = *x.

4.3 Spaces and Protocol Dispatch

The compiler/runtime-system interface ensures that the runtime system is invoked before and after every access. The synchronization routines are also implemented as runtime library calls. As a result, the runtime system gains control at all events required for full access control. What is required is a mechanism by which the runtime system can determine for each access and synchronization, the protocol that is associated with the accessed data. Once the protocol of an access can be identified, the runtime system can dispatch events to the appropriate protocol handlers automatically.

In Section 3.1, we stated that declarations can be conservatively marked shared, even if the data to which they refer are strictly local (allocated on the stack or by Ace_Malloc). If a programmer takes advantage of this feature, the compiler may insert runtime system interface calls around accesses to data that are not shared. To handle these accesses correctly, the dispatch mechanism must differentiate between local and shared addresses. As described earlier, *Ace* addresses contain a bit that distinguishes shared and local data (Figure 4.4). Before performing the actions described below for protocol dispatch, the runtime system checks whether the address accessed is local. If the most significant bit of the address is not set, then the runtime system dispatches the event to a null handler. If it is set, the address is in shared memory, and the access is dispatched to the protocol associated with its corresponding space. This mechanism ensures that if necessary, accesses can be marked conservatively as shared.

The space abstraction implies that the protocol associated with a datum is that of the space from which it was allocated. Therefore, to determine the protocol of a datum, the runtime system must first determine the space from which it was allocated, and then, the protocol associated with that space. For barrier operations, the space is an argument to the operation, but the runtime system must still determine the address of the function to call. In this section, we describe how this information is derived by the *Ace* runtime system. We discuss how it can determine the space of any address, and then, show how the runtime system can determine the current protocol associated with a space.

On the CM-5, recall that when a space is created, each space is assigned a memory arena for allocating data. The memory allocation mechanism ensures that data from different spaces are not mistakenly allocated from the same coherence unit by allocating memory aligned to page boundaries to the space memory arenas. All locations in a page, therefore, belong to the same space. At system initialization time, a *space-page table* is allocated to store the space identifiers of pages in the system. On an access to shared data, the entry corresponding to the page to which the data belongs is retrieved from the space-page table. If the entry is zero, the runtime system contacts the home processor, that is, the processor that allocated the data, to obtain the space identifier of the space associated with the data. The larger 64-bit address of the Cray T3E allows for a simpler and more efficient implementation. The address of an object contains the space identifier of the space from which the object was allocated, which can be isolated by bit arithmetic on the address.

When dispatching to protocol handlers on an access or synchronization to data in a space, the runtime system first derives the identifier of the space associated with the event. Once it has obtained the space identifier, it can access the *space table* array to determine the address of the space structure corresponding to the identifier. The appropriate protocol handler can then be found by accessing the `methods` field of the space structure. In this fashion, given a space identifier, the runtime system has a handle to the associated protocol and the appropriate protocol handlers. The pseudocode for the protocol dispatch process on the CM-5 is shown in Figure 4.9.

4.4 Protocol Development and Addition

In our discussions so far, we have treated protocols as abstract objects that maintain the consistency of data. This section describes what constitutes a protocol, how new protocols are

```
/∗Local/Global check∗/
if (global address & 0x80000000 == 0)
    return
endif
/∗Check page entry of local space page table∗/
page_no = global address / PAGE_SIZE;
if (space_page_table [page_no] == 0)
    Send message to home requesting space number.
    Wait till reply arrives.
    space_page_table [page_no] = reply;
endif
space_id = space_page_table [page_no];

/∗Get address of space structure associated with space_id ∗/
space_structure = space_table [space_id];

/∗Invoke appropriate event handler ∗/
space_structure→methods [event] (global_address);
```

Figure 4.9: Pseudocode for determining the space of an access on the CM-5.

integrated into the system, and the mechanisms used by protocol writers to develop them. For the most part, *Ace* places few restrictions on the implementation of protocols. For example, for communication, a protocol could use any architectural mechanism available on the underlying platform.

The ability to change the protocol of a data object implies that allowing each protocol to implement customized cache state lookups and address translations is not viable. Since the protocol of a data object may change dynamically during the execution of a program, it is essential that the state of the object and the address translation method be uniform across protocols. The *Ace* runtime system provides a hardwired address translation mechanism and a caching substrate. Protocols must use these mechanisms.

We begin our discussion by describing the structure of a protocol and the steps that must be performed to extend the system with a new protocol (Section 4.4.1). We then introduce the *Ace* caching model (Section 4.4.2), which includes a discussion of the address translation mechanism, and the routines available to protocol writers for manipulating the state of objects. We describe the communication model in Section 4.4.3. The next section, Section 4.5, describes in detail, the implementation of the protocols in the *Ace* standard protocol library.

4.4.1 What is an *Ace* Protocol?

Ace supports *full access control*, that is, it allows protocol writers to specify the actions that execute before and after accesses and at synchronization points. A protocol, therefore, consists of a set of handler functions, one for each of these events. Each of these routines, except for the barrier routine, takes as an argument the address of the location that is being accessed (or locked/unlocked). The barrier routine takes as an argument the space for which the barrier operation is to be performed.

A protocol writer also develops a routine that is invoked when a protocol is associated with a space using Ace_NewSpace or Ace_ChangeProtocol. This routine is used to initialize the state of a protocol. Finally, a protocol can initialize the state of allocated data by specifying a handler that is invoked when data is allocated from an associated space. Table 4.2 summarizes the handlers defined by protocols and their naming conventions.

Table 4.2: Event handlers defined by a protocol and their naming conventions. $*$ denotes the protocol name.

Event	Argument	Name
Before read	*address_t*	$*$_StartRead
After read	*address_t*	$*$_EndRead
Before write	*address_t*	$*$_StartWrite
After write	*address_t*	$*$_EndWrite
Lock	*address_t*	$*$_Lock
Unlock	*address_t*	$*$_Unlock
Barrier	*space*	$*$_Barrier
Initialize space and protocol	*space*	$*$_FillSpace
Initialize allocated data	*address_t*	$*$_Create

A protocol writer develops handlers for each of the events and registers them with the protocol's initialization handler ($*$_FillSpace). As mentioned previously, the protocol's initialization routine registers its handlers by invoking the internal routine _Init_Space with the addresses of the protocol's event handlers as arguments. If desired, the protocol writer may specify the null handler, _Null_Hndlr, or default protocol handlers for events.

Once a protocol's handlers are developed, the protocol is registered with the system by running a Tcl/Tk [71] script. The script appends relevant information to a system configuration file that is used by the *Ace* compiler to determine the protocols available to the system. To summarize, adding a protocol to the system is relatively simple. Once event handlers are developed by the protocol writer, a script is run which gathers relevant information about the protocol. The protocol can then be used by programmers.

Figure 4.10 gives an example of protocol addition. The protocol writer specifies the name of the protocol and the events (access and synchronization points), for which non-null handlers have been defined. The compiler derives the name of the initialization routine for the protocol by concatenating the name of the protocol with _FillSpace (for example, Update_FillSpace). When a program associates this protocol with a space, Update_FillSpace is invoked, which subsequently, registers the protocol's event handlers. The last field in the figure specifies whether calls to the protocol can be optimized (Section 5.3 discusses this parameter in greater detail).

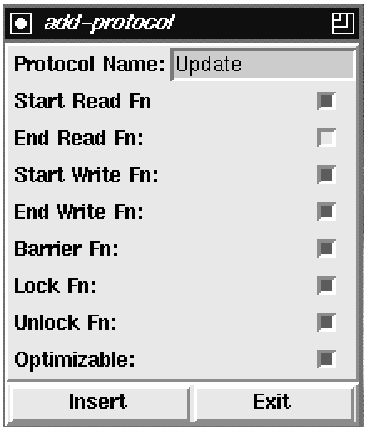


Figure 4.10: Adding a protocol.

4.4.2 Caching Model

To ensure that data objects have a uniform state description regardless of the protocol associated with it, the *𝒜ce* runtime system handles the low-level details of caching objects. It determines the address in local memory where a cached copy of a shared object resides, and provides a routine for translating global addresses. The protocols are responsible for manipulating the states of data and for communicating data between processors appropriately.

The *𝒜ce* runtime system supports a directory-based caching scheme [60, 91, 98]. In this scheme, a directory entry is allocated for each coherence unit at its *home* processor, that is, the processor that allocated it from the processor's private memory. Cached copies are allocated memory at each processor when necessary. We begin this discussion by presenting an overview of the caching model of the *𝒜ce* system. We then describe the routines for directory state management and the mechanism for address translation.

Cache State

In shared-memory systems, the cache state lookup may be performed in various ways. In some systems, custom hardware is used to the check the state of a global address in the cache or in a directory in parallel with the execution of an access [53, 57, 61, 76]. The benefit of this approach is clear; the check is transparent to the program and has no cost.

In systems where hardware support is not available, one can simulate its effect by leveraging virtual memory mechanisms. In Shared Virtual Memory (SVM) systems [62], each page is considered a coherence unit and a page fault on data is treated as an access to invalid data. While no overhead is apparent to the program when data is valid, the cost of an access to invalid data (a page fault) can be substantial. In certain cases, it may be better to circumvent the virtual memory mechanism by performing the state lookup in software [99]. To allow for the implementation of shared virtual memory, the operating system must support primitives such as those described by Appel and Li [9] that allow programmers to specify handlers that execute on page faults and manipulate the state of pages.

On certain platforms, shared virtual memory may not be an option because the underlying machine does not provide the operating system support necessary (as on the CM-5 and the T3E) or one wishes the system to operate on a finer granularity than the coarse page-based granularity of SVM's. In these cases, distributed shared-memory systems perform lookups of cache state in software. Two recent systems, Shasta [83] and CRL [47], reflect two contrasting methods of implementing cache state lookups in software.

At system initialization, Shasta (and Blizzard [85]) allocates a table at each processor that has an entry for the state of each cache block in shared memory. The memory overhead of this scheme can be substantial — with 64 processors and 64-byte cache blocks and 1-byte state per cache block, the size of the state tables in the system is equal to the total amount of shared memory available.

Virtual memory support is essential for the efficient implementation of this mechanism. Physical memory for virtual pages of the state table is allocated only on demand. When a page in the state table is first mapped to a physical page, the runtime system must be invoked so that it may initialize the state of the page appropriately. For this, the operating system must support a routine such as mprotect that allows user-level specification of handlers that are invoked at page faults. The memory overhead of this state table scheme is untenable on systems without appropriate virtual memory support, such as the CM-5 or T3E, or where a process's virtual memory is limited. If the addresses accessed by a processor are clustered together, it is possible that only a few pages of the table will be mapped on each processor, and the physical memory requirements for the state table will not be too high. The advantage of this scheme is that lookups can be implemented efficiently by using bit arithmetic on a global address to derive the address of the corresponding state table location.

In contrast, CRL takes advantage of the fact that each processor will probably access only a small fraction of remote data. It conserves memory by allocating space to hold the state of an object only when it is first accessed. The location of the state of each object is stored in a hash table and the state of an address can be obtained by performing a hash lookup on the address. Certain features of the CRL programming model make this a reasonable scheme. First, coherence units can be of arbitrary size and are not broken up into smaller cache block units. Second, pointer arithmetic on global addresses is not allowed. These two factors ensure that the number of addresses seen by the system is small, and hence, the number of collisions in the state hash tables is small. The efficiency of the hash table lookup degrades in the presence of pointer arithmetic where arbitrary addresses may be accessed.

In *Ace*, we use a simple but effective technique to combine the benefits of the two approaches mentioned above. We use an indexed table to store the state of remote data (Figure 4.11 depicts the table layout on the CM-5). On the CM-5, the top-level table stores the addresses of tables that store the states of 512 64-byte cache blocks. The second-level tables are allocated lazily on the first access to a block in their address range. On the Cray T3E, due to the larger address space, an additional level is necessary to reduce the memory overhead. With this mechanism, in the worst case when each processor accesses almost all data in shared memory, the memory overhead is comparable to that of the Shasta implementation. In the common case, when processors access mainly locally allocated data and some remote data, considerable memory is saved at the cost of a few extra operations.

Figure 4.11 also depicts the layout of an *Ace* cache entry on the CM-5. The structure of a cache entry on the T3E is similar except that since addresses are 64-bits, the address field is eight bytes long, and therefore, a cache entry is sixteen bytes long. When an object is allocated by Ace_GMalloc or is accessed for the first time, a cache entry for the object is created and initialized by the runtime system according to the pseudocode in Figure 4.12. *Ace* provides a routine, _Cache_Find, that given an address, returns the cache entry of a block. The implementation of this routine is similar to that of the insertion routine except that it assumes that the cache entry already exists. We now describe the layout of cache

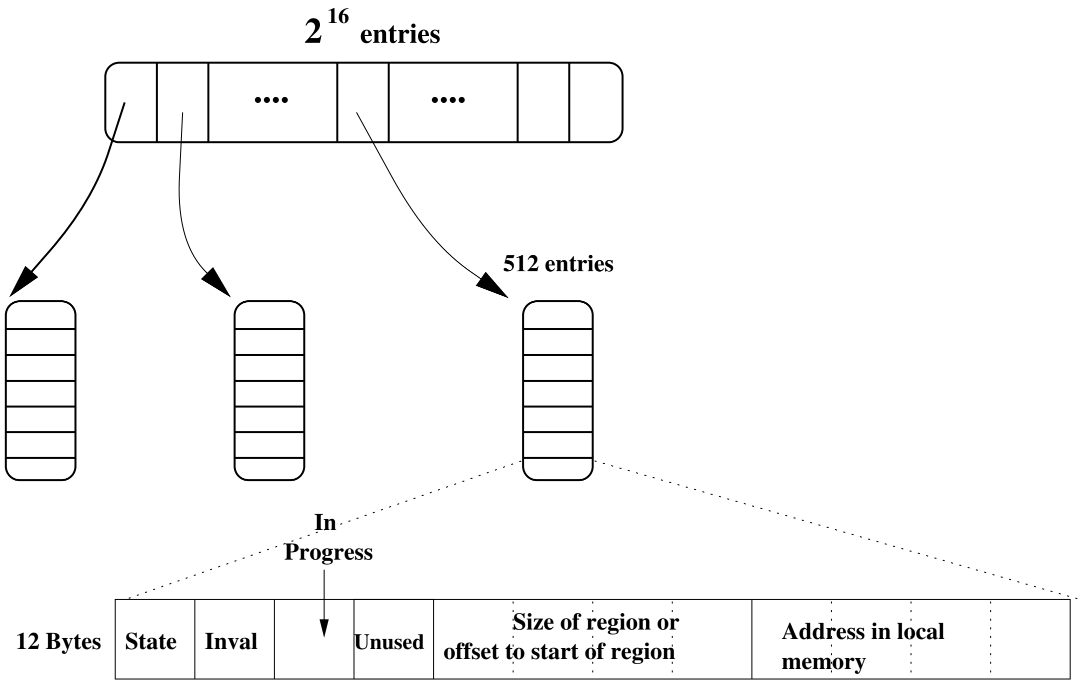


Figure 4.11: Indexing method for cache state lookup and structure of a cache entry (internal type, Cache) on the CM-5. The Cray T3E has one extra level of indexing.

```
#define NUM_BLOCKS 512
#define BLOCK_SIZE 64
index = address / (NUM_BLOCKS * BLOCK_SIZE);
if (cache_index_table[index] == NULL)
    cache_index_table[index] = allocate_level();
    Zero the memory pointed to by cache_index_table[index].
endif
level2 = cache_index_table[index];
offset = address % (NUM_BLOCKS * BLOCK_SIZE);
offset /= BLOCK_SIZE;
if (level2[offset] is not initialized)
    /* Allocate space for local copy */
    local_copy = allocate_local_copy();
    for each block in coherence unit
        Initialize cache entry.
    endfor
endif
```

Figure 4.12: Pseudocode for the insertion of a cache object on the CM-5.

entries and the routines provided for their manipulation. Table 4.3 summarizes these routines. In the table, note that the internal type for cache entries is Cache, and that for *Ace* addresses is address_t.

The first byte of a cache entry contains the local state of the object. The constants ReadWrite, ReadOnly, and Invalid are predefined by the system for use by protocols. The CACHE_TAG and CACHE_SET_TAG routines are used to access this field. The second byte is used by some protocols to record the fact that an invalidation has arrived. The third byte is used by protocols to mark that the protocol is busy executing an operation. The CACHE_INVAL and CACHE_INPROGRESS routines are used to access these bytes. The fourth byte of the first word is left unused.

Since *Ace* coherence units may be of variable size, a protocol must be able to derive the size of any given object. This information is necessary to determine the size for data transport or to make modifications to the state of an object as a whole. The second word in the cache byte is used for this purpose. If a block is the first cache block in a coherence unit, the size field contains the number of bytes in the unit. Otherwise, it contains a

Table 4.3: Routines for manipulating cache entries.

Routine	Description
Cache _Cache_Find (*address_t*)	Return cache entry
CACHE_TAG (*Cache*)	Return cache state
CACHE_SET_TAG (*Cache*, *state*)	Set cache state
CACHE_INVAL (*Cache*)	Return cache invalid byte
CACHE_SET_INVAL (*Cache*, *inval*)	Set cache invalid byte
CACHE_INPROGRESS (*Cache*)	Return cache inprogress byte
CACHE_SET_INPROGRESS (*Cache*, *in_progress*)	Set cache inprogress byte
CACHE_SIZE (*Cache*)	Get cache size
CACHE_ADDR (*Cache*)	Get local cache address
REMOTE_CACHE_FIND (*processor*, *address_t*)	Return cache entry at remote processor (*Only on T3E*)
REMOTE_CACHE_WRITE (*processor*, *address_t*, *Cache*)	Write cache entry at remote processor (*Only on T3E*)

negative offset, which specifies how far, in terms of cache blocks, the starting block of the coherence unit is from the current block. Given an address, the size of the corresponding coherence unit can be obtained by invoking CACHE_SIZE, which is implemented as given in Figure 4.13. Finally, given a cache object, the CACHE_ADDR routine returns the corresponding local address.

On the CM-5, the natural programming abstraction is to *push* data, that is, to modify or obtain remote data, one sends an active message to the remote processor that either modifies the data or requests them. On the Cray T3E, since a processor can operate on remote data without interrupting the remote processor, it is easier (and more efficient) to *pull* data. Modifications are done by reading the remote data, performing the modifications locally, and then writing the data back. As such, on the CM-5, updates of cache state are performed by sending a message to the remote processor to perform the update. On the T3E, however, routines for reading the remote state of an object and writing it back are necessary. The REMOTE_CACHE_FIND and REMOTE_CACHE_WRITE routines serve this purpose.

Directories

As mentioned previously, the *Ace* caching model is directory-based. The implication is that a system-wide state is stored for each coherence unit at its home processor. Directory

```
proc CACHE_SIZE (address_t address)
    Cache entry = _Cache_Find (address);
    if (entry→size > 0)
        return entry→size;
    else
        address += entry→size ∗ BLOCK_SIZE;
        CACHE_SIZE (address);
    endif
endproc
```

Figure 4.13: Pseudocode for determining the size of a cache object.

entries are handled in a similar fashion to cache entries. Figure 4.14 shows the structure of the directory table on the CM-5. The T3E directory table structure is similar, but the constants are different. At each processor, a directory index table is used to store the directory entries for the coherence units allocated at that processor. New coherence units are entered into this table by the Ace_GMalloc routine after allocating memory for an object. Unlike the cache table, the directory table need only handle addresses that are allocated from its memory. As a result, the directory table is significantly smaller than the cache table.

Table 4.4 lists the routines available to protocol writers for manipulating directories. As shown in the table, the internal type for directories on the CM-5 is Directory. The T3E directory entry layout is exactly as that on the CM-5. On the T3E, the DIR_REMOTE_FIND and DIR_REMOTE_WRITE routines obtain and update remote directories. The DIR_LOCK and DIR_UNLOCK routines can be used to ensure that only one processor updates a directory at a time. The implementation of most of these routines is straightforward. We concentrate on discussing the method for managing the sharers of an object.

The *Ace* implementation for managing the sharers of an object is borrowed from that in Tempest [76], and is similar to that described by Weber [96]. If the number of sharers of an object is less than the DIR_NUM_PTRS constant (in our case, five), the processor identifiers of the sharers are stored as distinct bytes in the directory entry. Otherwise, the sharer field represents the sharers as a bit vector. The DIR_ADD_SHARER routine inserts sharers into

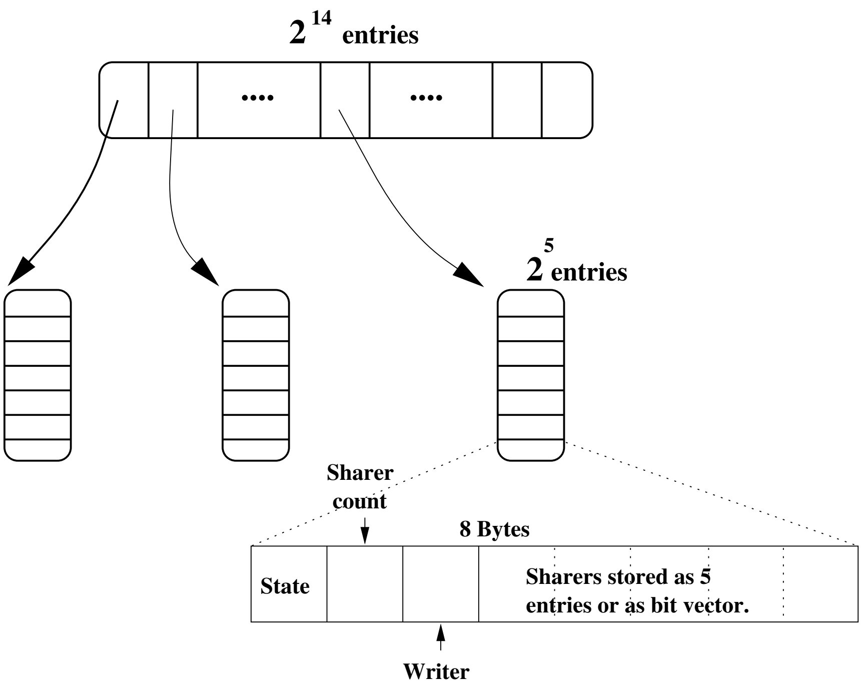


Figure 4.14: Layout of a directory entry and directory index table on the CM-5.

Table 4.4: Routines for manipulating directory entries.

Routine	Description
DIR_STATE (*Directory*)	Get state of directory.
DIR_SET_STATE (*Directory*, *new state*)	Set state of directory
DIR_COUNT (*Directory*)	Get number of sharers.
DIR_SET_COUNT (*Directory*, *count*)	Set number of sharers
DIR_PTR (*Directory*, *num*)	Get sharer number *num*
DIR_BITVEC (*Directory*)	Get sharer bitvector
DIR_SET_BITVEC (*Directory*, *bitvec*)	Set sharer bitvector
DIR_WRITER (*Directory*)	Get the writer
DIR_SET_WRITER (*Directory*, *writer*)	Set the writer
DIR_LOCK (*Directory*)	Lock a directory
DIR_UNLOCK (*Directory*)	Unlock a directory
DIR_ADD_SHARER (*Directory*, *id*)	Add processor to directory's sharers.
DIR_REMOTE_FIND (*id*)	Get remote directory *(Only on T3E)*
DIR_REMOTE_WRITE (*id*, *Directory*)	Write remote directory *(Only on T3E)*

a directory entry, converting from one format to another appropriately. When accessing sharers, protocol writers must explicitly check the count of a directory to determine whether to use DIR_PTR or DIR_BITVEC.

Address Translation

On hardware shared-memory systems, address translation is done in hardware, and hence, is transparent to a program. Several distributed-memory platforms, such as the CM-5, IBM SP-2 [44], and networks of workstations [8, 83], do not have a hardware-supported shared address space. On some of these machines, it is straightforward to simulate a shared address space. Using virtual memory techniques, the virtual address space is divided among the processors so that each processor is nominally the "home" of a range of the address space. The address space is also divided into shared and private components. For example, in Shasta [83], all addresses above a predefined address are designated to be shared and the range of virtual addresses above this limit is divided equally among the processors. Using a virtual memory mechanism such as mmap, cached copies of data are placed at the same virtual address on every processor.

Certain machines, for example, the CM-5 and T3E, do not provide the virtual memory support necessary to ensure that cached copies are accessible by the same address on each processor. Therefore, before accessing a cached copy of remote data, it is necessary to translate its global address into the physical address in local memory where it resides. Since a map may have to be performed on each access to global data, the translation of addresses can cause significant overhead. For example, in CRL, for each access to global data, a hash table lookup must be performed.

The *Ace* runtime system optimizes the translation process by managing the global address space carefully. As mentioned previously, the *shared* bit (Figure 4.4) distinguishes private data from shared data. When the address that is mapped is private or is local to the processor, its location in local memory can be derived quickly by using bit arithmetic to extract the lower 25 bits of the address on the CM-5 (lower 43 bits on the T3E). Since it is fairly common for a processor to access data allocated by it, the address optimization, derived from the technique used in Olden [79], can reduce the cost of mapping substantially. When the address accessed is remote, we use the same indexed table as that used in the cache state lookup to determine the local address of its cached copy. As mentioned previously (see Figure 4.11), for each coherence unit, we store along with its state, its address in local memory. Since the runtime-system interface separates map calls from protocol event calls, typically two state lookup operations must be performed for each access. The first finds the local address of the accessed data, while the second checks the state of the object.

4.4.3 Communication

The *Ace* runtime system provides protocols with a communication layer that implements many commonly used routines. The *Ace* communication layer provides an rpc-like call for obtaining data from remote processor (the requesting processor is suspended till the data arrives), for sending the data associated with an *Ace* global address to another processor (the data may be of arbitrary size), and for performing scatter/gather operations on lists of *Ace* global addresses.

On the CM-5, the *Ace* communication layer is built on top of the provided CMAML Active Message layer. The functionality of the *Ace* communication layer can be simulated on message-passing systems that do not support Active Messages by using asynchronous messages. A special message tag is used to indicate that the message sent contains the address (or name) of the function that is to be invoked at the remote processor. Each processor issues unconditional receives (receives that accept any tag) periodically, and dispatches to the appropriate handler if it receives a message with the special message tag.

A key issue in the design of the communication layer on the CM-5 is deciding whether to use interrupts or polling to handle message arrival. Interrupts are expensive on most machines. In addition, the unpredictable nature of interrupt-handler invocations makes it harder for a protocol writer to determine if a protocol implementation is correct. While polling for messages is potentially more efficient, it is difficult to determine when and where one should poll; polling too often can be more expensive than using interrupts, polling too rarely may lead to deadlock. It is impossible to determine *a priori* the appropriate times at which to poll since the communication behavior is often data dependent.

In our implementation, we have used interrupts to drive the receipt of messages. Since protocols cannot control when messages are received, the communication layer provides routines that allow protocols to buffer messages that they wish to defer, and retrieve them when they are in state in which they can handle the messages. Table 4.5 lists the routines provided by the *Ace* system.

As mentioned before, on the T3E, processors do not send messages to affect the state of objects, but manipulate remote memory directly. *Ace* runtime system uses the T3E's shmem library to implement communication. The _Request and the _Store routines are essentially wrappers around this library. Since processors do not send messages to other processors, it is not necessary to provide routines for blocking messages.

On the CM-5, active message handlers are guaranteed to execute atomically. It is possible to use this atomicity to ensure that updates to an object's state by two handlers do not interfere with each other. Since on the T3E, a remote processor may update a processor's state without the processor's knowledge (using REMOTE_CACHE_WRITE), some mechanism is necessary to ensure the atomicity of updates. If a processor requires a protocol handler to execute atomically with respect to other handlers in the protocol, it can use the ATOMIC_START and ATOMIC_END routines for this purpose. If every handler is delimited by these calls, it is guaranteed that at the most one handler will modify data at a processor at any given time. These routines are implemented using the atomic swap operations provided in the shmem library. These calls, however, are expensive, and their use must be avoided where possible.

4.5 Standard Protocol Library

The intent of this section is to provide a high-level description of the semantics and implementation of the protocols in our standard library. An understanding of these protocols is helpful in putting our experimental results in Chapter 6 in the proper context. While the implementations of these protocols on the CM-5 and the T3E differ because of variances in the underlying architecture and programming environment, the logical structures of the protocols are similar on both platforms. We focus on the implementation of these protocols on the CM-5 because the Active Message programming abstraction used on the CM-5 is more portable than the T3E programming model. In our experience, deriving the corresponding T3E versions of these protocols is relatively straightforward. We be-

Table 4.5: Routines for communication.

Routine	Description
_Request (*home*, *address_t*)	Request data from processor. On CM-5, Blocks until data arrives
_Store (*proc*, *address_t*)	Store data at processor, and on CM-5, wake it up
_Sleep	Block until a hander calls _Wakeup
_Wakeup	Wakeup blocked process
_SendData (*to*, *address_t*, *fn*)	Send data to processor and invoke *fn* *(Only on CM-5)*
_Send (*to*, *fn*, ...)	Send active message to processor with up to four arguments *(Only on CM-5)*
_SendBulkData (*to*, *address_t* $*$, *count*)	Send data specified by an array of addresses to processor
_BlockMsg (*address_t*, *rqstor*, *type*)	Block a message. *(Only on CM-5)*
_CheckBlockedMsgs (*address_t* $*$)	Retrieve a blocked message *(Only on CM-5)*
ATOMIC_START (*processor*)	Start critical section at processor *(Only on T3E)*
ATOMIC_END (*processor*)	End critical section at processor *(Only on T3E)*

gin this section by describing the implementation of the default sequentially consistent, invalidation-based protocol (Section 4.5.1). We then discuss two protocols, UPDATE (Section 4.5.2) and STATIC (Section 4.5.3), which support different flavors of an update protocol. Lastly, we outline the semantics and implementation of a protocol, SYNCH, which is well suited to synchronization variables.

4.5.1 Default Protocol

The default protocol is the workhorse of the *Ace* system, and as such, its efficiency is integral to the performance of the system. Our experience with this protocol has been that a simple design that optimizes the common case of read and write hits is best. We have found that a straightforward implementation of strict sequential consistency, that is, linearizability, allows for the simplest implementation. There are certain implementations of sequential consistency, for example, the one suggested by Afek *et al.* [2], that are looser in the sense that they allow a protocol to avoid performing actions on write as long as they ensure that memory is in a consistent state before reads. Such protocols have more complicated sequential consistency implementations that are not well suited to handling the common case of read and write hits. As a general-purpose protocol, neither invalidation-based nor update-based sequentially consistent protocols have any advantage in communication over the other. Since sequentially consistent update protocols involve more complexity than invalidation-based protocols, we choose the latter for our implementation. The default protocol, as most other protocols in the *Ace* system, is extremely streamlined — it is approximately 325 lines of C code.

Pseudocode for the write event handlers in the default protocol on the CM-5 is provided in Figure 4.15, and the code for the active message handlers called by these event handlers for handling directory actions is shown in Figure 4.16; the code for read events is similar. While the active message routines execute atomically upon invocation, that is, they run uninterrupted, the event handlers receive no such guarantee. There must be some mechanism in the start-of-write event handler to ensure that the state of the object does not change after it is checked (possibly due to an invalidation), until the write operation completes.

One possible technique of achieving mutual exclusion in event handlers is to disable

```
proc SC_StartWrite (address_t id)
    Cache cc = _Cache_Find (id);
    /* All further invalidation messages will be held */
    CACHE_SET_INPROGRESS (cc, ReadWrite);
    if (CACHE _TAG (cc) == ReadWrite)
        return;
    endif
    CACHE_SET_TAG (cc, ReadWrite);
    Request permission from object's home processor.
endproc

proc SC_EndWrite (address_t id)
    Cache cc = _Cache_Find (id);
    CACHE_SET_INPROGRESS (cc, FALSE);
    if (CACHE_INVAL (cc))
        CACHE_SET_TAG (cc, Invalid);
        CACHE_SET_INVAL (cc, FALSE);
        _SendData(Home(id), id, Directory_RWAck);
    endif
endproc
```

Figure 4.15: Write event handlers for the default protocol.

```
proc Directory_RWAck (address_t id)
    Directory dir = _Dir_Find (id);
    _Store (current writer, id);
    DIR_UNLOCK (dir)
endproc

proc Directory_WriteRqst (int rqstor, address_t id)
    Directory dir = _Dir_Find (id);
    unsigned state;
    DIR_LOCK (dir) /* Blocks messages if necessary */
    state = DIR_STATE (dir);
    DIR_SET_STATE (dir, ReadWrite);
    if (state == ReadOnly)
        Invalidate all readers.
    else
        Invalidate current writer, and set current writer to rqstor.
    endif
endproc

proc Invalidate_RW (int home, address_t id)
    Cache cc = _Cache_Find (id);
    if (CACHE_INPROGRESS (cc) == ReadWrite)
        CACHE_SET_INVAL (cc, TRUE);
    else
        CACHE_SET_TAG (cc, Invalid);
        _SendData(home, id, Directory_RWAck);
    endif
endproc
```

Figure 4.16: Directory handlers for the default protocol.

interrupts when the event handler is invoked, and re-enabling interrupts in the end-of-write routine. On most architectures, and specifically on the Thinking Machines CM-5, disabling and enabling interrupts are system calls. The overhead of performing this action on every access is prohibitive. Our start-of-write event handler (Figure 4.15, Lines 1–10) uses a more efficient technique. It sets the INPROGRESS byte in the state information of the accessed object (Figure 4.15, Line 4). This ensures that the local state of the object does not change until this byte is reset in the end-of-write handler.

If the state of the object in the start-of-write routine is Invalid or ReadOnly, a message is sent to invoke the Directory_WriteRqst handler (Figure 4.16, Lines 7–22) at the home processor. The home processor first checks whether it is busy servicing another request (Line 10). If it is, it blocks the incoming request until it is free to process it. If the object is currently in ReadOnly state, all the readers are invalidated; otherwise, it sends a message that invokes the Invalidate_RW (Figure 4.16, Lines 24–32) active message handler at the current writer.

If the processor at which the invalidation routine is called is currently busy with an operation on the object, it sets the INVAL flag and quits. Otherwise, it returns an up-to-date copy of the data to the home processor and invokes the Directory_RWAck procedure at the home processor (Figure 4.16, Lines 1–5). This routine forwards a copy of the data to the new writer of the object. The home processor then services any messages that may have been blocked on the object. Once a processor's write request is satisfied, it proceeds with the write operation until it invokes the end-of-write handler (Figure 4.15, Lines 12–20). The end-of-write handler checks to see whether an invalidation message was received while the write was being performed, and if so, invalidates the local copy and forwards the data to the home processor.

A special optimization is implemented when a write requester already has read permission on the object. In this case, since the requester possesses the most recent copy of the object, data transfer is unnecessary. The home processor waits till invalidations to the other readers are complete before informing the requester that it can proceed with the write. Reads are handled similarly to writes. If the data is not available locally, a message is sent to the home processor. The home processor downgrades data appropriately, and returns a fresh copy of the object to the requester. When an invalidation is received during a read operation, unlike in the case of writes, it is acknowledged immediately. The asymmetry is due to the fact that an acknowledgement on a write must return the most recent copy of the data, which is unavailable till the write completes.

The Cray T3E implementation executes exactly the same protocol actions. In the CM-5 implementation, the directory routines are executed by sending an active message to the home processor. The T3E, however, is better suited to "pull-based" communication, where a processor operates asynchronously on another's memory. In the T3E implementation, all protocol handlers are executed on the processor that initiates the protocol request. To ensure mutual exclusion between modifications of an object's state, processors invoke the ATOMIC_START and ATOMIC_END calls (see Table 4.5) in the start-of-write and end-of-write handlers.

4.5.2 UPDATE

In general, there are two mechanisms by which a processor may be informed that its copy of an object is stale: a message is received that orders the processor to invalidate its copy or an updated version of the cached object is received. The appropriate mechanism to use in a protocol depends on the sharing pattern it is meant to handle. Sometimes, for example, in certain producer-consumer problems, there is a high degree of stability in the sharing pattern. The set of processors that access a given object remains relatively constant over time. On writes to an object, it is more efficient to update all sharers of the object directly, rather than invalidating them and forcing them to request an updated copy explicitly. On the other hand, when the sharers of an object are not predictable or stable, update protocols may result in extraneous transfer of objects to processors that no longer require the data.

The UPDATE protocol is designed for situations where the set of sharers of an object changes gradually. In addition, it assumes that strict consistency of the data is unnecessary. The protocol is release consistent and assumes that there is a single writer of data between synchronization points. Updates to objects are not propagated to sharers instantaneously, but are delivered by the next synchronization point. The protocol defines handlers for synchronization, start-of-read, and start-of-write events, and is approximately 230 lines of C code.

```
1  proc ForwardData (int rqstor, address_t id)
2      Space sp = _FindSpace(id);
3      Sharer_List *sl = (Sharer_List *) sp→aux;
4      Add src and id to the sharer list, that is, sl.
5      _Store (src, id);
6  endproc
7
8  proc Directory_ReadRqst (int rqstor, address_t id)
9      Directory dir = _Dir_Find (id);
10     int writer = DIR_WRITER (dir);
11     DIR_SET_WRITER (dir, rqstor);
12     Send message asking current writer to forward data.
13 endproc
14
15 proc Directory_WriteRqst (int rqstor, address_t id)
16     Invalidate all current readers and writer.
17     DIR_SET_WRITER (dir, rqstor);
18 endproc
19
20 proc Update_Barrier(Space sp)
21     DoUpdates (sp);
22     Wait till all updates are complete.
23 endproc
```

Figure 4.17: Important handlers for the UPDATE protocol.

Pseudocode for the UPDATE protocol on the CM-5 is presented in Figure 4.17. The handlers for start-of-read and start-of-write events are similar to that of the default protocol. On a read or write miss, a request message is sent to the home processor. On a read miss, however, instead of invalidating the current writer, the home processor forwards the request to it (Line 12). The writer fulfills the request and appends the requesting processor and object address to a sharer list (Line 3–4). Since the update protocol may be used for different spaces and each space may have a different sharer list, the sharer lists are stored in the aux fields of the respective spaces. At a subsequent barrier associated with a space, the writer uses its sharer list to determine where to send up-to-date versions of objects.

On a write request on an object, the home processor sends invalidation messages to the current writer and readers (Line 16). The writer invalidates its copy of the object and forwards it to the new writer. The invalidation of readers allows the new writer to build a new list of sharers for that object. Typically, in applications that use the update protocol, the writers of objects change as infrequently as the sharers. Over time, however, the sharer list becomes stale and many unnecessary messages may be sent. Therefore, instead of forwarding the sharers of an object when its writer changes, we require that the new writer build a less stale sharer list anew.

In the T3E version, the sharer list for an object is maintained at its home, not at the current writer. At each barrier, processors obtain an up-to-date copy of each object they own from its current owner and propagate updates to all current sharers of the object. This approach is better on the T3E because with the ability to read remote memory without interrupting the remote processor, a home processor can obtain an up-to-date copy of data from their current writer efficiently. On the CM-5, the home processor must interrupt the writer to request the data. In this case, it is more efficient to store the sharer list at the writer and have the writer propagate the updates directly.

4.5.3 STATIC

Significant effort has been devoted in the past to the understanding of static irregular applications. The defining characteristic of this class of applications is that the access patterns are irregular and data-dependent, but do not change over the course of the execution of the application. Typically, these applications are structured as iterations over a data structure. The communication pattern is set early, typically in the first iteration of the program. The access patterns of subsequent iterations mirror that of the first iteration. Examples of static irregular applications include EM3D [25], multigrid applications [4], and CHARMM [43].

In general, an *inspector-executor* model is used to handle the communication in static irregular problems efficiently [28]. In this model, an *inspector* executes a skeleton of the main loop of the application to determine the communication pattern of the application. Once the access pattern is determined, the inspector attempts to derive the communication strategy best suited to the pattern. The *executor* uses this strategy during the execution of the loops. Since the access patterns do not change across iterations, the time spent in calculating the best communication strategy is amortized over the execution of several iterations.

We achieve an effect similar to the inspector-executor model in *Ace* by using a technique derived from that introduced by Falsafi *et al.* [34]. Our custom protocol for this access pattern, called STATIC, is in a sense, a generalization of the UPDATE protocol described earlier. The protocol assumes, as is normally the case for static irregular applications, that the execution uses *owner computes*, that is, all data is updated only by the processor that owns it [80].

Pseudocode describing the important handlers of the protocol on the CM-5 is provided in Figure 4.18. The protocol builds sharer lists in the first iteration of the application, that is, before any barrier has been called on a space associated with the STATIC protocol. In the first iteration, all reads to remote data will cause a miss in the start-of-read protocol handler. The miss results in a request message being sent to the home processor. At the home processor, the requesting processor and object address are stored in a sharer list before the request is fulfilled. At subsequent barriers, the sharer list is used to propagate updates to the appropriate processors.

Unlike the UPDATE protocol, the STATIC protocol is guaranteed that an object is written only by its home processor. These writes are always legal, and therefore, start-of-write and end-of-write events are discarded. In addition, since the sharer lists do not change over time, they are only built during the first iteration. Subsequent iterations will not experience any cache faults; the data are updated before they are accessed. Since the sharer list is invariant after the first iteration, the STATIC protocol can process this list at the first barrier and use bulk transfer to perform updates (Line 26). This step involves setting up buffers at senders and receivers to hold the values, and setting up the bulk transfer channels (senders and receivers must agree on the order in which objects are placed in the buffers). Since this processing is performed only once, its cost is amortized over the execution of subsequent iterations. With the UPDATE protocol, the expense of processing bulk transfer lists each iteration nullifies the benefits of bulk transfer. The T3E version is similar. The bulk transfers are implemented using the system-provided shmem_put routine. It may be more efficient to use the scatter/gather capabilities of the Cray T3E [7] to implement bulk transfer. We have not yet investigated this approach. The STATIC protocol is about 300 lines long.

```
proc Directory_ReadRqst (int rqstor, address_t id)
    Cache c = _Cache_Find (id);
    Fulfill request. Send data to rqstor.
    /* Add entry to sharers list */
    AddEntry (rqstor, id);
endproc


proc Static_StartRead (address_t id)
    unsigned home = Home(id);
    if (home != _ace_self_addr)
        Cache c = _Cache_Find (id);
        if (CACHE_TAG (cc) == Invalid)
            CACHE_SET_TAG (c, ReadOnly);
            Request data from home processor.
        endif
    endif
endproc


proc Static_Barrier (Space sp)
    /* Get sharer list for this space. */
    Sharer_List *sl = (Sharer_List *) sp→aux;

    if first iteration
        Process sharers list and determine the set of objects to
            be sent to each processor.
    endif
    DoUpdates (sl);
    Wait till all updates have been received.
endproc
```

Figure 4.18: Key handlers for the STATIC protocol.

4.5.4 SYNCH

In parallel applications, a single variable is often used to synchronize the order of execution at processors. An example is the head of a task queue or a counter variable. The default protocol is not well suited to accesses of this sort. A key observation that can be used to optimize accesses to these variables is that each write and read is delimited by locks. In this case, it is more efficient to couple the transfer of data with the locking mechanism. SYNCH is designed for data that meet the criteria that all accesses to the data are delimited by locks. The pseudocode for this protocol on the CM-5 is provided in Figure 4.19. The protocol constitutes 103 lines of C code. The T3E version is similar and uses the shmem library's atomic memory operations to ensure mutual exclusion and to obtain data.

Since all accesses in this protocol are surrounded by locks, event handlers at any of the accesses are unnecessary. The only routines provided are custom lock and unlock handlers. Since the transfer of data is implicitly associated with obtaining a lock on an object, the lock handler is essentially the same as the start-of-write operation in the default protocol (Figure 4.15), and is not shown in the SYNCH pseudocode. If a processor does not have lock permission, it requests it from the home processor, which in turn forwards the request to the current owner of the lock. If the current owner is not using the lock, it immediately transfers the data to the requester. Otherwise, it waits until an unlock operation on the object is completed before forwarding the data.

It may appear that the default protocol could use a similar implementation, and transfer data with locks, thus obviating the necessity for a SYNCH protocol. If, however, both locks and access event handlers manipulate the state of objects, inconsistencies may occur. For example, an end-of-write action in the default protocol will set the INPROGRESS byte to FALSE. If a lock is currently active, the clearing of this variable implies that the next request for the lock will be granted, even though an unlock operation has not been performed. The SYNCH protocol receives a guarantee that a lock operation will be called before any access and an unlock operation after it. It, therefore, can discard read and write events and perform the operations necessary for these accesses at the lock and unlock operation. The default protocol must define read and write operations. To avoid conflicts with these access handlers, the lock operations are forced to be as general as possible.

```
proc ReleaseLock (int rqstor, address_t id)
    Cache cc = _Cache_Find (id);
    if (CACHE_INPROGRESS (cc) == ReadWrite)
        /* have not received lock yet or still using lock */
        CACHE_SET_INVAL (cc, rqstor | 0x80);
    else
        CACHE_SET_TAG (cc, Invalid);
        CACHE_SET_INVAL (cc, FALSE);
        _Store(rqstor, id);
    endif
endproc

proc GetLock(int rqstor, address_t id)
    Directory dir = _Dir_Find (id);
    int owner = DIR_WRITER (dir);
    DIR_SET_WRITER (dir, rqstor);
    Call ReleaseLock on current owner.
endproc

proc Synch_Unlock(address_t id)
    Cache cc = _Cache_Find (id);
    CACHE_SET_INPROGRESS (cc, FALSE);
    if (CACHE_INVAL (cc) != 0)
        int rqstor = ((unsigned char) CACHE_INVAL (cc)) &  0x80;
        _Store(rqstor, id);
        CACHE_SET_TAG (cc, Invalid);
        CACHE_SET_INVAL (cc, FALSE);
    endif
endproc
```

Figure 4.19: Key handlers for the SYNCH protocol.

4.6 Discussion

The implementation of the *Ace* programming model is partitioned cleanly between the compiler and the runtime system. The compiler/runtime-system interface is based on full access control — the compiler inserts calls to runtime-system routines around accesses to data. This is not, however, the only interface possible. For example, the interface could be more spare — the compiler inserts calls to a read runtime-system interface routine for loads and to a write routine for stores. The runtime-system routines would dispatch to protocol event handlers before and after the access and perform the access.

On the other hand, the interface could be more intricate — the runtime system could reveal the communication routines and the cache state lookup to the compiler for optimization. Our choice of an interface was motivated by the fact that it had been shown to be efficient in runtime systems such as CRL [47]. Determining the relative benefits of various interfaces would be an interesting extension of our work.

In implementing a caching model, the ability to manage the address space of the platform facilitates implementation. With such a mechanism, it is possible to arrange for cached copies of data to have the same address as the global address. Furthermore, one can develop an easy correspondence between an address and the location of its state. In its absence, as on the CM-5 and T3E, more complex schemes such as those described in Section 4.4.2 are necessary. Since these caching mechanisms are executed often, they may affect the performance of a program significantly. It is, therefore, imperative that the effect of these accesses be minimized.

Neither the Cray T3E programming model nor that of the CM-5 holds a great edge over the other in terms of ease of implementation of *Ace* or development of *Ace* protocols. The ability to access any location in the machine's memory in the T3E is of great help. Since, however, *Ace* cannot ensure that cached copies of data are located at the same address, much of the advantages of direct remote access are not present (one must still be able to calculate where a remote object lies). In terms of efficiency, the ability to access remote data without incurring an interrupt is a great advantage of the T3E. When debugging programs, however, we found it difficult to isolate bugs on the T3E. We found Active Message programs easier to debug because of the guarantee that handlers are atomic.

The mechanism for protocol dispatch is executed often as well. The process performs several indirect accesses, for example, to the method field of a space structure, that are hard to implement efficiently. In the worst case, communication may be required. As such, optimizing dispatches to protocols are of great importance as well. The next chapter studies how a compiler may optimize the performance of interface calls and protocol dispatches.

Aside from the caching substrate and the address translation mechanism, *Ace* places no restrictions on protocols. Where possible, a protocol is free to use architectural mechanisms. For example, Typhoon [76] is an architecture with some support for customizable protocols. Programmers may specify handlers that are to execute on access faults to cache blocks, that is, when the data is not in an appropriate state for the access. The cache state lookup is performed in hardware. For a protocol that requires control only at access faults, as in the default protocol, a protocol writer could use the Typhoon-provided mechanisms to register the protocol's handlers with the hardware and specify null handlers to *Ace*. Accesses to data managed by this protocol will be handled directly by the hardware. Other data in the application could use protocols that are implemented using *Ace* mechanisms. Similarly, on FLASH [53], one could use this mechanism to choose between custom protocols that are implemented in software and system protocols that use hardware support.

On these machines, a problem arises in the fact that the hardware mechanisms may manage the state of an object in a manner not compatible with that used by the *Ace* runtime system. When a program changes the protocol of an object between one that uses the hardware mechanism and one that uses the software technique, there must be a means for handling the transition between the two so that the data is manipulated into the correct "base" state. The issue of the appropriate transition mechanism requires further study.

The implementation that we have described is geared towards systems on which software shared memory is necessary. Our implementations of spaces and protocol dispatch, however, do not make assumptions about the shared-memory model, and can be used on shared virtual memory systems as well, or on Typhoon and FLASH, as described above. The *Ace* model, however, is not well suited to hardware shared-memory systems with no support for message passing or software shared-memory protocols, such as shared-memory multiprocessors. In these systems, the overhead of customizable shared memory relative to the cost of the hardware mechanism is likely to be too high to be of use.

The burden of developing protocols rests on the programmer. As mentioned in Section 2.5, it is possible to use systems such as Teapot [21] to lessen this load. An interesting issue, not tackled by Teapot, is the set of communication mechanisms sufficient for the implementation of most protocols. The set provided by *Ace* is not complete either. For example, one would wish a system to provide routines that perform common operations such as invalidating a cache block. We found custom protocol development to be relatively easy compared to writing general purpose protocols such as the sequentially consistent invalidation-based protocol. The difficulty in general purpose protocols is that all possible state transitions must be handled. In custom protocols, assertions can often be made that reduce the state space of the protocol. As a result, fewer situations need to be considered by the programmer. For example, in certain dynamic update protocols, a writer need not acquire exclusive access before proceeding with a write as long as the result of the write is propagated to all sharers.

All roads in *Ace* pass through the runtime system. Great care has been taken to make it as spare and efficient as possible. The standard protocols have been streamlined as is evident from the fact that all of them have fewer than 500 lines of code. Yet, on message-passing machines, the viability of our software shared-memory approach is contingent on our compiler and its optimization phases, the subject of the following chapter.

Chapter 5

The *Ace* Compiler

In the absence of hardware support, either virtual memory mechanisms [62] or purely software techniques [83] are used to implement shared-memory systems. The use of compiler techniques in improving the performance of these systems, especially software distributed shared-memory, has been relatively unexplored. Compiler techniques appropriate for a system with underlying shared virtual memory support has been studied to a certain extent [32]. With respect to software shared-memory systems, however, past research has focused on linktime techniques [83]. At linktime, though, important information, such as type information, is unavailable. A compiler has access to more information about the semantics of an application. Compile-time implementations of software distributed shared-memory present greater opportunities for optimizations and promise to be more efficient.

In these systems, the compiler ensures that accesses to global shared memory are handled correctly and efficiently. In *Ace*, as described previously in Section 4.2, the runtime system presents an interface to the compiler that encapsulates the functionality of implementing the *Ace* programming model into a few key operations. The *Ace* compiler transforms programs written using a shared-memory abstraction into code annotated with interface calls that, in conjunction with the runtime system, executes correctly on distributed-memory message-passing architectures. Furthermore, the compiler assists in the support of *full access control*, that is, it ensures that protocol handlers are invoked at appropriate points during the execution of a program. The compiler is also responsible for optimizing the use of the interface calls to ensure efficient performance.

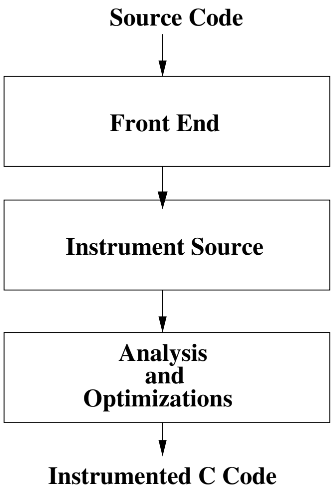


Figure 5.1: Structure of the *Ace* compiler.

The *Ace* compiler is structured as a series of phases (see Figure 5.1), the output of which is C code instrumented with *Ace* runtime system calls. This output is then compiled on the host machine using a native compiler (e.g. `gcc`) and linked with the runtime system to obtain an executable. This chapter is structured into two halves. We first describe the components of the compiler that implement the *Ace* programming model (Section 5.2). We then describe the analyses and optimizations that are essential for good performance (Section 5.3). Finally, we conclude this chapter with a discussion of the strengths and weaknesses of the *Ace* compiler in Section 5.4. We begin by summarizing the important aspects of the compiler's intermediate format and implementation.

5.1 Intermediate Format

The compiler was built using the SUIF toolkit [97] from Stanford University, which provides a flexible compiler skeleton that facilitates the construction of research compilers. The toolkit supports an object-oriented implementation of a tree-based intermediate format. Each of the operators in the intermediate format, for example, load and store operators, are represented by classes. The internal implementation of these operators are hidden from the programmer, and high-level routines are provided to manipulate their fields. For example, routines are provided to set and access the address field of a load operation. A procedure consists of an ordered list of expression trees and a program is represented by a list of file objects, each of which contains a list of procedures.

In SUIF, symbol tables and type information are represented by objects as well. This information as well as the intermediate format is designed to capture the high-level semantics necessary for easy implementation of analysis phases. In addition to the low-level constructs found in most intermediate formats, the SUIF format includes high-level constructs for loops, conditional statements, and array accesses. Furthermore, all SUIF nodes can be annotated with information from high-level analyses that can be used in more low-level analyses.

The SUIF compiler skeleton is organized as a set of phases, where each phase is a distinct program. Each phase reads in the result of the previous phase, performs analysis and/or transformations, and then, writes out the results to a file. The ANSI C front-end is based on a modified version of lcc [36], and converts C programs into the SUIF intermediate format. A back-end is provided to transform SUIF intermediate code back into C code. Compiler developers may insert phases between the front-end and the back-end to perform analyses and transformations.

The *Ace* compiler uses a modified SUIF front-end. New phases were written that process the SUIF intermediate representation of an *Ace* program to perform instrumentation, interprocedural dataflow analysis, and optimizations. The rest of the chapter is devoted to describing each of these phases in detail.

5.2 Implementing the *Ace* Programming Model

As mentioned previously, most of the work in supporting the *Ace* shared-memory model is performed by the runtime system. It is the compiler's responsibility to invoke the runtime-system interface calls appropriately. In this section, we describe the front-end of the compiler, which prepares an *Ace* program for the remaining phases of the compiler. We then describe the instrumentation phase of the compiler.

5.2.1 Front End

In most respects, the *Ace* compiler front-end is similar to that of a standard C compiler. Extra functionality, however, handles the requirements of the *Ace* programming language. Recall that all shared data must be allocated in the heap; so, one cannot have shared variables, only pointers to shared data. Table 3.1 (page 40) gives examples of valid and invalid declarations. During parsing, the front-end ensures that no variable is declared shared, that is, all shared data are accessed through pointers. The shared declaration is treated by the front-end similarly to the const and volatile quantifiers in ANSI C. Before a variable is entered into the symbol table, the front-end evaluates its type to verify that the shared quantifiers have been applied correctly. The compiler reports an error if it determines that the declaration is incorrect.

The compiler also ensures that *Ace* types are used properly. The address of a space variable or a protocol cannot be taken. Space variables can only be assigned to each other and may not be cast to any other type. Protocols are constants and cannot be assigned to any variable (recall that in Section 3.1.2, we stated that all protocols are predefined by the system, and *Ace* provides a mechanism that allows addition of new protocols to this set).

In the generated SUIF intermediate code, all instructions that access shared data (all dereferences to pointers to shared data or stores to shared data) are annotated, so that later phases can isolate these accesses easily. Figure 5.2 gives an example of the intermediate code generated by the front-end from a sample *Ace* fragment. Each operator annotated shared may access a location in shared memory. Furthermore, each operator is qualified by a type that often determines the granularity of an access. For example, a type of int for a load implies that a word will be loaded, and a type of double implies that a doubleword will be accessed. In the example, the field annotation is used to preserve the high-level information that the pointer arithmetic is actually a dereference of a field in a structure.

5.2.2 Instrumentation

After an *Ace* program is converted into the SUIF intermediate format by the modified front-end, the next phase of the compiler instruments the intermediate representation of the program with calls to runtime-system interface routines. The SUIF toolkit provides a routine

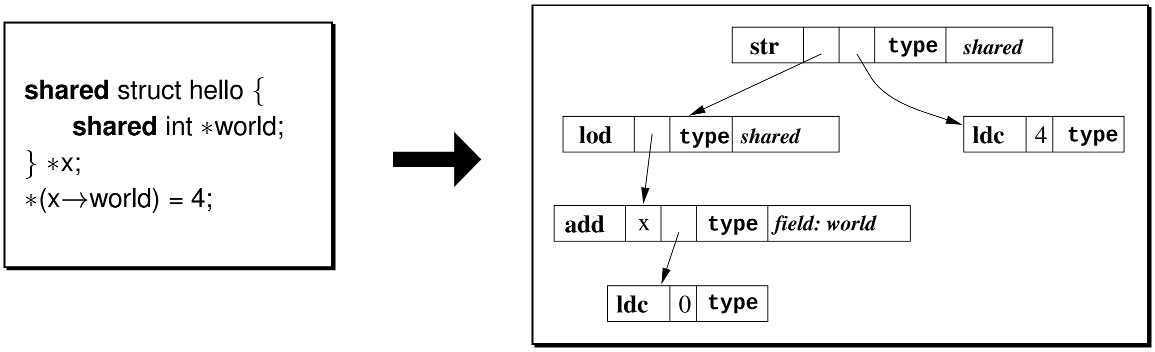


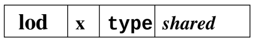
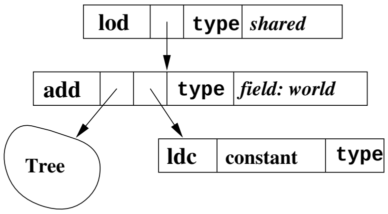
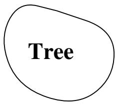
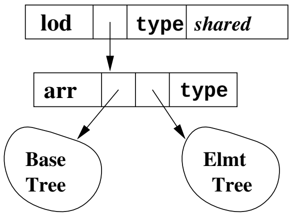
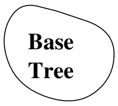
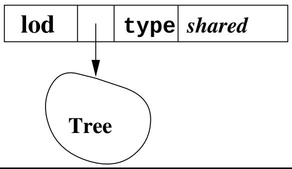
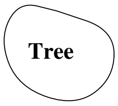
Figure 5.2: Intermediate format representation for ∗(x→world) = 4. **lod** = Load contents of address, **str** = Store value in address, **ldc** = Load constant, **add** = Add two values. *type* refers to the type of the instruction. Fields in italics denote annotations.

that traverses all expression trees in a procedure's intermediate representation recursively, and invokes a user-specified function on each node in the trees. The *Ace* compiler uses this routine to visit all nodes in the intermediate representation of a program and replaces each load and store operation that is annotated **shared** with a set of expressions that calls the appropriate *Ace* runtime-system routines, using temporary variables as necessary. We describe, in this section, the process for loads (stores are handled similarly). Figure 4.8 (page 61) gives an example of the C equivalent of the instrumented intermediate code generated from an *Ace* code fragment.

Recall that the compiler/runtime-system interface routines take as argument the address that is being read or written (Section 4.2). The address argument is not used directly by the runtime system or *Ace* protocols, but is used to determine the correct protocol to handle the access or to obtain the state of the coherence unit to which the address belongs. In the current implementation of the runtime system, certain routines, such as state lookup of an address or determining the size of the coherence unit to which it belongs (Section 4.4.2), are more efficient if the argument passed is the starting address of the coherence unit.

Since the path through a protocol routine is generally more efficient when it is passed the starting address of a coherence unit, the *Ace* compiler, when instrumenting an access with runtime-system interface calls, first attempts to derive the starting address, or *base*, of the coherence unit associated with the address being accessed. Table 5.1 shows how the base is computed for various accesses. For example, for $A[i]$, the base is A.

Table 5.1: Base addresses for various types of loads. Trees denote any expression tree.

Input	Description	Base
	Load symbol `x`	`x`
	Load field of structure	
	Load element of array	
	All other loads	

An implicit assumption in using the base of an access as the argument for runtime-system interface calls is that the base and the address accessed belong to the same coherence unit. This is true if applications do not make any assumptions about the relative physical layout of distinct coherence units. In other words, programmers do not use pointer arithmetic to access arrays beyond bounds or generate new addresses by adding arbitrary constants to 𝒜ce addresses. The correctness of such irregular pointer arithmetic is highly implementation-specific, and is not guaranteed under 𝒜ce.

Once the base address is computed, the load operation is replaced by a set of expressions as shown in Figure 5.3. The figure shows a load operation that is part of some expression tree in a procedure. The compiler first removes the destination tree, $\mathcal{D}$, from the procedure's

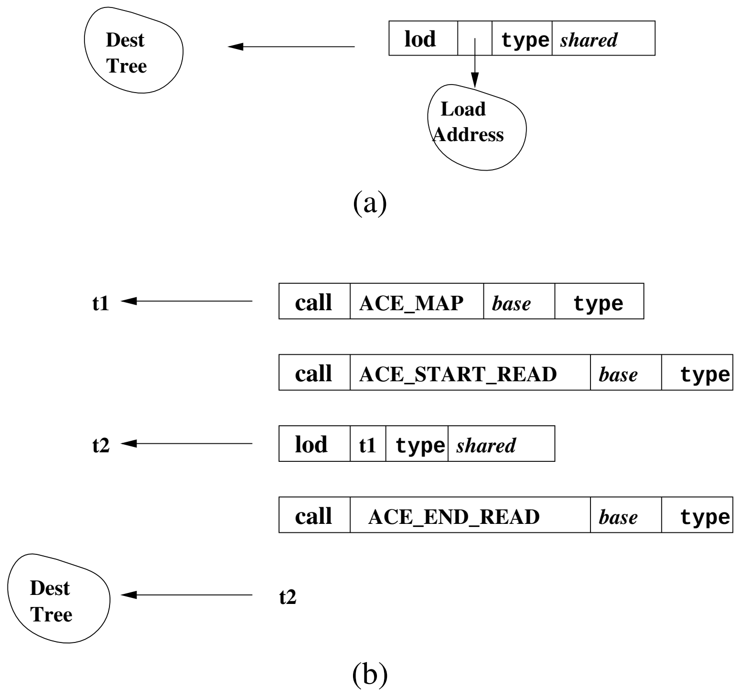


Figure 5.3: (a) Load expression before instrumentation. The load operation is part of some destination expression tree. (b) Set of expressions inserted in the place of the load instruction.

instruction list. In its place, it inserts an expression that calls ACE_MAP on the base of the load instruction, and stores the result in a temporary variable, say t1. The result of the ACE_MAP call is the address in local memory of the accessed global address. The next step is to generate an ACE_START_READ call on the base. When a program is executed, once this instruction is processed, t1 will contain a consistent value of the object being accessed. The compiler, therefore, generates an expression that performs a load on t1, the result of which is stored in another temporary, say t2. An expression that calls ACE_END_READ is then appended. Finally, the original tree, $\mathcal{D}$, to which the load belonged is re-inserted with the load operation replaced with an access to t2.

Stores to shared data are handled similarly by the $\mathcal{A}$ce compiler. In the SUIF intermediate format, each expression tree is guaranteed to have at the most one update to a memory location. Processing expression trees in a bottom-up fashion ensures that the instrumentation for the loads on which a store depends is inserted before the store is processed.

5.3 Optimizations

The overhead of executing the *Ace* runtime library calls for each access to global shared memory is a serious obstacle to achieving good performance with *Ace* programs. It is, therefore, necessary to implement optimizations in the *Ace* compiler that reduce the overhead of these library calls. Implementing optimizations in the *Ace* compiler, however, presents several challenges. In general, the semantics of *Ace* programs force compilers to make conservative estimates of the effects of a program's execution on memory.

The consistency model of an application determines the amount of transformation that can be performed on the program. With strict consistency models such as sequential consistency, many common optimizations, such as code motion, are not allowed [66]. A sequentially consistent protocol guarantees that the memory accesses made during the execution of a parallel program can be put into a partial order. A program transformation that moves an access above another may cause an execution that violates the partial order condition, even if the locations accessed are different. However, weaker consistency models, such as release consistency, allow the compiler to assume that code between synchronization points will not be affected by code executing on other processors. As a result, several common transformations that are used in sequential compilers, such as common subexpression elimination, can be applied to the program [52].

In *Ace*, the compiler is complicated by the fact that each program may have several protocols that support different consistency models. It is, therefore, important to be able to isolate, for each access, the protocols that may be invoked to handle that access, and the limitations on transformations placed by the consistency model of the protocol. The optimization phases implemented in the *Ace* compiler reorder accesses, and therefore, could potentially generate inconsistent executions with strict protocols. When, however, the order of accesses to different locations between synchronization points is unimportant, as in the case of release consistent protocols, the optimizations are safe.

As mentioned previously in Section 4.4.1, when registering a protocol, *Ace* protocol writers specify whether its semantics allow optimizations. A protocol writer specifies that a protocol is not optimizable if accesses must appear atomic with respect to other accesses or if the order of accesses between synchronization points is important. We refer to protocols that allow optimizations as *safe* protocols. The information about whether a protocol is safe or not is stored in a system configuration file and is available to the compiler.

Before optimizing a program, it is necessary to determine for each access, the set of protocols that may be invoked to handle that access. An access to shared memory can be optimized only if the compiler can determine that for all possible executions, the access will be handled by safe protocols. The *Ace* compiler derives the protocols invoked to handle an access in two steps. First, for each access to shared memory, it identifies the set of spaces from which the accessed data may have been allocated. Second, it determines, at each global access, the set of protocols that are possibly associated with each space. The composition of these two sets gives, for each global access, the set of protocols that may be invoked at that point in the program. The compiler can then use the configuration file to check whether each protocol in the set computed for an access is safe.

For example, consider the dereference of the variable nd at Line 20 of the code fragment in Figure 5.4. In the example, tree_node and list_node point to data allocated from tree_space and list_space respectively. After the if-then-else branch (Lines 16–19), nd may be aliased with either tree_node or list_node, and therefore, may point to data allocated from either list_space or tree_space. At that execution point, tree_space is associated with the SC protocol, and list_space is associated with the UPDATE protocol. The set of protocols that may invoked at this point is therefore, { SC, UPDATE }. The *Ace* compiler annotates each access to shared data with the set of protocols that might be invoked at that point during the execution of the program.

The preceding example illustrates the analyses that must be performed to identify protocols correctly and precisely. In most applications, as in the example, the allocation of data is in a different procedure from their use. As a result, interprocedural analysis is necessary. We now describe the implementation of the interprocedural analyses that perform space identification and protocol identification, and then, examine the three optimizations phases of the *Ace* compiler (Section 5.3.3).

```
Space tree_space, list_space;
shared struct node *tree_node, *list_node;
shared struct node *nd;

void allocate_data () {
    tree_space = Ace_NewSpace (SC);
    list_space = Ace_NewSpace (SC);
    tree_node = Ace_GMalloc (tree_space, sizeof(struct node));
    list_node = Ace_GMalloc (list_space, sizeof(struct node));
}

void main () {
    allocate_data ();
    compute (tree_node, list_node);
    Ace_ChangeProtocol (list_space, UPDATE);
    if (...)
        nd = tree_node;
    else
        nd = list_node;
    printf("%d\n", nd->value);
}
```

space set = {tree_space, list_space}
protocol set = {(tree_space, SC), (list_space, UPDATE)}
protocols invoked = {SC, UPDATE}

Figure 5.4: Example of identifying the protocols that may be invoked to handle an access.

5.3.1 Space Identification

The *space identification* phase determines, conservatively and precisely, the set of spaces from which each datum in shared memory may have been allocated during any possible execution of the program. We can formulate the space identification problem as a reaching definitions dataflow analysis [5]. Since, however, *Ace* programs have parallel semantics, the reaching definitions problem has a different flavor.

On sequential machines, a definition is said to reach an instruction if there exists some path in the control flow graph between the definition and the destination instruction on which the definition is not killed. The explicit parallel semantics of *Ace* programs, however, imply that, in general, a compiler cannot determine the control flow of a program. At any given time during the execution of an explicitly parallel program, each of the processors may be at any instruction in the program. As a result, with respect to the effects of a program on memory, an instruction on a processor may be followed by an arbitrary instruction on another processor. When calculating dataflow information, such as the definition of the space of a variable through an access, a compiler must assume that the instructions executing on different processors *interfere*, that is, each instruction affects the information generated by the others. It may be possible to use the synchronization information to determine the regions of a program that cannot possibly execute concurrently (non-concurrency analysis) [64]. This analysis is helpful in limiting the set of instructions that may interfere with each other. While this analysis can provide some information about the control flow of a program, it is generally too coarse-grained.

Since in *Ace* programs, precise control flow information is unavailable, a flow-sensitive analysis, which relies on the control flow graph to calculate intraprocedural information, cannot be used. Unlike flow-sensitive analysis, flow-insensitive analysis does not require any flow information. Flow-insensitive analyses assume that the execution of the program may consist of arbitrary sequences of instructions, that is, any instruction may follow another. This is exactly the scenario present in explicitly parallel programs. Flow-insensitive analysis allows us to process explicitly parallel programs as if they were sequential programs. The *Ace* compiler uses a flow-insensitive analysis for the space identification phase.

In the space identification problem, each Ace_GMalloc call defines the space of a location in shared memory. We wish to find, for each access to shared data, the set of definitions to that location that could reach that point. The flow-insensitive nature of the analysis implies that the instructions in a program may be evaluated in any arbitrary order. As a result, the reaching definitions problem becomes trivial. Between any definition and use, there is always is an execution order in which a path exists where the definition is not killed (consider the evaluation order in which the use immediately follows the definition). The implication is that information is never killed — any definition of the space of a location in shared memory is visible to all accesses to that location. What is required for space identification, therefore, is a means of marking locations in shared memory with appropriate spaces, and determining the location in shared memory to which an access refers.

Since *Ace* is based on C, variables may be aliased with each other, and a definition of a location through one variable may affect an access through another. Once again, the imprecision of the flow-insensitive analysis implies that one cannot be certain of the location in memory to which an access refers. The evaluation order used by this type of analysis to derive an alias may not correspond to any feasible execution. To be conservative, however, flow-insensitive analysis must assume that two locations are aliased for the entire program if any instruction may create such an alias. To summarize, the space identification problem is one of identifying conservatively the sets of locations in shared memory that might be aliased with each other, and marking a set with the appropriate space if there exists an Ace_GMalloc instruction such that any of the locations in the set may refer to the result of the instruction.

The *Ace* compiler performs the alias analysis and the space identification concurrently using a flow-insensitive, alias analysis algorithm similar to that introduced by Steensgaard [90].[1] The algorithm uses a type inference scheme to build up a *storage shape graph* [23] in almost linear time and linear space in practice (theoretically, the bounds are quadratic [89]). At the end of the analysis, the storage shape graph is a finite, conservative estimate of all pointer paths into and through the heap that could arise through any execution of the program. The nodes in the storage shape graph are annotated with the set of spaces from which they may have been allocated. Given an access into shared data, the

[1]Our type inference algorithm was developed independently, but later versions of our algorithm were influenced by Steensgaard's work.

compiler can determine the spaces of the location accessed by simulating the access in the storage shape graph to obtain the node of the graph that is referred to by the access, and retrieving the annotations associated with that node.

We describe the essential steps of the analysis through the example in Figure 5.5. In the example, we show a sample program, and the results of the analysis after processing each line in the program. Initially, after Line 5, y and z are pointers to pointers to integers allocated from space_1 and space_2 respectively. The call to fn on Line 6 results in y being aliased with the formal parameter t. After Line 7, the contents of $\star$y and $\star$z point to the same location, and so, the nodes to which they point are merged. Note how the if-then control flow is ignored. Line 8 does not add any information to the graph. Finally, since y is aliased with the formal parameter t, Line 12 adds an edge from l to the location pointed to by $\star$y. After the entire program is processed, the compiler can follow the pointers in the storage shape graph to determine that $\star\star$y will access a location that is either in space_1 or space_2.

Implementation

We present a high-level overview of the implementation of the space identification algorithm. Our implementation has been, in many respects, influenced by Steensgaard [89]. Readers interested in an alternative and more detailed description of a type inference-based flow-insensitive analysis algorithm are referred to the paper cited above. In our description, we ignore the mechanisms for handling function pointers, which are derived from those in Steensgaard [89].

In this analysis, all high-level language type information in a program is ignored. Type inference over a custom domain of types is used to build a conservative estimate of a storage shape graph. The algorithm assigns types to locations and merges types when necessary. In the end, the space of any access can be found by evaluating its type in the computed solution. We first outline an implementation in which there are only pointer types in the type domain. We then describe how the implementation could be extended with more types for greater accuracy.

In our implementation, an associative set data structure, $\mathcal{A}$, stores the current types of the variables in the program, and initially, is set to be empty. Consider the case in which

```
1   shared int **y, **z, *l;
2
3   void main () {
4       *y = Ace_GMalloc (space_1, sizeof (int));
5       *z = Ace_GMalloc (space_2, sizeof (int));
6       if (...) fn (y);
7       else *y = *z;
8       **y = 4;
9   }
10
11  void fn (shared int **t) {
12      l = *t;
13  }
```

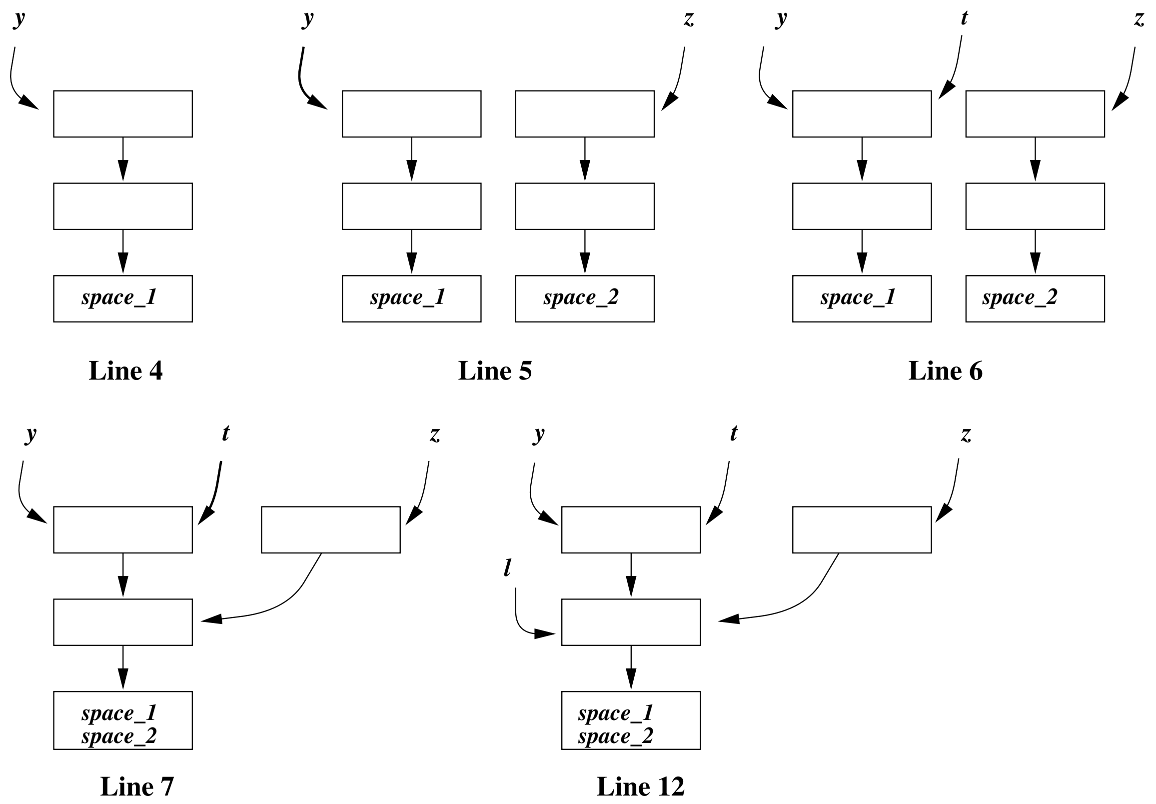


Figure 5.5: Example of process of deriving the set of spaces from which a location in shared memory may have been allocated.

the type domain consists of only a single type, τ, of the form: (τ_1, *space list*), where τ_1 can either be NULL or point to another type. Each type is assigned a unique type identifier. Two types are considered equivalent if they are described by the same type identifier (unlike the usual structural equality of types).

The algorithm processes each procedure in the program. The expression trees of each procedure are interpreted according to a set of rules. The rules ensure that if, after the execution of Statement S in a program, two locations x and y may be aliased, then, after evaluating Statement S under the rules, x and y will be represented by types $\tau_1 = (\tau, \ldots)$, and $\tau_2 = (\tau, \ldots)$, respectively. Furthermore, the rules ensure that if location x in memory may be allocated from a space s, then the space list for the corresponding type will contain s. The storage shape graph represented by these types is monotonic. Once two types are set to be equal, they will be equal in the final solution. Figure 5.6 list the evaluation rules for this type domain. In the figure, **deref** has the following semantics:

deref $(\tau_1, \ldots) \to \tau_1$

deref (NULL, ...) $\to$ **new** Pointer(NULL, NULL)

The rule for assignment states that if two locations in memory are assigned to each other, the result is to merge the types pointed to by the source and destination of the assignment. For an example of this process, please refer to Figure 5.5, and consider the effect of the actions of Line 7. In the figure, evaluating `*y = *z` merges the nodes referred to `*y` and `*z` in the storage shape graph. Types are implemented as fast union-find objects [92], and are merged using this process. The semantics of the **merge** operation is as follows:

merge $\tau_a = (\tau_1, space_1)$ and $\tau_b = (\tau_2, space_2) \to$ Perform union operation on τ_a and τ_b. Merge lists $space_1$ and $space_2$. If τ_1 and τ_2 are non-null, **merge** τ_1 and τ_2.

When two expressions, $expr_a$ and $expr_a$ are added or subtracted, either of the two expressions may be an address. The result of the operation, therefore, could be a pointer to any the locations pointed to by either $expr_a$ or $expr_a$. Of course, neither of the two expressions may be an address. When evaluating such an instruction, however, one does not know that either of the two operands will not be dereferenced in the future. The high-level types of the operands are insufficient since the operands may be cast to pointers at some further

eval $expr_a = expr_b$
  τ_1 = **eval** $expr_a$
  τ_2 = **eval** $expr_b$
  Merge **deref** τ_1 and **deref** τ_2

eval Ace_GMalloc(*space*, ...)
  τ_1 = **new** Pointer(NULL, *space*)
  τ_2 = **new** Pointer(τ_1, NULL)
  return τ_2

eval ret *expr*
  τ_1 = **eval** *current proc*
  τ_2 = **eval** *expr*
  eval $\tau_1 = \tau_2$

eval $*expr$
  return deref eval *expr*

eval $expr \rightarrow field$
  return eval $*expr$

eval $expr_a$ *op* $expr_b$
  τ_1 = **eval** $expr_a$
  τ_2 = **eval** $expr_b$
  τ_3 = Merge **deref** τ_1 and **deref** τ_2
  return τ_3

eval *symbol*
  τ = Look up *symbol* in associative set A.
  if (*symbol* not found)
    τ = **new** Pointer(NULL,NULL)
    Insert (*symbol*, τ) in A
  endif
  return τ

eval $fn(arg_1, arg_2, \ldots, arg_n)$
  for $i = 1 \ldots n$
    τ_i^f = **eval** formal argument i of *fn*
    τ_i^a = **eval** actual argument i of call
    eval $\tau_i^f = \tau_i^a$
  endfor
  return eval *fn*

eval $expr_a[expr_b]$
  return eval $*expr_a$

Figure 5.6: Evaluation rules for space identification assuming a single type. **deref** dereferences a pointer.

point. The analysis, therefore, assumes conservatively that the locations pointed to by the two types ought to be merged.

The major difference between our implementation and that of Steensgaard is that we must consider space information as part of a type annotation. A call to Ace_GMalloc generates a new pointer type which points to another pointer type that is annotated with the appropriate space. When the instruction that assigns the result of the Ace_GMalloc to an expression is evaluated, the space annotation will be passed along to the expression's type (the types of both source and destination of the assignment will be dereferenced and merged). Aliasing of that particular location with others will result in a merging of the spaces of both locations. In this fashion, the space list of a location is always a conservative estimate of the set of associated spaces.

When a symbol is evaluated, the associative list, $\mathcal{A}$, is first checked to see if a type has been entered for that symbol. If not, a new pointer type is generated and entered into the list. This process applies to global and local variables, and to function names. The type of a function is a placeholder for the return value of the function. When during the evaluation of a procedure, a **return** statement is encountered, the analysis merges the return value with the type of the function being processed. When a function call is evaluated, the first step in the analysis is to simulate the assignment of each actual parameter to the corresponding formal parameter of the function. The analysis then returns the type of the function. Since the return type is merged with the arguments of each **return** statement in the called function, the analysis can guarantee that the return type will summarize the effects of all possible returns from the function.

Finally, dereferences of expressions are handled in an intuitive fashion. Since the only types available are pointers, arrays and structures are considered as monolithic units. Any access to an array is handled as if it were an access to the first element of the array. A type that corresponds to an array captures the memory effects of all the elements in the array. In the one-type type domain, structures are handled similarly to arrays. The type corresponding to a structures represents the effects of all fields in the structure. At the end of this analysis, for any access to memory, we can simulate the access in the storage shape graph by running **eval** on the access. The space annotations on the resulting type is a conservative estimate of the types that could be associated with that location.

While the technique outlined above is simple and efficient, it fails to distinguish between fields of structures. Adding two additional types, Structure and Blank, to the type domain enables us to obtain more accuracy in our results. Blank types denote variables that are never dereferenced, and are also the default type for objects until the analysis can determine that they are either pointers or structures. Pointer types continue to describe objects that contain a reference to other types. Structure types refer to locations that are accessed as structures. If an object is used both as a structure and as a pointer, for example, s→field and ∗(s + x), then its type to set to be a pointer and its fields are coalesced into a single unit as in the previous version of this algorithm.

A structure type stores the types for each of its fields. When two structures with different fields are merged (for example, struct a ∗a_ptr = struct b ∗b_ptr), if there is a prefix of the field list that is common to both structures, then each of these fields are merged individually. The remaining fields are consolidated into a single unit and merged. Steensgaard describes this process, and provides details for the evaluation rules for structures in his paper [89].

5.3.2 Protocol Identification

At the end of the space identification phase, a conservative estimate of the set of spaces associated with any access can be determined by evaluating the access according to the rules in Figure 5.6. The space annotations on the resulting type is the desired solution. To identify the set of protocols that may be invoked to handle that access, the compiler must compute the set of protocols that may be associated with each space in the access's space set when the access is executed.

For protocol identification, it is desirable to have a flow-sensitive analysis. Without flow information, it is impossible to determine whether a definition of a space follows another. The set of protocols associate with a space at any point, therefore, would be assumed to be all protocols that are ever associated with the space. For example, in Figure 5.4, at Line 20, a compiler would be unable to determine that the definition to list_space at Line 15 follows the definition at Line 7, and would have to assume conservatively that list_space may be associated with both SC and UPDATE protocols. While a flow-sensitive analysis is useful, one would expect that the parallel semantics of $\mathcal{A}ce$ programs would force the use of a flow-insensitive analysis as in the space identification phase. We shall show that it is possible to use flow-sensitive analysis for protocol identification.

The problem of deriving the set of protocols that may be associated with a space at any point during the execution of a program can also be cast a reaching definitions problem. Space variables are defined by Ace_NewSpace and Ace_ChangeProtocol, and we wish to obtain, for each access to shared data, a conservative estimate of the space definitions that may reach that point. Assume that we have performed alias analysis on space variables, and we have a function, $\mathcal{A}$, such that $\mathcal{A}(v)$ returns all space variables in the program that *may be* aliased with variable v. The Gen and Kill sets for $\mathcal{A}ce$ statements in a basic block are as follows:

- S: *space_var* = Ace_NewSpace (*protocol*)

  $$\text{Gen}(S) = \{\forall x \in \mathcal{A}(space\_var),\ (x,\ protocol)\}$$

  $$\text{Kill}(S) = \{(space\_var,\ *)\}$$

- S: Ace_ChangeProtocol (*space_var*, *protocol*)

  $$\text{Gen}(S) = \{\forall x \in \mathcal{A}(space\_var),\ (x,\ protocol)\}$$

  $$\text{Kill}(S) = \{(space\_var,\ *)\}$$

- S: *Other statements* $\rightarrow$ no change.

Notice that, given alias information, the only statements that affect the dataflow information are Ace_NewSpace and Ace_ChangeProtocol calls. Fortunately, the semantics of these calls state that at any given point during the execution of the program, the protocol associated with a space is the same on all processors. Furthermore, these commands have the semantics of barrier operations — they appear to start and complete at the same time on all processors. Therefore, when a processor is executing one of these instructions, every other processor will be executing an instruction with the same effect, and hence, these instructions will not interfere. Between these statements, the space-protocol associations on one processor are not affected by any instruction executed on any other processor. The ramification of this lack of interference is that the sequential control flow graph of the program can be used to propagate space-protocol association information accurately.

The protocol identification phase is implemented as a standard context-insensitive, interprocedural reaching definitions analysis [5]. The alias analysis for determining if two space variables refer to the same space is performed concurrently with the space identification phase. A new type, Space, is added to the type domain. This type is similar to Blank in that it is never dereferenced. The rule shown below is added to the set of evaluation rules to handle calls to Ace_NewSpace:

eval Ace_NewSpace (*protocol*)
  return new Space()

Since the address of spaces cannot be taken, the alias analysis is straightforward. The only time that two space variables are aliased is if one is assigned to another directly, or through a parameter in a procedure call. Since spaces can only be assigned to other spaces, they will never be merged with any of the other types in the type domain.

At the end of the space identification phase, for each space, we can determine a conservative set of aliases. The set of protocols associated with a space at any instruction in the program can then be calculated using an iterative, forward propagation strategy. The Gen and Kill sets are used to calculate an approximation of the solution for each basic block. The In and Out sets for each basic block are as follows, where **pred** B refers to the set of predecessor basic blocks to B:

$$\mathsf{Out}(B) = \mathsf{In}(B) + \mathsf{Gen}(B) - \mathsf{Kill}(B)$$

$$\mathsf{In}(B) = \bigcup_{i \in \mathbf{pred}\, B} \mathsf{Out}(i)$$

A worklist of the procedures in the program that are yet to be evaluated is maintained by the analysis. Initially, the In sets of all procedures are set to be empty, and all procedures are added to the worklist. When the evaluation of a procedure changes the In set of another procedure p, p is added to the worklist. The analysis ends when the worklist is empty, that is, a steady-state solution is reached. Details on the implementation of such an algorithm can be found in Chapter 10 of Aho *et al.* [5].

When the context-insensitive analysis ends, we can determine for each instruction, the set of protocols that may be associated with a space at that point during any execution of the program. The protocol identification phase traverses the entire program, and annotates each runtime-system interface call, such as ACE_START_READ, with the set of protocols that be invoked to handle the shared-memory access associated with that access. As mentioned previously, this set can be computed by composing the spaces that may be associated with that access with the set of protocols associated with each space at that point.

5.3.3 Program Optimizations

In this section, we describe the optimizations performed by the *Ace* compiler. The *Ace* compiler moves interface calls out of loops where possible, merges redundant calls, and directly dispatches to protocols. The first two optimizations can be performed to an access only if all protocols which may handle the access are safe. At the end of the space and protocol identification phases, each runtime system interface call is tagged with the set of protocols that may invoked to handle the access associated with the call. For any given access, the compiler scans the system configuration file to check whether all the protocols that may handle the access are safe. If all protocols in an access's set are safe, then all optimizations to the access are permissible.

Moving Calls out of Loops

For an access to shared memory within a loop nest, the compiler determines the outermost loop to which the protocol calls associated with the access can be moved. ACE_MAP and ACE_START_$*$ calls are moved above a loop, while ACE_END_$*$ calls are moved below a loop. Please refer to Figure 5.7 for an example.

Protocol calls associated with an access to shared data can be moved out of a loop if it can be determined that the address accessed does not change within the loop. The analysis required to obtain this information is similar to loop invariance analysis. Standard normal loop invariance analysis attempts to determine whether the contents of a location is invariant across iterations of a loop. In contrast, our analysis is based on determining whether the address that is accessed (not its contents) changes across iterations. For example, in

```
for (i = 1; i < 10; i++)
    A_tmp = ACE_MAP (array);
    ACE_START_READ (array);
    y += A_tmp[i];
    ACE_END_READ (array);
endfor
```

```
A_tmp = ACE_MAP (array);
ACE_START_READ (array);
for (i = 1; i < 10; i++)
    y += A_tmp[i];
endfor
ACE_END_READ (array);
```

Figure 5.7: Example of moving calls out of loops.

Figure 5.7, the contents of array A change in the loop, but since the address A itself is not modified, the calls associated with the access can be moved.

The fact that this optimization is only performed for safe protocols implies that the relative order of accesses between synchronization points is unimportant. The consequence is that between synchronization points, the effects of accesses at other processors do not interfere with accesses at a processor. As a result, the sequential control flow graph between synchronization points can be used to perform analysis.

We perform intraprocedural analysis to isolate the candidates for code motion. The runtime-system interface calls associated with an access can be moved out of a loop if the address is accessed in every iteration of the loop, the address accessed is not modified in the loop, and there are no synchronization points in the loop. An address is assumed to be modified in the loop if it may be the result of a store, or if a procedure call is made in the loop. An address is not accessed in every iteration of the loop if it is surrounded by a conditional statement. Once the compiler determines that a protocol call can be moved out of the loop, its expression tree is deleted from the current loop's body. It is then inserted before or after the expression tree for the current loop in its parent's instruction list, depending on whether the instruction is to be moved above or below the loop.

Merging Redundant Protocol Calls

If between two synchronization points, there are two ACE_MAP calls to the same address, and the address is not modified between the two, the *Ace* compiler can reuse the result of the first one and remove the second. Furthermore, if the protocol actions associated

```
A_tmp = ACE_MAP (array);
ACE_START_READ (array);
z = A_tmp[0];
ACE_END_READ (array);
A_tmp = ACE_MAP (array);
ACE_START_READ (array);
for (i = 1; i < 10; i++)
    y += A_tmp[i];
endfor
ACE_END_READ (array);
```

→

```
A_tmp = ACE_MAP (array);
ACE_START_READ (array);
z = A_tmp[0];
for (i = 1; i < 10; i++)
    y += A_tmp[i];
endfor
ACE_END_READ (array);
```

Figure 5.8: Example of removing redundant calls.

with the two ACE_MAPs are both reads or both writes, the *Ace* compiler can use the highest ACE_START_$*$, and the lowest ACE_END_$*$, and remove the rest. Figure 5.8 provides an example.

Merging two calls effectively implies that the second access is visible to the system at the same time as the first, and is only possible if the order of accesses to different variables between synchronization points is unimportant. The analysis in this phase is exactly that of available-expressions analysis [5]. A basic block is processed to determine, for each instruction, the set of results of ACE_MAP calls that are alive at that point. The only statements that generate information in this analysis are those of the form $S : t_0 =$ACE_MAP (*address*), which add $\{(address, S, t_0)\}$ to the list of available expressions, where S is a pointer to the expression tree for the statement. Any update to a location kills all addresses that may be aliased with it. Procedure calls and synchronization kill all addresses.

The analysis processes each instruction in a basic block in order. When an ACE_MAP call is evaluated, the compiler first checks whether the argument of the call is one of the available expressions. If it is not, the expression is added to the list. If it is, then the ACE_MAP call is removed, and all references to its result are replaced by the result of the earlier ACE_MAP call. In addition, if the corresponding accesses are both reads or both writes, the ACE_START_$*$ call of the ACE_MAP being processed and the ACE_END_$*$ call of the earlier ACE_MAP are removed. The ACE_END_$*$ action of the earlier ACE_MAP is set to be that of the current ACE_MAP. Figure 5.9 provides the pseudocode for this analysis.

```
proc EvalBlock (Basic block)
    A = {NULL} /* Set of available expressions */
    for i = Each instruction in block
        if (i: t1 = ACE_MAP (address))
            if (address, i0, t0) ∈ A)
                Replace t1 in block with t0
                Remove instruction i from block
                if (i0 and i are both reads or both writes)
                    Remove ACE_START_* associated with i
                    Remove ACE_END_* associated with i0
                    The ACE_END_* associated with i is now associated with i0
                else
                    The ACE_START_* and ACE_END_* associated with i
                        are now associated with i0
                endif
            else
                Add (address, i, t1) to A
            endif
        else if (i: Call or Synchronization)
            A = {NULL}
        else if (i: other statements)
            K = set of locations that may be updated by statement
            for each location l in K
                Remove l from A.
            endfor
        endif
    endfor
endproc
```

Figure 5.9: Pseudocode for basic block analysis for merging redundant calls.

```
A_tmp = ACE_MAP (array);
ACE_START_READ (array);
z = A_tmp[0];
for (i = 1; i < 10; i++)
    y += A_tmp[i];
endfor
ACE_END_READ (array);
```

```
A_tmp = ACE_MAP (array);
UPDATE_START_READ (array);
z = A_tmp[0];
for (i = 1; i < 10; i++)
    y += A_tmp[i];
endfor
UPDATE_END_READ (array);
```

Figure 5.10: Example of direct dispatch. The compiler determines statically that the access to array will always be handled by the UPDATE protocol.

Avoiding Dispatching Overhead

In this transformation, the expression trees of each procedure are traversed. When a call to an *Ace* protocol dispatch routine is encountered, for example, ACE_START_READ, the compiler checks the instruction's protocol annotations. If there is a single protocol associated with that instruction, the dispatch call can be replaced with a direct call to the appropriate protocol event handler.

Recall that when a protocol is added (Section 4.4.1), the protocol's name and its list of event handlers are added to a system configuration file. The names of the event handlers are derived by concatenating the protocol name with the corresponding event. When a unique protocol is associated with an access, the compiler removes calls to protocol dispatch handlers and generates a call to the corresponding protocol event handler in its place. If a protocol defines certain handlers to be null handlers, then calls to that protocol action can be removed completely. Figure 5.10 gives an example.

5.4 Discussion

Compile-time implementations of software shared-memory have several benefits over link-time transformations. Shasta [83] supports fine-grain shared-memory by processing object files at linktime. However, much information that is available at the language level is lost by linktime. For example, Shasta attempts to merge (batch) redundant protocol actions by checking if two accesses use offsets from the same base register. Since type information is not available at linktime, merging is safe only if the offsets are separated by less than the lowest possible granularity of all data in the application. At the language level, using type information, it is easier to determine if two accesses belong to the same coherence unit. Link-time optimizations and optimizers are also less portable than compilers.

The SUIF toolkit was invaluable in implementing a compiler quickly. The SUIF front-end is a modification of lcc, which is well described by Fraser and Hanson [36]. As a result, we were able to modify the front-end to suit our purposes fairly easily. The ability to implement phases of a compiler as separate programs is of great use as well, since each phase could be developed and tested independently. The object-oriented intermediate format encapsulates much of the intricacies of code manipulation into methods.

The disadvantages of the SUIF compiler are its slow speed and large size. Once developed, it is difficult to distribute our compiler because of the sheer size of the SUIF package. All combined, our SUIF directory occupies about 100 MBytes of disk space. Since SUIF is designed as a research compiler, its building process compiles the code for all libraries and phases with the debugging flag set (the -g flag). The toolkit is not guaranteed to execute correctly if it is compiled with optimizations. Furthermore, since each phase in a compiler is a separate program, compilation of *Ace* programs requires the creation of several heavy-weight processes. The compiler is, therefore, quite slow. For a research compiler, however, trading speed for ease of development is acceptable.

The accuracy of the analyses implemented in the *Ace* compiler would be enhanced if we were to augment them with non-concurrency analysis [64]. The process of merging these kinds of analysis in our context is an area of future study. While the flow-insensitive information obtained from our analysis is imprecise, it is generally sufficient for our purposes. For the space-identification analysis, in *Ace* programs, it is rare for variables that point to data from one space to be aliased with pointers to another space. Therefore, while the alias information may be imprecise, the annotations on the merged nodes pointed to by aliases are generally precise.

In *Ace*, protocol writers specify whether a protocol may be optimized or not. Our loop motion and merging redundant calls optimizations were designed specifically for protocols with semantics similar to that of release consistency, while the direct dispatch optimization can be applied to any protocol. Instead of hardwiring the optimizations in a compiler and requiring programmers to determine the suitability of the optimizations to their protocol, a more natural approach would be for the compiler to determine the transformations appropriate for a protocol. A deeper understanding of the formal specifications of protocols is necessary before a compiler could derive automatically the best transformations for each protocol.

One optimization not implemented in the compiler that could possibly be of benefit is the inlining of protocol routines where possible. Furthermore, as mentioned before in Section 4.6, the compiler/runtime-system interface could expose more of the internal runtime-system routines to the compiler to allow for the ability to perform more optimizations. With respect to the optimizations currently implemented, interprocedural implementations of loop motion and merging redundant calls could be of use. The correct implementations of such analysis require further investigation. A discussion of the merits of the current optimizations and possible improvements to the analyses is deferred until the next chapter, where we examine the experimental results essential to understanding the efficiency of the compiler.

Chapter 6

Experiments

The utility of any parallel system is predicated on its performance. In this chapter, we examine the efficiency of our prototype system through a series of experiments. The purpose of these experiments is to expose the strengths and weaknesses of the *Ace* implementation. We also compare our system against a relatively new, efficient software distributed shared-memory system, CRL [47], to put our experimental results and *Ace*'s performance in perspective.

The experiments in this chapter fall into two categories. First, using micro-benchmarks, we identify the overhead of the key operations in the *Ace* runtime system. The true measure of a system's performance, however, are real applications. To this end, we use eight representative applications as measuring sticks. These benchmarks are used in several ways: we identify the access patterns in the applications, and the protocols that optimize communication performance, we study the improvements and speedups achievable using customizable protocols, and we examine the performance of the *Ace* compiler relative to hand-optimized code. We evaluate *Ace*'s performance on two architectures, the Thinking Machines CM-5 [59] and the Cray T3E [86].

The chapter begins with a description of the computing platforms used by our experiments (Section 6.1). Section 6.2 outlines the results obtained from the micro-benchmark experiments. We present a brief overview of the benchmarks used in our experiments in Section 6.3. We, then, validate the base performance of our system by comparing the efficiency of *Ace* with that of CRL (Section 6.4). The use of customizable protocols for our application set is studied in detail in Section 6.5. Our experiments in this section reveal the advantages of customizable protocols. The efficiency of the *Ace* compiler and of its optimizations is discussed in Section 6.6. Finally, in Section 6.7, we summarize the information gleaned from our experiments, and suggest possible refinements to improve the performance of *Ace*.

6.1 Experimental Platforms

The Thinking Machines CM-5 [59] is a message-passing machine with relatively efficient support for fine-grained message passing. Each processing node of the CM-5 is equipped with a 32 Mhz SPARC v7 processor rated at a peak performance of 32 Mips or 5 MFlops.[1] The experiments were run on a 128-processor CM-5 system running version 7.3 Final 1 Rev 3 of the CMOST operating system, with version 3.2 of the CMMD message passing library.

The interprocessor communication in *Ace* uses the CM-5's Active Message library (CMAML) [95]. For reference, the overhead of sending an Active Message request is approximately 2.03 μs, that of processing a reply is 1.90 μs, and the achievable bandwidth for large transfers (using CMAML_scopy) is 8.4 MBytes/s [54]. The CM-5 has a dedicated network for collective communication, and therefore, implements these operations efficiently. Broadcasts can achieve a bandwidth of 0.8 MBytes/s, reductions have a latency of approximately 18.3 μs, and barrier synchronization, a latency of 5 μs [54]. On the CM-5, all application code as well as the runtime system code was compiled with gcc -O2 (version 2.6.2).

The Cray T3E [86] is a distributed-memory machine with some hardware support for a shared address space. In contrast to the CM-5, communication is essentially a one-way operation; with hardware support, a processor can read or write any location in the machine's memory without any remote processor's intervention. The T3E also supports atomic increment and swap operations on memory. Each node in a T3E has a 64-bit 450Mhz DEC Alpha 21164 microprocessor, and is connected to the other nodes by a high bandwidth interconnect. Scott reports that the realized bandwidth on an early prototype of the T3E

[1]We did not use the vector units on the CM-5.

(using slower processors) for memory-to-memory copies for data transfers of size 4K bytes was about 300 MBytes/s [86]. On the T3E, all code was compiled using the native Cray C compiler with optimizations set at O2. All experiments were run on the 512-node Cray T3E machine at the Pittsburgh Supercomputing Center.

6.2 Micro-Benchmarks

The bottlenecks in the performance of the system can be isolated by considering the overhead of key primitives in the *Ace* runtime system. The micro-benchmarks used to isolate the cost of runtime system calls follow a common template. The operation that is to be measured is repeated in batches of steadily increasing numbers. For example, to measure the cost of mapping data, first the time taken to perform one map is measured, then the cost of performing two maps is measured, and so on, until we measure the execution time of n maps. Linear regression on the results of these experiments gives us the cost of the operation, as well as, the error bounds of the answer. We use these bounds to check the consistency of our results.

Table 6.1 outlines the results of our experiments for various runtime-system operations on 16-byte regions. The faster processors and interconnect in the Cray T3E results in the T3E performing significantly better than the CM-5. Recall that the address translation mechanism is optimized for maps of data that have the same owner as the mapping processor (Section 4.4.2). The table reveals the benefit of this optimization. The execution time of a map operation on the CM-5 drops significantly from 1.9 μs to 0.3 μs. A similar effect is seen on the T3E as well. Since maps are executed frequently during the execution of programs, these savings are extremely important.

The rest of the table refers to the cost of operations under the default sequentially consistent protocol. The cost of a read miss includes the cost of transferring the data to the requesting processor and is substantially higher than the cost of a hit. The cost of a write hit on the T3E is high due to the cost of the ATOMIC_START and ATOMIC_READ calls that are executed in these handlers. For read and write hits, directly invoking the event handler reduces the time spent in handler by about 25% as compared to using the *Ace* runtime sytem to perform the dispatch. We provide the times for a start of read hit in the table; the

Table 6.1: Runtime system overheads for 16-byte regions. All times in μs. The protocol actions (Read and Write) numbers are for the default protocol.

Operation	CM-5	T3E
Mapping miss	33.8μs	5.05μs
Mapping hit (remote)	1.9μs	0.21μs
Mapping hit (local)	0.3μ	0.02μs
Start a read (miss)	64.4μs	21.37μs
Start a read (hit)	2.9μs	0.25μs
End a read	2.7μs	0.09μs
Start a read (direct dispatch)	2.1μs	0.18μs
Start a write (miss, 1 inval)	72.0μs	27.70μs
Start a write (miss, 3 inval)	80.7μs	50.09μs
Start a write (miss, 6 inval)	96.4μs	77.01μs
Start a write (update)	45.0μs	20.83μs
Start a write (hit)	2.9μs	3.43μs
End a write	2.8μs	3.32μs

benefit of direct dispatch for the other events is similar. The cost of a write miss, which also involves data transfer, depends on the number of invalidations that must be performed. On the CM-5, having to perform six invalidations causes the cost of a write miss to rise to 96.4 μs as compared to 72.0 μs when only one invalidation needs to be performed. The default protocol, however, optimizes the performance of write misses when the faulting processor attempts to upgrade the state of a region. In this case, the cost of a write miss is reduced significantly.

The bandwidth attainable by the *Ace* system is determined by measuring the cost of a start-of-write miss on regions of increasing size. We ensure that the miss will require only one invalidation, and therefore, the cost of obtaining write permission for a region is roughly constant for all region sizes. The cost of the entire operation then becomes a function of the size of the region transmitted from the home processor to the requesting processor. Figure 6.1 plots the cost of a start miss with respect to the size of the region being transferred. Applying linear regression to the points in the graph gives a slope of 0.16 μs/byte for the CM-5. In other words, the bandwidth achieved is 6.25 MBytes/s, which is slightly less than the peak bandwidth achievable using the Active Message transfer routine

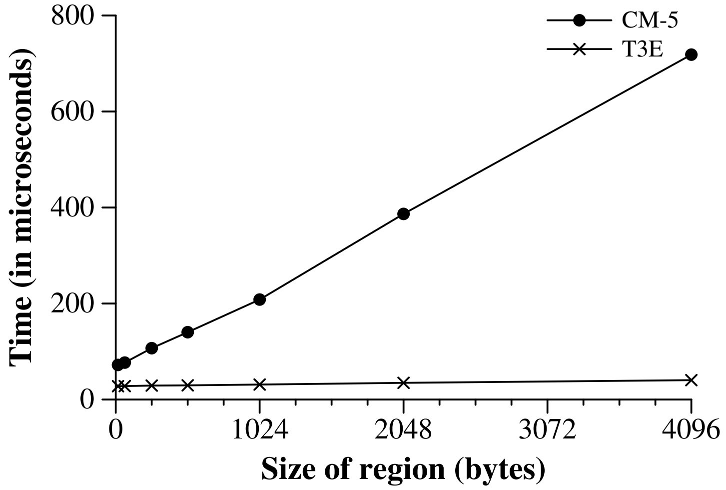


Figure 6.1: Execution time of a start-of-write miss operation in μs with respect to region size.

(8.4 MBytes/s). A similar experiment for the Cray T3E gives a bandwidth of 324 MBytes/s for *Ace* data transfers. Our bandwidth numbers are slightly better than those reported in Scott [86]. Note, however, that the experiments performed by Scott were on an early prototype that had slower processors and parts than the production machine we used.

The *Ace* library operations, maps and accesses, are expensive enough to cause significant overhead in the presence of a large number of such calls. Compiler optimizations, such as eliminating redundant calls, are therefore, essential in reducing this overhead. The advantage of the direct dispatch optimization is also evident from the data in Table 6.1. Furthermore, our experiments reveal that the bandwidth of the *Ace* implementation on the CM-5 is not high, but is constrained largely by the underlying machine's bandwidth limitations. The Cray T3E implementation achieves a bandwidth similar to that reported in by Scott [86].

6.3 Overview of Benchmarks

The performance of real applications is the best measure of a parallel system's efficiency. In our experiments, we use eight benchmarks culled from various sources — Barnes-Hut [11], Blocked Sparse Cholesky (BSC) [81], EM3D [25], Gröbner [18], LU decomposition [46],

Table 6.2: Benchmarks used in experiments.

Name	Inputs
Barnes-Hut [11]	16,384 bodies, 4 time steps, tolerance = 1.0, eps = 0.5
Sparse Cholesky(BSC) [81]	TK15.O
EM3D [25]	32,000 E and 32,000 H vertices, 20% remote edges, degree 10, 100 time steps
Gröbner basis [18]	Lazard
LU decomposition [46]	512×512 matrix, block size = 16×16
Power [63]	10,000 leaves, 1201 internal nodes
Traveling Salesman(TSP) [46]	12 cities
Water [88]	512 Molecules, 3 time steps

Power [63], Traveling Salesman problem (TSP) [46], and Water [88] — to study the performance of our system(Table 6.2). In this section, we provide a brief overview of our benchmarks. Their implementation is examined in detail in Section 6.5.

For each application, we study the use of customizable protocols for the important data structures and compare the performance of implementations with customizable protocols with those that use only the default protocol. Falsafi *et al.* [34] have demonstrated previously that customizable protocols can improve performance substantially in the context of two systems: a system that uses virtual memory protection mechanisms to support fine-grain shared-memory and an all-software system that inserts checks before each access that cannot be determined to be a stack reference. We provide results for application-specific protocols with *Ace* to demonstrate their usefulness for region-based distributed shared-memory systems. In addition, to our knowledge, we are the first to have studied application-specific protocols for BSC, Gröbner, LU, Power, TSP, and Water.

In the following sections, hand-optimized code refers to *Ace* programs in which the compiler/runtime-system interface calls are inserted into the application in an optimal manner (simulating the effect of a smart compiler). The *Ace* versions of Barnes-Hut, and Water were developed from the CRL versions of these programs, which in turn, were derived from the SPLASH code [88]. TSP and LU were developed from CRL versions as well. The *Ace* version of BSC is similar to that in the SPLASH suite. Gröbner was derived from the Multipol [17] benchmark and EM3D from the Split-C [25] version. Finally, Power from the Olden [79] system was the basis of the *Ace* version. Porting the *Ace* versions of these benchmarks from the CM-5 to the Cray T3E required *no* changes to the application code. We found the ability to assume a conventional shared-memory model with an invalidation-based sequentially consistent protocol while developing applications to be of great utility. The *Ace* compiler takes responsibility for generating calls to runtime system routines appropriately removing much of the burden that is placed on programmers of region-based systems such as CRL.

The input data sets were chosen using many considerations. For most of the benchmarks that have corresponding versions in SPLASH, that is, Barnes, LU and Water, the inputs are those recommended in the SPLASH documentation. For EM3D, Gröbner, and Power, we use the input data sets that have been used in experiments previously performed by other researchers with these applications so as to provide a valid base for comparison. For EM3D, we use the data sizes used by Falsafi *et al.* [34]. The input data for Gröbner and Power are the same as those used by Chakrabarti *et al* [18] and Lumetta *et al.* [63], respectively. For TSP, we use an input data set provided with the CRL 1.0 distribution, and finally, for BSC, we use the same data set that we used in an earlier study on irregular parallelism [72].

6.4 *Ace* vs CRL

We validate the performance of our system by comparing it with a recent software distributed shared-memory system, CRL [47]. CRL was originally developed for the CM-5 and Alewife [3]. We compare the execution times of our set of benchmarks on both systems on the CM-5. We do not perform comparisons on the T3E because CRL is unavailable on that platform. Rewriting CRL for the T3E would result in the protocol for CRL taking a form similar to our default protocol. As such, it would be difficult to make a valid comparison on the T3E. The inputs to the benchmarks used in our experiments are listed in Table 6.2.

For a fair comparison of the *Ace* and CRL runtime systems, we use the *same* source

Table 6.3: Times in seconds with CRL and *Ace*(using default protocol) on 32 processors.

Application	CRL	*Ace*
Barnes-Hut	6.9s	4.5s
BSC	4.2s	4.2s
EM3D	0.9s	0.8s
Gröbner	3.2s	4.0s
LU	1.7s	1.5s
Power	3.9s	3.9s
TSP	1.2s	1.5s
Water	1.3s	0.8s

files for both systems, Barnes-Hut, LU, TSP, and Water were taken from the CRL 1.0 distribution and ported to *Ace* by replacing CRL primitives with the corresponding *Ace* calls. For BSC, EM3D, Gröbner, and Power, hand-optimized code written directly for the *Ace* runtime system was converted to CRL using the reverse process. The only case in which an application had to be modified was Gröbner, where a slight change was necessary to work around a fairness bug in CRL. We also *do not* use customized protocols in this experiment, both systems run a sequentially consistent invalidation-based protocol.

The experiments compare the relative performance of the *Ace* and CRL runtime systems. For this purpose, we use hand-optimized code in both cases. The performance of compiled *Ace* code relative to hand-optimized code is examined in Section 6.6. Note that that while the hand-optimized code uses the default protocol, the execution of the programs may not be sequentially consistent. For maximum performance, the hand-optimized code merges and moves runtime-system interface calls using domain-specific information, and therefore, as described in Section 5.3, the execution of the program may not be sequentially consistent. As such it is not clear how to compare the CRL runtime system execution times with the compiler results in Table 6.14 (whether to compare it with the sequentially consistent execution time of the compiler, or the optimized execution time of the compiler).

Figure 6.3 summarizes the results of our experiments. In general, *Ace* programs may have an additional indirection in the dispatch of protocol calls. The *Ace* runtime system, however, uses a streamlined sequentially consistent protocol, and a more efficient mapping

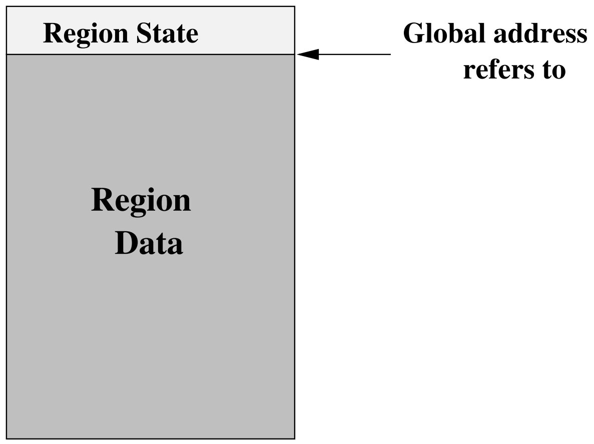


Figure 6.2: Layout of a coherence unit in CRL.

technique. Due to these optimizations, on fine-grained applications such as EM3D and Barnes-Hut, *Ace* performs better than CRL. On the other applications, the performance of *Ace* is comparable to that of CRL. For Gröbner, as we shall see in the next section, minor perturbations in the execution of a program can have a significant effect on its execution time. The difference in times between *Ace* and CRL on Gröbner can be attributed to the fact that the CRL version happens to have a better execution order.

CRL has an advantage in the way in which it performs cache state lookup. In CRL, the state of a region is co-located with the region's data as shown in Figure 6.2. Since CRL does not allow address arithmetic on global addresses, it is guaranteed that all addresses passed to calls to its runtime system will point to the start of a region. The state of this region can be found simply by adding an offset to the address of the region in local memory.

In the presence of pointer arithmetic, the runtime system may be passed an address that points to the middle of a region. The runtime system, in general, cannot determine where in a coherence unit an address may point, and therefore, cannot simply calculate an offset to the state. In *Ace*, since pointer arithmetic is allowed, a more complex scheme for looking up the state of a region (as described in Section 4.4.2) is required. An earlier implementation of *Ace* did not allow pointer arithmetic and used the same cache state lookup scheme as that used in CRL. In an earlier study [74], we compared our previous version of *Ace* to CRL for a subset of our benchmarks. In comparing the two versions of *Ace*, we found that for some of our benchmarks, the execution time is affected somewhat

(5%-20%) by the cache state lookup scheme.

The purpose of this comparison is to demonstrate that the base performance of *Ace* is on par with that of other competing systems. The next section demonstrates how customizable protocols can be used to improve the efficiency of applications.

6.5 Application Performance

In the following sections, for each benchmark, we provide the results of execution under each set of protocols that can be used with the application. If the protocols used are not part of the standard library (Section 4.5), then their semantics are outlined as well. For each benchmark, we also provide speedup curves relative to the performance of the algorithm on one processor. The uniprocessor version was derived from the parallel code by stripping away all synchronization, maps and protocol calls. Finally, where relevant, we provide profiling data gathered during the execution of the applications. In the code used for the following experiments, the transformation of the *Ace* programs into the *Ace* runtime-system code was done by hand (we compare the relative efficiency of compiler-generated code to hand-optimized code in Section 6.6).

For reference, the results of our experiments are summarized in Table 6.4. For each application, the table provides the execution times with the default protocol and the best customizable protocol, as well as the speedup gained by using customizable protocols. For reference, we list the profiling results from the execution of our experiments on the CM-5 in Table 6.5. Since the profiling statistics measure the number of calls to runtime-system routines, they depend mainly on the application and not on the underlying platforms. The profiling numbers on the T3E are therefore similar. The profiling results are for the entire execution of the program.

6.5.1 Barnes-Hut

Barnes-Hut [11] simulates the evolution over time of bodies in a gravitational system. At each time step, the program computes new positions and velocities for all bodies. Each iteration in the algorithm is composed of two phases. The first stage builds an oct-tree

Table 6.4: Comparison of using a single, sequentially consistent protocol and application-specific protocols in *Ace* on 32 processors. "Speedup" refers to the speedup of the customizable protocol versions relative to the single protocol versions.

	CM-5			T3E		
Application	Default	Custom	Speedup	Default	Custom	Speedup
Barnes-Hut	4.5s	3.8s	1.2	0.6s	0.4s	1.5
BSC	4.2s	4.2s	1.0	1.0s	1.0s	1.0
EM3D	0.79s	0.12s	6.6	0.14s	0.017s	8.2
Gröbner	4.0s	0.3s	13.3	2.41s	1.73s	1.4
LU	1.5s	1.3s	1.2	0.30s	0.25s	1.2
Power	3.9s	–	–	0.35s	–	–
TSP	1.5s	0.8s	1.9	0.24s	0.07s	3.4
Water	0.82s	0.60s	1.4	0.055s	0.036s	1.5

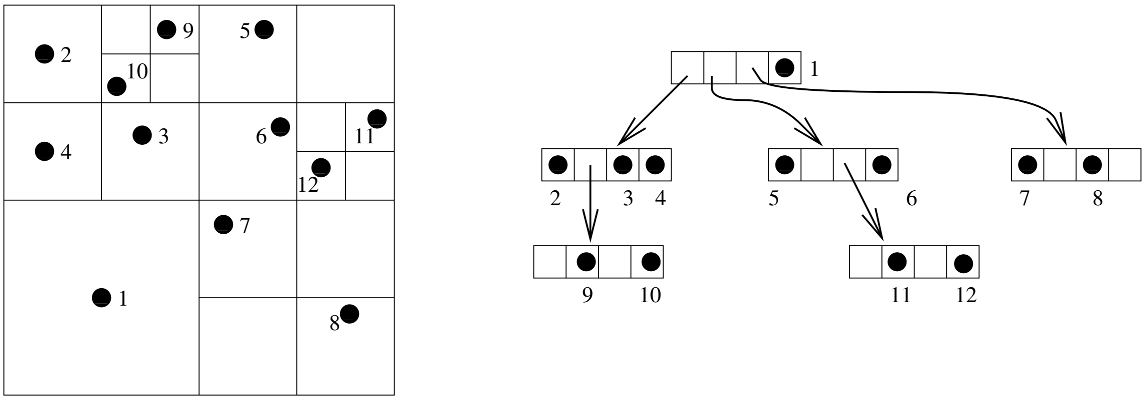


Figure 6.3: Quad-tree for bodies in two dimensions. Filled circles represent bodies.

based on the locations of the bodies in 3-space (Figure 6.3 shows a quad-tree for bodies in 2-space. Oct-trees for 3-space are built similarly). This tree is not balanced; it is deeper in regions that contain many bodies.

The second phase consists of a force calculation, which computes new accelerations and velocities for the bodies, and a position update, which computes bodies' new positions. To calculate the force on a particular body, the algorithm performs a depth-first traversal of the tree from the root. When evaluating an interior tree node, known as a cell, the algorithm first determines if the bodies in the cell are distant enough to be approximated as a point mass. If so, the point mass's contribution is computed and the traversal is terminated. If

Table 6.5: Profiling results on 32 processors for benchmark set. All values represent number of calls executed per processor. “Start of read”, …, “End of Write” represent the number of calls to these events. “Dispatch” is the number of times when the protocol of an access was not known statically and had to be dispatched. “Cache misses” is the number of times a event handler was called with the data in an invalid state. “Maps” represent the number of address translations performed.

App.	Version	Start of Read	End of Read	Start of Write	End of Write	Dispatch	Cache Misses	Maps
Barnes	*Default*	585,441	585,441	22,053	22,053	3,141	16,583	593,154
	Update	585,438	536,906	48,088	42,968	3,442	12,590	593,151
BSC	*Default*	211,198	211,161	4,252	4,252	0	1,591	4,252
	Custom	808	0	0	0	0	740	4,252
EM3D	*Default*	2,620,060	2,620,060	268,004	268,004	0	631,852	2,888,062
	Update	2,620,060	620,060	68,003	66,003	0	23,391	2,888,062
	Static	2,620,060	620,060	66,003	66,003	0	23,391	2,888,062
Gröbner	*Default*	988	988	1,732	1,732	0	1,749	984
	Custom	968	968	223	223	0	176	965
LU	*Default*	2,550	2,550	488	488	0	355	3,060
	Custom	2,152	2,152	131	131	0	358	3,060
Power	*Default*	181	181	370	370	0	273	532
TSP	*Default*	10,063	10,063	250	250	0	33	8
	Custom	10,002	10,002	12	12	0	7	8
Water	*Default*	41,112	41,112	1,696	1,696	0	2,563	515
	Custom	41,112	41,112	610	562	0	1,413	515

not, the algorithm traverses the cell's eight subtrees.

In each step, bodies are assigned to processors so that the amount of work performed by each processor during the step is roughly equal. In addition, the partitioning attempts to keep spatially adjacent bodies on the same processor. A processor is responsible for inserting the bodies it owns into the oct-tree, and for calculating the effects of the body-body interactions on them.

Protocols

The accesses to bodies in this algorithm follow a certain pattern. During a time-step, there is exactly one processor that writes to a body and several processors that may read it. Furthermore, the set of processors that read a body changes slowly. Whether a processor accesses a body depends on the body's position. Since the movement of bodies is gradual across time-steps, the set of processors accessing any given body is quite stable. Finally, it is not imperative that body updates be propagated to processors instantaneously. As such, the update protocol discussed in Section 4.5.2 is ideal for this application. The protocol keeps track of the current list of sharers of a given body, and at the end of each time-step propagates updates to all sharers. The looser semantics of the update protocol (release consistency) allows for more efficient communication.

Results

Table 6.6 summarizes the results of executing the $\mathcal{A}ce$ implementation of Barnes-Hut with 16,384 bodies (speedup curves are shown in Figure 6.4). The times reported are the average execution time per iteration. The first iteration is used to "warm" the system, and is not considered in the average. On the CM-5 and T3E, the custom protocol version executes approximately 16% faster than the one that uses the default protocol.[2] Profiling statistics reveal that on shared-memory accesses, the custom protocol version incurs 25% fewer cache misses than the base version. In addition, since the update protocol specifies null handlers for the end-of-access events, there are fewer invocations of protocol routines as

[2]Here, and in the rest of the document, "a executes $x\%$ faster than b" implies that $x = 100 \times \frac{time(b) - time(a)}{time(b)}$.

Table 6.6: Execution times (in seconds/iteration) for Barnes-Hut.

	CM-5		T3E	
Procs	Default	Update	Default	Update
1	56.9s	56.9s	4.3s	4.3s
2	44.5s	43.1s	5.4s	4.9s
4	24.8s	22.5s	3.1s	2.7s
8	13.7s	11.8s	1.7s	1.4s
16	7.5s	6.3s	0.9s	0.7s
32	4.5s	3.8s	0.6s	0.4s

well. Note that the effects of the custom protocol are more pronounced as the number of processors increase when there is a larger amount of communication.

A fine-grained application such as Barnes-Hut presents a challenge to a software distributed shared-memory system. The amount of work performed between accesses to remote data is not high, and therefore, the overhead of supporting shared memory plays a significant role in the overall execution time. Yet, *Ace* achieves reasonable speedup and compares favorably with other software distributed shared-memory systems in its performance on Barnes-Hut. For comparison, on Blizzard-E [85], a shared-memory system implemented on a CM-5, a version of Barnes-Hut, similar in implementation to ours, and using application-specific protocols described in Falsafi *et al.* [34] spends an average of 7.9s per iteration on 32 processors.

6.5.2 Blocked Sparse Cholesky

The Cholesky factorization of a sparse, symmetric, positive-definite matrix results in a matrix that facilitates the solution of the linear system of equations associated with the input matrix. The Blocked Sparse Cholesky (BSC) [81] algorithm first partitions the input matrix, based on its sparse structure, into *blocks* (see Figure 6.5). There are two operations defined on blocks: *block update* and *block factorization*. A block is factored once all updates to it have completed, the number of updates being determined by the sparse structure of the matrix. At the beginning, there are several blocks to which no updates will be made; these blocks are factored immediately. A factored block then interacts with other factored blocks

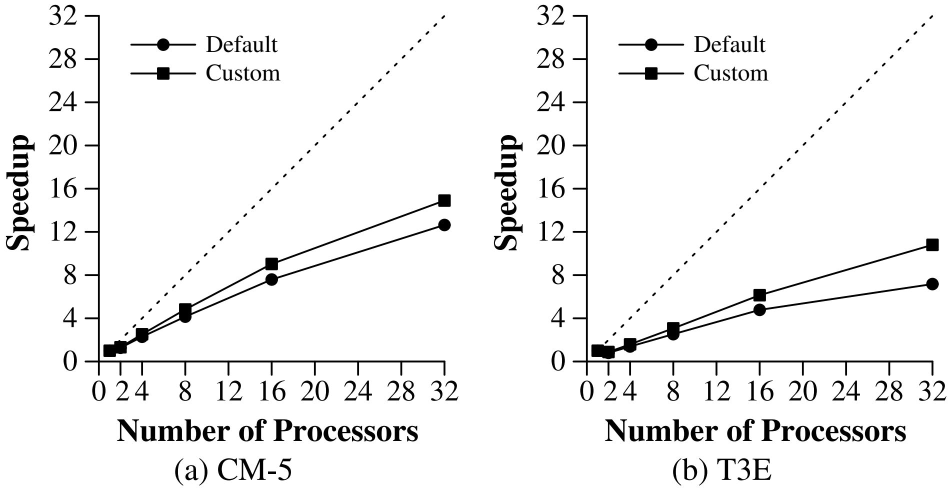


Figure 6.4: Speedup curves for Barnes-Hut.

or with itself to update other blocks. No block is completed until its ancestors are factored, and once a block is factored, no further updates to it will be made. The algorithm completes once all blocks have been factored.

The efficiency of BSC can be increased by dividing a subset of the blocks into disjoint sets called *domains*. The domains are chosen so that they are independent, that is, they receive no updates from any other block or domain. At the onset of computation, these domains are factored concurrently. After a domain is factored, the series of block updates defined by the domain is performed. At the end of the domain-block update phase, the updates and factorizations of the remaining blocks proceed as described in the previous paragraph. Of the stages in this algorithm, the block factorization and update phase is the most computationally intensive.

The implementation of BSC allocates domains to processors so that nodes have roughly the same amount of work to perform during domain factorization. The blocks are distributed in a 2-D cyclic manner across nodes to ensure that each processor receives approximately the same number of blocks. Updates are handled with a task queue at each processor. Once a block is factored, its owner processor inserts this information into the task queues of processors that may require the factored block. When a processor processes a notification that a block is complete, it determines whether it can use this block to perform an update to a block it owns. The processor checks whether both source blocks required

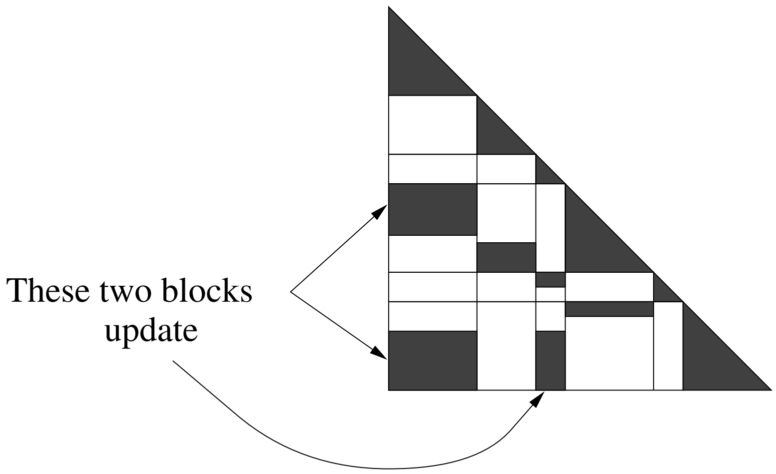


Figure 6.5: Breaking a sparse, symmetric, matrix into blocks. Shaded portions represent blocks, unshaded portions are zeroes in the matrix. As shown, an update operation takes two source blocks and modifies a third.

for an update are complete, and if they are, updates the appropriate destination block. Each processor is responsible for updating the blocks it owns.

Protocols

A characteristic of the BSC algorithm is that block and domain data are written only by their home processor, and no processor reads remote data until all writes to the data have been performed. Using this information, one can design a protocol that simplifies the actions performed on read and writes. For writes, no processing is necessary — the data is guaranteed to be resident and in an exclusive state. For reads, a routine that requests data from the home processor when a cached copy does not exist is sufficient.

Results

In our experiments with BSC, we use as input, TK15.O, a benchmark sparse matrix distributed with SPLASH. Table 6.7 and Figure 6.6 provide the execution times and speedup curves for this application. The times reported are those spent in the factoring stage of the algorithm.

The speedup obtained by the application on 32 processors is low, 7.3 on the CM-5 and

3.0 on the T3E. This, however, is due to the inherent irregularity of the application that makes load balancing extremely difficult. This problem is especially severe on the Cray T3E where the fast processors reduce the amount of time a processor is busy performing a task. On the T3E, the lack of work results in the algorithm running slower on 32 processors (for our input) than on eight processors.

Due to the inherent load imbalance in the application, on the CM-5, even an efficient message-passing implementation [73] runs only 20% faster than the $\mathcal{A}ce$ version. The discrepancy in the two implementations can be attributed more to a limitation of shared memory than to any $\mathcal{A}ce$ bottleneck. In the message-passing implementation of BSC, a processor appends information to a remote processor's queue by sending it an active message with the relevant information. When the message reaches the remote processor, an atomic handler is launched that adds the task to the queue. This process is entirely asynchronous and is correct because the relative order of additions to a task queue is unimportant. In shared-memory systems, however, the only medium of communication is memory. Before appending a task, a processor must obtain a lock on the remote task queue. When the remote processor processes the task, the information is not available locally; the protocol must request it from the processor that created the task. This process requires much more communication than the message-passing version. If, in the shared-memory implementation, we replace the task queue routines with the message-passing version, the difference in execution time between the two drops to 5%.

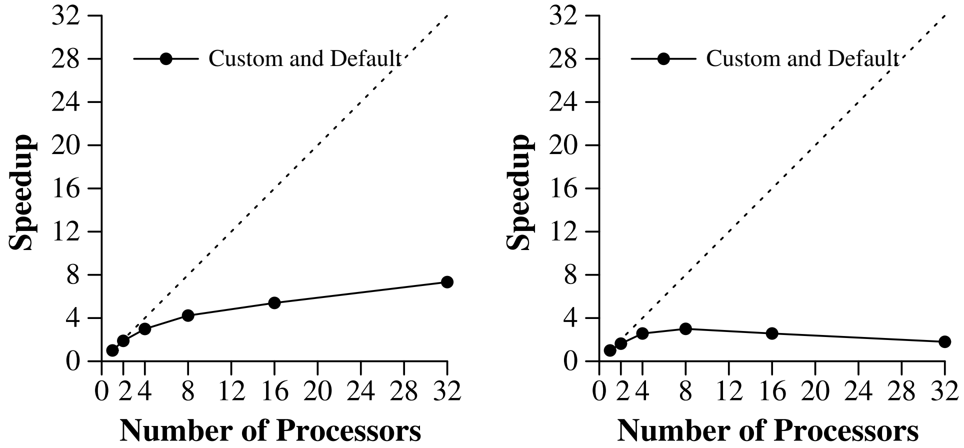
With regard to custom protocols, in BSC, the protocol overhead from accesses to blocks or domains is minimal. The communication mechanism necessary for efficiency is a bulk transfer mechanism to transport blocks between processors. Since the implementation allocates blocks as atomic units, the default protocol uses the bulk transfer mechanism for blocks automatically. Any further gains from customizable protocols is insignificant, as can be seen from the results.

6.5.3 EM3D

EM3D models the propagation of electromagnetic waves through objects in three dimensions [25]. The problem is formulated as a computation on a bipartite graph with directed

Table 6.7: Factorization times (in seconds) for BSC.

	CM-5		T3E	
Procs	Default	Custom	Default	Custom
1	30.8s	30.8s	1.8s	1.8s
2	16.3s	16.3s	1.1s	1.1s
4	10.3s	10.3s	0.7s	0.7s
8	7.2s	7.2s	0.6s	0.6s
16	5.7s	5.7s	0.8s	0.7s
32	4.2s	4.2s	1.0s	1.0s



(a) CM-5 (b) T3E

Figure 6.6: Speedup curves for BSC.

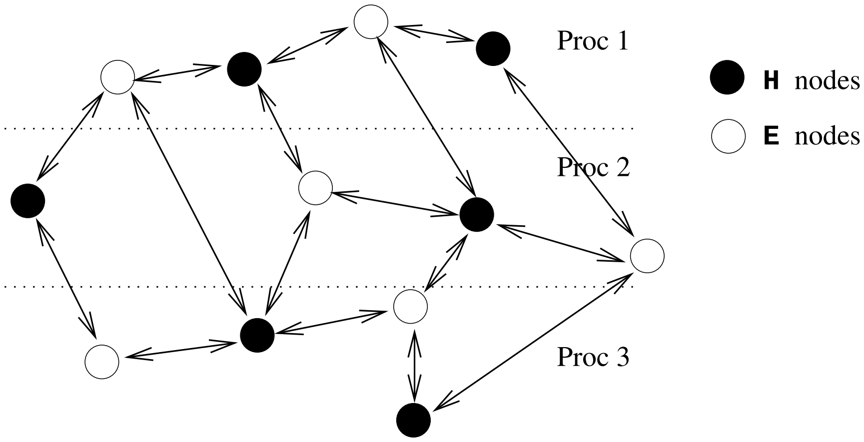


Figure 6.7: Sample bipartite graph for EM3D.

edges from E nodes, which represent electric fields, to H nodes, which represent magnetic fields, and vice versa (see Figure 6.7). The main loop of EM3D computes the change in E and H values over time. In each iteration, new E values are computed from the weighted sum of neighboring H nodes, and then, new H values are computed from the weighted sum of neighboring E nodes. The graph structure is invariant during this process. In the *Ace* implementation, each processor is allocated an equal number of E and H nodes, and is responsible for updating these nodes every iteration.

Protocols

The data access patterns of EM3D are similar to those of Barnes-Hut. During a time-step, only one processor writes to any given node. The UPDATE protocol, which is used in Barnes-Hut, is applicable for this application as well. There are, however, certain characteristics of EM3D that allow further optimization. While in Barnes-Hut, the set of processors that read an object during a time-step changes gradually, in EM3D, the set remains constant. The underlying graph data structure is invariant across time-steps which implies that the communication pattern does not change either. In addition, each node is written only by the processor that owns it. The STATIC update protocol (see Section 4.5.3), which stores a list of the data communicated during the first iteration, and performs updates of these data at subsequent barriers using bulk transfer, is ideal for this application. This protocol is

```
e_space = Ace_NewSpace(SC)
h_space = Ace_NewSpace(SC)
Build the graph.
Ace_ChangeProtocol(e_space, STATIC)
Ace_ChangeProtocol(h_space, STATIC)
for i = 0 ... # of time steps
    Compute new E values.
    Ace_Barrier(e_space)
    Compute new H values.
    Ace_Barrier(h_space)
endfor
```

Figure 6.8: Pseudocode for EM3D main loop.

similar to that described by Falsafi *et al.* [34] for EM3D.

In the *Ace* implementation, two spaces are allocated, one for E nodes and one for H. Though the communication pattern of these data structures is the same, the actual data transferred, and hence, the update lists, differ. The barrier after the calculation of E (H) nodes is associated with the E (H) node space and forwards these values, which are used in the subsequent H (E) calculation. Figure 6.8 provides the pseudocode for the main loop of this application.

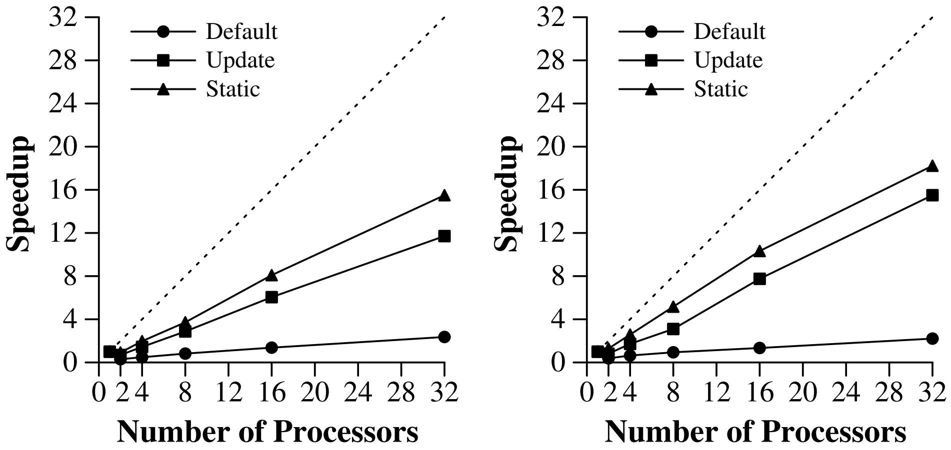
Results

Experiments with EM3D use graphs with 32,000 E nodes and 32,000 H nodes. Each node has degree 10, and on 32 processors, 20% of the edges cross inter-processor boundaries. The algorithm was run for 100 time-steps. We ran experiments with the default, UPDATE, and STATIC protocols, the results of which are presented in Table 6.8 and Figure 6.9. The time reported is the average time per iteration, not including the first iteration, which is used to "warm" the system.

As in Barnes-Hut, the amount of work performed between accesses in this application is extremely low. This factor, compounded with the producer-consumer access pattern, which is not well suited to an invalidation-based protocol (see Section 1.2), results in the base version of EM3D running slower than the sequential version on 8 processors! The

Table 6.8: Execution times (in seconds/iteration) for EM3D.

	CM-5			T3E		
Procs	Default	Update	Static	Default	Update	Static
1	1.86s	1.86s	1.86s	0.31s	0.31s	0.31s
2	6.27s	2.70s	2.02s	0.81s	0.38s	0.24s
4	3.91s	1.27s	0.94s	0.48s	0.18s	0.12s
8	2.27s	0.64s	0.50s	0.33s	0.10s	0.06s
16	1.36s	0.30s	0.23s	0.23s	0.04s	0.03s
32	0.79s	0.16s	0.12s	0.14s	0.020s	0.017s



(a) CM-5 (b) T3E

Figure 6.9: Speedup curves for EM3D.

benefits of the custom protocols are two-fold — they reduce the number of protocol action invocations (several handlers are null handlers), and use a much more efficient communication mechanism. The advantage of the static update protocol is that it builds the sharer list only in the first iteration, unlike the update protocol which builds these lists each iteration. With customizable protocols, a respectable speedup of 16 on the CM-5 and 18 on the T3E is obtained on EM3D.

On Blizzard-E [85], running a comparable version of EM3D with a similar STATIC protocol [34], has an execution time of 0.1 seconds/iteration on 32 processors. Blizzard-E implements access checks using a hardware technique. Once the access pattern is determined in the first iteration and updates are forwarded appropriately, subsequent iterations do not incur any access faults, and therefore, will not have any overhead. In *Ace*, since checks are done in software, every iteration incurs some overhead in executing these checks.

Additional insight into the performance of this application is obtained by considering profiling statistics gathered during its execution (see Table 6.5). The table shows that EM3D performs approximately 2.9 million address translations per processor. Additional experiments reveal that 2.5 million of these calls were to addresses where the home processor of the address is the same as the accessing processor. Recall that the *Ace* runtime system optimizes address translations for this special case (Section 4.4.2). Using the costs in Table 6.1 for the CM-5, we can calculate that without the address translation optimization, the mapping time for these accesses would be 4.75 seconds ($2.5 \times 10^6 \times 1.9\mu s$) as opposed to 0.75 seconds with the optimization. Further calculations show that, with the STATIC protocol, the time spent in the start-of-read routine constitutes about 20-30% of the execution time of the main loop (there are 2,000,000 start read calls in the main loop). Approximately, an equal amount of time is saved by removing invocations to the other protocol event handlers. Finally, notice the large disparity in the number of cache misses in the two cases. With the custom protocols, there are $\frac{1}{27}$ as many cache misses as with the default protocol!

6.5.4 Gröbner

The Gröbner basis of a set of polynomials is another set of polynomials with the same roots as the initial set. The basis possesses certain additional properties that facilitate the calculation of these roots. The process of deriving a Gröbner basis of a set is analogous to Gaussian elimination. The basis set grows monotonically through the algorithm. Each task consists of processing a pair of polynomials to generate a new polynomial. If the new polynomial is not reducible by any of the polynomials currently in the basis set then it is added to the basis. When a polynomial becomes a member of the basis, it may generate a set of new pairs that must be processed. The algorithm ends when all pairs of polynomials have been processed.

The *Ace* version of the Gröbner basis benchmark is based on the parallel implementation developed by Chakrabarti *et al.* [18]. In the algorithm, the list of pairs of polynomials

```
1  while not done
2      Get a polynomial pair, (p,q), from queue.
3      Generate new polynomial s, from p and q.
4      Try to reduce s using the current basis set.
5      if (s is not reduced)
6          ACE_LOCK(tail of basis set)
7          Try reducing s again.
8          if (s is not reduced)
9              /* Add s to basis set. */
10             basis_tail→next = s
11             basis_tail = s
12             Add new tasks to global queue.
13         endif
14         ACE_UNLOCK(tail of basis set)
15     endif
16 endwhile
```

Figure 6.10: Pseudocode for main driver in Gröbner.

that are to be evaluated is stored as a global task queue. The basis set is stored as a linked list in shared memory. A processor, when idle, removes a task, which corresponds to a pair of polynomials that is to be evaluated, from the global queue. When appropriate, processors append polynomials to the basis set or new tasks to the global queue, using locks for synchronization.

Figure 6.10 gives the pseudocode for the main driver routine for this application. Notice that the algorithm attempts to reduce the new polynomial s twice on Line 4 and Line 7. During the evaluation of polynomials, most new polynomials generated by pairs are reduced by the basis set. This algorithm, therefore, seeks to reduce the contention on the lock governing the basis set by first attempting to reduce s on a potentially stale version of the basis set. If it passes through the first test, than the irreducibility of the new polynomial is tested more thoroughly by obtaining a lock on the tail pointer of basis set linked list, which ensures that the basis set remains constant during the reduction process. This optimization is described in greater detail in Chakrabarti *et al.* [18].

Protocols

The major sources of contention in this algorithm are the locks on the basis set and task queue data structures. In both cases, all accesses to the relevant data (the tail of the basis set and the head of the task queue) are surrounded by locks. We can, therefore, use the SYNCH protocol described in Section 4.5.4 to optimize the handling of these accesses.

Results

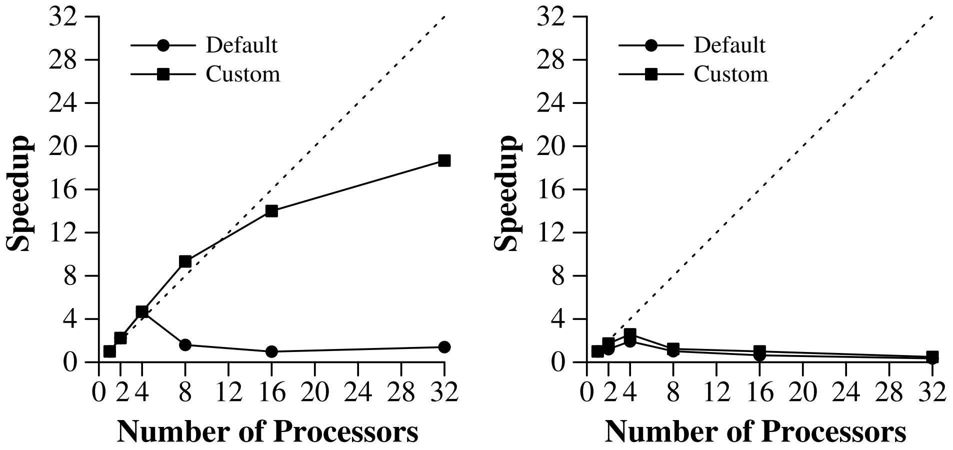
We use Lazard, which is distributed with the Multipol suite [17], as the input to our implementations. The execution times and speedup curves for Gröbner are presented in Table 6.9 and Figure 6.11. The performance of Gröbner basis is highly sensitive to the order in which tasks are performed [18, 84]. Superlinear speedup may occur when a "good" polynomial is added to the basis that allows other polynomials to be reduced quickly. On the CM-5, this pattern is revealed in our experiments as well where we obtain superlinear speedup on up to eight processors with the SYNCH protocol. With the default protocol, the central queue and basis list locks become bottlenecks, and performance drops severely with more than four processors. The more efficient SYNCH protocol results in reasonable speedups on up to 32 processors.

On the T3E, there is not enough parallelism to justify the use of more than four processors. The more efficient SYNCH protocol does reduce the communication overhead and the overall execution time as compared to the default protocol. An interesting point in this application is that the CM-5's slower processor sometimes actually helps the algorithm. It appears that "better" polynomials may be added to basis when a processor does not evaluate the polynomials quickly. We shall see an additional example of this peculiar characteristic when we consider the compiler results in Section 6.6. With respect to the benefits of customizable protocols, it must be noted that since the algorithm's performance is sensitive to subtle variations, it is not clear how much of the performance improvement can be attributed to better communication performance. It is possible that the customizable protocol version generates a better task ordering serendipitously.

Comparing $\mathcal{A}ce$'s performance with the results reported by Chakrabarti *et al.* [18], we find that our implementation performs significantly better. They report a time of 55.5s on

Table 6.9: Execution times for Gröbner.

	CM-5		T3E	
Procs	Default	Custom	Default	Custom
1	5.6s	5.6s	0.85s	0.85s
2	2.5s	2.5s	0.70s	0.49s
4	1.2s	1.2s	0.44s	0.33s
8	3.5s	0.6s	0.83s	0.69s
16	5.7s	0.4s	1.32s	0.85s
32	4.0s	0.3s	2.41s	1.73s



(a) CM-5 (b) T3E

Figure 6.11: Speedup curves for Gröbner.

one processor, and 2.6s on ten processors for Lazard on a CM-5. Scales and Lam [84] have remarked that their version of Gröbner on the CM-5 appears to have better heuristics than the Chakrabarti version and obtains much better serial and parallel times as well. We believe that the same reason accounts for *Ace*'s better performance. The speedup reported by Scales and Lam is comparable to that attained by *Ace*.

6.5.5 LU Factorization

Like Blocked Sparse Cholesky factorization, LU factorization is an integral component of systems that solve sets of linear equations. LU factorization, however, is used when the

0	1	0	1	0	1
2	3	$\boxed{2}$	3	2	3
0	1	0	1	0	1
2	3	2	3	2	3
0	1	0	1	0	1
2	3	2	3	2	3

Block size = 4x4

(a)

```
1   for each block i = 1...N
2       Factor block B_ii
3       Ace_Barrier()
4       Update blocks B_*i and B_i*
5       Ace_Barrier()
6       for j = (i+1)...N
7           for k = (i+1)...N
8               B_jk = B_jk - B_ji × B_ik
9           endfor
10      endfor
11      Ace_Barrier()
12  endfor
```

(b)

Figure 6.12: (a) Partition of a 24×24 matrix on 4 processors (b) Pseudocode for algorithm.

input matrix is dense. The blocked LU algorithm, based on that described by Rothberg *et al.* [82], divides a $n \times n$ matrix, A, into a $N \times N$ array of blocks of size $b \times b$. The processors are arranged in a rectangular grid, and the blocks are partitioned among the processors in a wrapped fashion (see Figure 6.12a). In the following B_{ij} refers to the block in the ith row and jth column of the block array. For example, in Figure 6.12a, B_{23} refers to the highlighted block.

In the LU factorization algorithm, all processors collaborate in performing the calculations associated with each block column (please refer to Figure 6.12b for pseudocode). In each iteration, the processor that owns the diagonal block B_{ii}, $0 \leq i < N$, factors it. The updated B_{ii} is then used by the processors that own blocks B_{*i} or B_{i*} to factor these blocks. Finally, all other B_{jk}, $j,k > i$ are updated by their owners according to the formula on Line 8 in Figure 6.12b.

Protocols

The access patterns of blocks are similar to those of BSC, blocks are updated only by their owners, and once updated, blocks are read only. We can, therefore, use the same protocol as that used for BSC, but the results are similar — not much benefit. Unlike BSC, however, one can predict the access patterns in LU factorization statically. Recall that the set of processors is initially arranged as a rectangular grid. An invariant during the execution of the application is that data is transmitted from one processor only to another in its row or its column in this rectangular grid. For example, in Figure 6.12a, Processor 0 may communicate with Processors 1 and 2, but never with 3.

A customized protocol can take advantage of this regularity in communication. A counter on each processor is used to track the current *phase* of the program, that is, how many barrier operations have been performed. When a processor executes a start-of-read operation and the data is not resident locally, it sends a message to either its north or left neighbor in the processor grid, depending on the direction in which the home of the required data lies. If the neighbor has the data, it services the request, otherwise, it defers processing the message until the data arrive at the processor. When the home processor receives a request, if its phase is the same as that of the requesting processor, it can satisfy the request. Otherwise, it queues the request and satisfies it at the next barrier call.

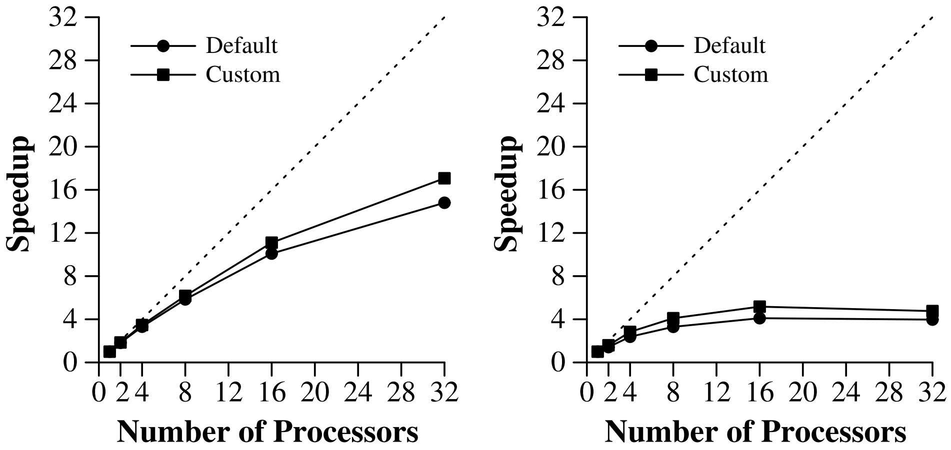
The phase information allows the protocol to achieve the same effect as global synchronization on Ace_Barrier calls by using data requests as lazy synchronization mechanisms. The correctness of the protocol is ensured by the fact that each object has a single writer and all reads and writes are separated by barriers. A benefit of this protocol is that the handling of requests is distributed among all processors. When a set of processors requires a data object, the home processor does not handle all requests. A request for data is handled by the processor neighboring the requesting processor.

Results

In Table 6.10 and Figure 6.13, we provide the results of executing LU factorization on a 512×512 matrix with 16×16 blocks. As in BSC, the major communication optimization in LU factorization is to use bulk transport to transfer blocks between processors. Since both the default and custom protocols use user-specified granularity, both receive the benefits of this optimizations. The additional benefit of using the custom protocol is approximately 1.2 on 32 processors on both platforms over using the default protocol. The speedup attained on 32 processors is approximately 17. On the Cray T3E, due to the fast processors at the nodes, the problem size is too small to achieve significant speedups on 32 processors. Further experiments on a larger data set (1024×1024 matrix, 32×32 blocks)

Table 6.10: Execution times (in seconds) for LU factorization.

	CM-5		T3E	
Procs	Default	Custom	Default	Custom
1	22.2s	22.2s	1.19s	1.19s
2	12.2s	12.0s	0.84s	0.75s
4	6.7s	6.4s	0.50s	0.42s
8	3.8s	3.6s	0.36s	0.29s
16	2.2s	2.0s	0.29s	0.23s
32	1.5s	1.3s	0.30s	0.25s



(a) CM-5 (b) T3E

Figure 6.13: Speedup curves for LU factorization.

obtained speedups of approximately 13.4 on 32 processors (as compared to 4.8 with the current data set).

6.5.6 Power

The Power benchmark [63] solves an optimization problem over a power network to determine the power prices that will maximize the benefit to the community. The power network is represented by a tree, with customers at leaves and the power plant at the root, and is typically organized as shown in Figure 6.14. In the example, totally, there are 10,000 leaf nodes, and 1201 internal nodes.

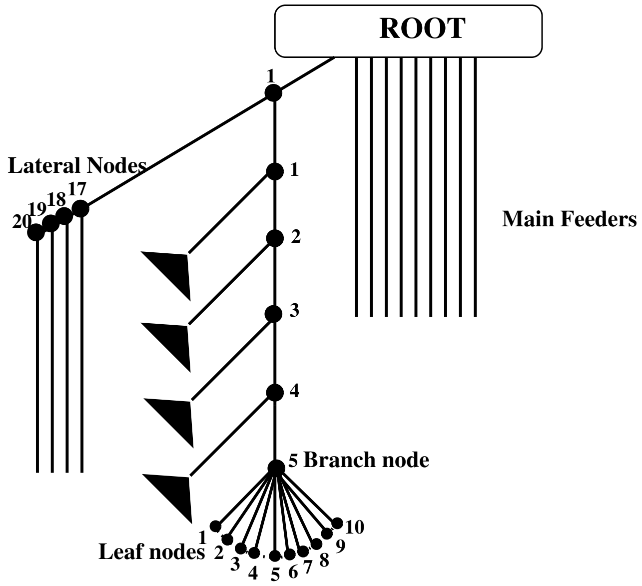


Figure 6.14: Power distribution tree. The root has 10 main feeders. Each feeder has 20 lateral nodes, each of which have 5 branch nodes. Each branch node has 10 leaves.

In our implementation, an equal number of lateral nodes (in Figure 6.14, there are 200 lateral nodes, one for each of the 10 main feeders) is assigned to each processor. All children of a lateral node are allocated from the private memory of the processor that owns the parent lateral node. At each step, the power prices are first propagated from the root down to each lateral node, and then, down the tree to branch and leaf nodes. The leaves use these values to calculate new customer demand. Finally, new power flows are propagated up the tree through the lateral nodes to the root. This process ends when a convergence criteria is met.

Protocols

The Power benchmark possesses a high degree of parallelism. The majority of the computation is spent in evaluating the leaf and branch nodes, which are located in a processor's private memory. As such, communication costs do not play a significant role in determining execution time. A custom protocol for this application is therefore unnecessary. As we shall see from the results below, the default protocol is sufficient for the limited communication in this benchmark.

Recall that in our implementation, the branch and leaf nodes are stored in private memory. If it is necessary to store them in shared memory so that their values may be accessed in a post-processing step, handling these accesses with the default protocol may cause high overhead. An efficient strategy might be to use a protocol with null event handlers during the execution of the algorithm, and then, to switch to the default protocol before the post-processing step.

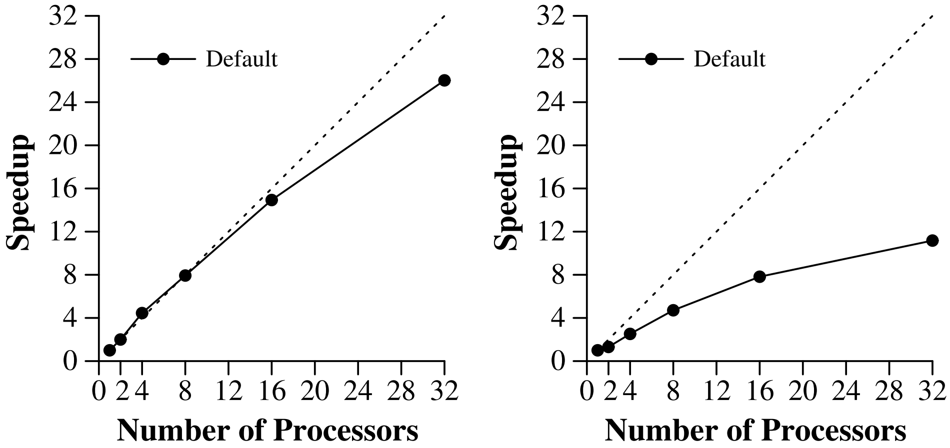
Results

The performance results and speedup curves for Power are provided in Table 6.11 and Figure 6.15. On the CM-5, the application achieves good speedup on up to 32 processors. We have determined from further experiments that on 32 processors, *Ace* protocol and communication overhead amount to less than 5% of the execution time. Any custom protocol, therefore, will not have much influence over the performance of the algorithm. Notice, however, that the speedup curves drops slightly between 16 and 32 processors. On larger numbers of processors, the propagation of values from the root through lateral nodes and vice-versa, may become a bottleneck. In this case, a protocol to optimize these accesses may be necessary. Lumetta *et al* [63] do not provide timings for executions on 32 processors. They report an efficiency ($\frac{speedup}{processors}$) of 75% on 64 processors on the CM-5. *Ace* obtains an efficiency of 81% on 32 processors, which is comparable to the results reported by Lumetta.

The performance of the Cray-T3E version is not nearly as efficient as that of the CM-5 — the maximum speedup attained is approximately 11.1. Even on two processors, the speedup is only 1.31. Further tests reveal that communication costs still constitute a small portion (7%) of the execution time of the algorithm. The Level 1 cache on a T3E is small (8 KBytes) and direct-mapped. With 10,000 leaves, it is unlikely that much data is reused between iterations on the power tree on one processor. The T3E, however, has a 96 KBytes 3-way associative Level 2 cache which could possibly hold all data in our input tree. In an uniprocessor execution, most of the data accesses are likely to hit in the second level cache. In the multiprocessor versions, however, communication between processors prob-

Table 6.11: Execution times for Power.

Procs	CM-5	T3E
1	101.1s	3.91s
2	50.8s	2.97s
4	22.5s	1.55s
8	12.8s	0.83s
16	6.8s	0.50s
32	3.9s	0.35s



(a) CM-5 (b) T3E

Figure 6.15: Speedup curves for Power.

ably results in data being knocked out of the second level caches; an external backmap on the T3E ensures local memory consistency by invalidating cache lines appropriately on memory-to-memory communication between processors. We believe that this second level cache effect may be the cause of lower speedup on the T3E.

6.5.7 Traveling Salesman Problem

Given a weighted graph, the Traveling Salesman problem is to find a minimum cost path through the graph that visits all vertices exactly once. Our algorithm for this problem, derived from that in the CRL benchmarks [46], solves the problem by brute-force evaluation. A processor designated as the master processor generates a set of path prefixes and stores

```
while all prefixes have not been processed
    ACE_LOCK(prefix_counter)
    prefix_counter++
    Get path prefix, p, associated with prefix_counter.
    ACE_UNLOCK(prefix_counter)
    Evaluate all paths starting with prefix p.
    Change minimum solution, if necessary.
endwhile
```

Figure 6.16: Pseudocode for Traveling Salesman main loop.

them in a globally shared array. Each processor attempts to determine the minimum path in the manner described by the pseudocode in Figure 6.16. Free processors use a global counter variable to identify the next prefix to be evaluated. A processor that takes responsibility for a prefix determines if any solution beginning with the prefix has length less than the current global minimum. This process continues until all prefixes are evaluated.

Protocols

The accesses to the global counter that parcels work to processors is the major bottleneck in this algorithm. The semantics of the global counter are that its value monotonically increases, and that each access to it returns a unique value. As shown in Figure 6.16, this effect is achieved by surrounding accesses to the counter with locks to ensure mutual exclusion, and incrementing the counter each time.

A custom protocol was designed for this application that optimizes accesses to this data object. In the custom protocol, when a processor requests a lock on the global counter, the home processor, instead of waiting for a start-of-write operation, returns the current value of the counter. It then increments a private copy of the counter to reflect the operation that is to occur at the remote processor. When, subsequently, another processor requests a lock, the home processor does not wait for the completion of outstanding locks, but provides the requesting processor with the new value (which is unique), and increments it again. As such, the start-of-access, end-of-access, and unlock operations can be discarded. Abstractly, this protocol implements an atomic fetch-and-increment operation using active message handlers. Since the T3E provides atomic swap and increment memory operations, this protocol can be implemented extremely efficiently on that machine. The semantics of the counter variable are met because of three invariants:

1. All accesses to the counter are preceded by a request for a lock that is sent to the home processor.

2. Once a processor receives a lock, it has the current value of the variable, and performs operations that will update the variable locally.

3. The home processor mirrors the effects of these updates before granting the lock, so that the next access from any processor can be satisfied instantaneously.

While, as usual, the only change in the application to use the custom protocol is the addition of an Ace_ChangeProtocol call, the location of this call is important. The protocol must be changed after the counter is initialized to 0, which occurs in a preprocessing routine. Otherwise, the task associated with the counter value 0 will not be executed.

Results

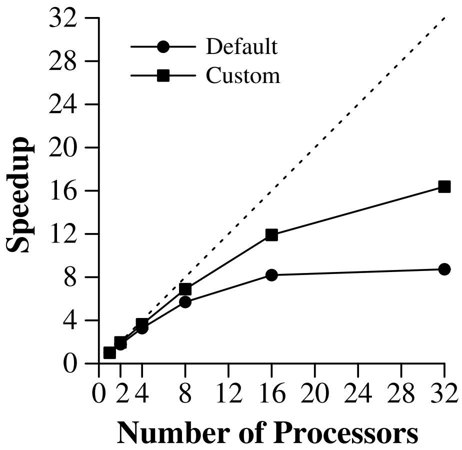
Table 6.12 and Figure 6.17 provides the results of executing the Traveling Salesman problem with a graph of 12 cities. It can be seen from the results that the global counter quickly becomes a bottleneck in the base case; the time for 16 and 32 processors are about equal. The default protocol is not well suited for variables that are updated frequently. Inefficiencies in the default protocol not only increase the communication overhead, but cause load imbalance by increasing the contention for locks. Switching to the custom protocol cuts the execution time almost in half. As expected, the benefits of the custom protocol are more evident at higher numbers of processors, when the counter variable is more of a bottleneck. This application is a good example of an instance where a programmer's insight can lead to a highly specialized and efficient protocol.

6.5.8 Water

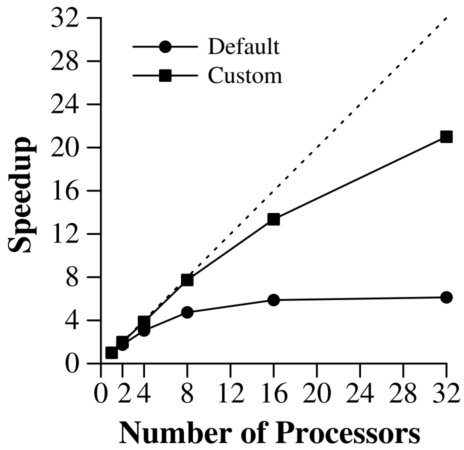
Water is an N-body algorithm that models the dynamics of a system of water molecules. The algorithm progresses in a series of time-steps, where each step consists of calculat-

Table 6.12: Execution times (in seconds) for Traveling Salesman.

	CM-5		T3E	
Procs	Default	Custom	Default	Custom
1	13.1s	13.1s	1.47s	1.47s
2	7.3s	7.0s	0.84s	0.74s
4	4.0s	3.6s	0.48s	0.38s
8	2.3s	1.9s	0.31s	0.19s
16	1.6s	1.1s	0.25s	0.11s
32	1.5s	0.8s	0.24s	0.07s



(a)CM-5



(b) T3E

Figure 6.17: Speedup curves for Traveling Salesman.

ing intra-molecular and inter-molecular potentials. In the *Ace* implementation, a globally shared array holds the state of each molecule. At the onset of computation, each processor assumes responsibility for an equal number of adjacent molecules in the global array.

Most of the computation time in Water is spent in performing inter-molecular updates. During this phase, the interactions of each molecule with the $\frac{n}{2}$ molecules following it in the array is computed, where n is the number of molecules in the system (abstractly, the array wraps around). An interaction involves reading two molecules, computing forces, and updating both molecules. To avoid accessing remote data every time a remote molecule is to be updated, when the molecule-molecule interactions are evaluated, the update values are collected in private temporary arrays. Once all interactions are evaluated, the molecules are updated using the values from the temporary arrays. Locks ensure that updates are mutually exclusive. Barriers delimit the other phases of the application.

Protocols

In the inter-molecular update phase, each processor updates approximately half of the molecules in the array. Alternatively, a molecule is updated by about half the processors in the system. In general, a molecule is first updated by its home processor, then by the adjoining processor that is the owner of the molecules preceding the home processor's molecules, etc. An efficient protocol for this sort of access pattern is to pipeline the transfer of data between adjacent processors. When a processor has finished updating a molecule, it forwards the molecule data to the adjacent processor. When that processor is ready to perform its update, the data is available locally.

Specifically, in the *Ace* implementation, the molecule array is partitioned among p processors so that Processor 0 has the first $\frac{n}{p}$ elements, processor 1 the next $\frac{n}{p}$, etc. This partitioning implies that a molecule updated at Processor i will generally be updated next at $i-1, i \neq 0$ or $p-1, i=0$. We call this processor the *adjoining* processor. The custom protocol for Water, called SHIFT, pipelines transfers to adjoining processors. Since updates to molecules are delimited by locks, SHIFT is based on specialized versions of the lock and unlock routines. All other events are discarded. The protocol has the following semantics:

1. Initially, the home processor has lock permission.

2. On a lock operation the requesting processor is blocked if it does not currently have the lock.

3. An unlock operation on a processor forwards the data associated with the unlock to the adjoining processor. The destination processor is granted a lock for that data. If blocked on that data, it is awoken.

At the beginning of the update phase, the program invokes the Ace_ChangeProtocol routine to change the protocol to SHIFT, which as a side-effect returns all memory to a base state. The state of all objects at its home processor is set to ReadWrite. The SHIFT protocol satisfies the first condition above by assuming that a processor has a lock on an object if the object is in a ReadWrite state. When an object is unlocked, its state is set to Invalid, and the data is forwarded to the adjoining processor. At the end of the update phase, the protocol is changed back to the default protocol. This will again return all the data to a base state where home processors have write permission.

In all phases other than the update phase, a processor writes only to data owned by it. All remote data accessed is only read, not updated. The custom protocol designed for these phases, called NULL, obtains remote objects on start-of-read actions if they are not resident in the local cache, and ignores all other events. The NULL protocol can be used in place of the default protocol for the phases other than the update phase.

Results

In our experiments, we consider implementations of Water that couple the SHIFT protocol with the default protocol, as well as with the NULL protocol. Table 6.13 and Figure 6.18 summarize the results of our experiments. A system of 512 molecules was evaluated over 4 time-steps. Timing from the first iteration was discarded, the results report an average of the remaining iterations. The column heading SHIFT reports the execution times using the SHIFT protocol combined with the default protocol, while SHIFT+NULL substitutes the NULL protocol in place of the default protocol.

The speedups achieved with the default protocol are reasonable. Using the custom protocol, however, improves performance by 22% on 32 processors on the CM-5 and by 35% on the T3E. The NULL protocol does not reduce the amount of communication in the

Table 6.13: Execution times (in seconds/iteration) for Water.

	CM-5			T3E		
Procs	Default	SHIFT	SHIFT+NULL	Default	SHIFT	SHIFT+NULL
1	11.70s	11.70s	11.70s	0.514s	0.514s	0.514s
2	6.80s	7.03s	6.68s	0.505s	0.487s	0.450s
4	4.00s	3.36s	3.18s	0.267s	0.256s	0.240s
8	2.05s	1.85s	1.75s	0.146s	0.135s	0.127s
16	1.19s	1.02s	0.96s	0.063s	0.055s	0.056s
32	0.82s	0.64s	0.60s	0.055s	0.036s	0.036s

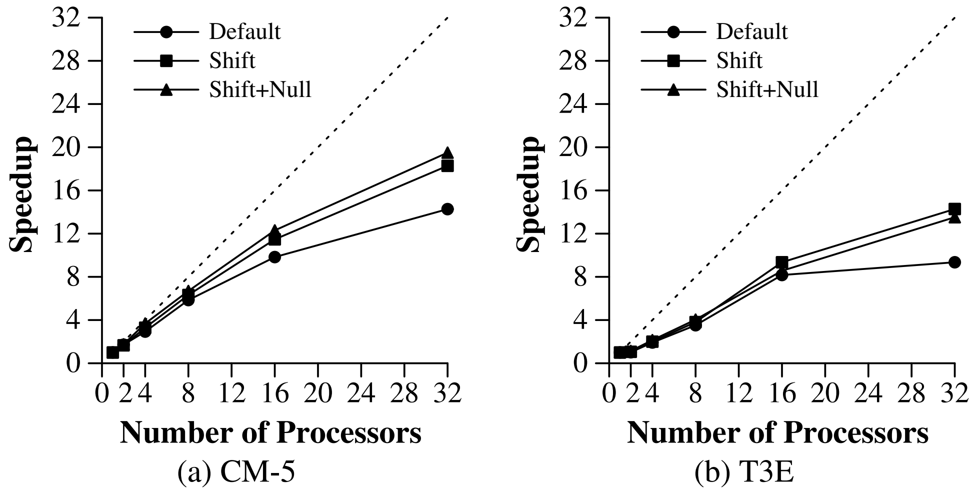


Figure 6.18: Speedup curves for Water.

program, but reduces the number of protocol invocations. Its effect is less substantial than that of the SHIFT protocol — on the CM-5, on 32 processors, the performance improvement is about 8% over using the SHIFT protocol with the default protocol, and on the T3E, its performance is about the same. As expected, its utility is higher when the number of processors is small. In these cases, the communication overhead is not as high and protocol invocation costs constitute a higher proportion of execution time.

From the results, it can be seen that on the CM-5, the execution time with the SHIFT protocol is greater than that of the default protocol on two processors. The reason for this anomaly is that the SHIFT protocol forwards data to one more processor than necessary. Since data is forwarded automatically on unlock operations, the last forward will not be used by the receiving processor. On two processors, the overhead of this extra data transfer hurts SHIFT's performance.

6.6 Evaluating the *Ace* compiler

We evaluate the performance of the *Ace* compiler by comparing the execution times of compiler-generated and hand-optimized code for our benchmarks. As mentioned earlier, the compiler-generated code is the result the compiler transforming a program written using the standard *Ace* programming model into code with calls to runtime-system routines inserted appropriately. The hand-optimized code represents standard *Ace* programs in which the programmer simulates the effect of an optimal compiler by inserting runtime-system calls explicitly where necessary. For each application, we use the protocols that have the best performance. The experiments were run on 32 processors, and we measure the execution time of unoptimized compiler-generated code and the effects of each optimization. Table 6.14 provides these results, as well as the best runtimes of hand-optimized code. The execution times of compiler-generated code range from 1.0-4.6 times slower than the corresponding hand-optimized code on the CM-5 and from 1.5 times faster to 2.1 slower on the T3E.

Profiling information collected by the *Ace* runtime system, presented graphically in Figures 6.19 and 6.20, is helpful in understanding the behavior of the compiler. The runtime system measures the number of calls to start-of-read, end-of-read, start-of-write, and

Table 6.14: Effects of compiler optimizations on benchmarks on 32 processors. LI = Loop Invariance, MC = Merging calls, DC = direct calls. All times in seconds. Hand-optimized times use customizable protocols.

Application	Platform	Base	LI	LI + MC DC	LI + MC Optimized	Hand
Barnes-Hut	CM-5	8.63s	7.76s	5.70s	5.63s	3.80s
	T3E	1.04s	0.86s	0.54s	0.53s	0.40s
BSC	CM-5	31.10s	7.13s	4.95s	4.86s	4.20s
	T3E	3.12s	1.34s	1.22s	1.02s	1.00s
EM3D	CM-5	0.46s	0.27s	0.20s	0.14s	0.12s
	T3E	0.036s	0.029s	0.025s	0.022s	0.017s
Gröbner	CM-5	2.33s	2.33s	2.35s	1.37s	0.30s
	T3E	0.87s	1.07s	0.95s	1.12s	1.73s
LU	CM-5	24.01s	5.06s	3.48s	3.03s	1.30s
	T3E	2.89s	0.61s	0.39s	0.36s	0.25s
Power	CM-5	4.06s	4.06s	3.92s	3.92s	3.90s
	T3E	0.39s	0.39s	0.36s	0.36s	0.35s
TSP	CM-5	2.06s	1.32s	1.19s	1.04s	0.80s
	T3E	0.30s	0.17s	0.16s	0.15s	0.07s
Water	CM-5	2.53s	2.53s	0.82s	0.74s	0.60s
	T3E	0.213s	0.213s	0.06s	0.056s	0.055s

end-of-write handlers in protocols. In addition, it keeps track of the number of maps performed, the number of cache misses, that is when a protocol determines that a read or a write operation cannot be satisfied immediately, and the number of dispatches to protocol event handlers (as opposed to direct invocations). The numbers are in the units of the average number of calls per processor during the execution of the program. Note that since profiling perturbs the execution behavior of programs, the results presented in Table 6.14 were collected with the profiling option turned off. The profiling data was obtained from subsequent executions of the benchmarks. As mentioned earlier, the profiling statistics were gathered on the CM-5, but the T3E values are similar.

Consider the profiling information for Barnes-Hut. The lack of difference in the number of dispatch calls between the "All optimizations" and "LI + MC" versions of Barnes-Hut in Figure 6.19 suggests that the compiler is unable to determine the spaces of accesses, and therefore, resorts to the default dispatch mechanism. In Barnes-Hut, of the three main data structures of the program — bodies, leaves, and cells — only bodies are associated with the UPDATE space; the rest are handled by the default protocol. The program, however, often casts these data structures into a generic data type, node. This casting causes the compiler to merge information about the three data structures into one. During the alias analysis phase (see Section 5.3), if a variable of type node is used to access parts of both a leaf and a cell, all space information about the two different data structures will be merged. As a result, the compiler cannot determine the space of most accesses and must insert calls to the default dispatch mechanism which determines the correct protocol handler to invoke at runtime. In Barnes-Hut, the merging calls optimization is more useful than any other in reducing the overhead of *Ace* protocol routines.

In Blocked Sparse Cholesky, the compiler does fairly well in emulating hand-optimized code. The large benefit of the loop invariance optimization is due to the presence of several heavily used computation loops that calculate various matrix products. In the absence of the loop invariance optimization, the number of runtime-system calls increases tenfold. Note that in BSC, as in most of the applications, the increase in the number of dispatches between different versions (for example, between the "All optimizations" and "LI + MC" versions) is more than the combined increase of event handler (start-of-read, end-of-read, start-of-write, and end-of-write) calls. Since dispatches to null handlers are not included in

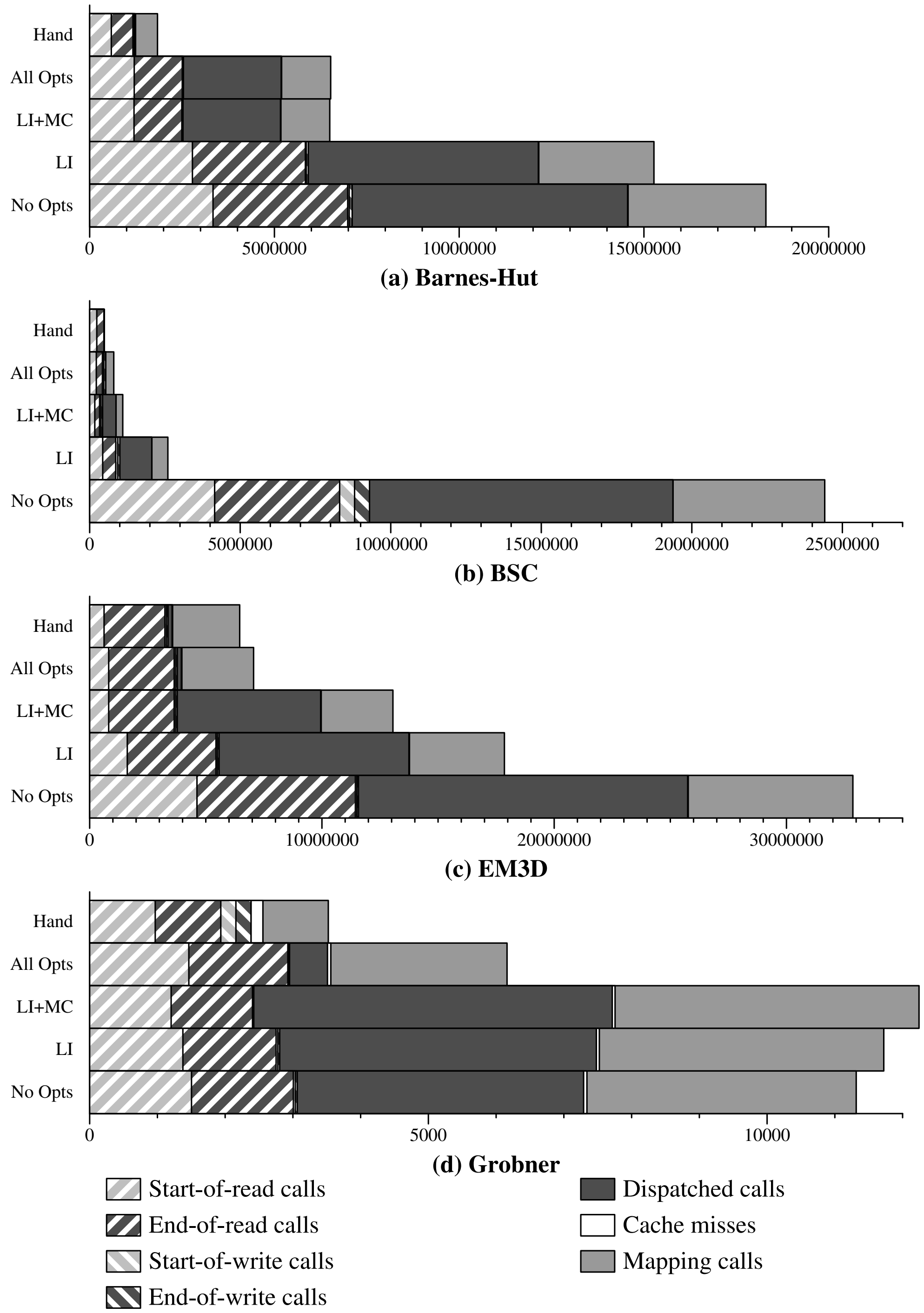


Figure 6.19: Profiling statistics for (a) Barnes-Hut (b) BSC (c) EM3D (d) Gröbner. The X-axis has units of number of calls. LI = Loop Invariance, MC = Merging calls, All Opts = Loop Invariance, Merging Calls, and Direct Calls, Hand = Hand-optimized code.

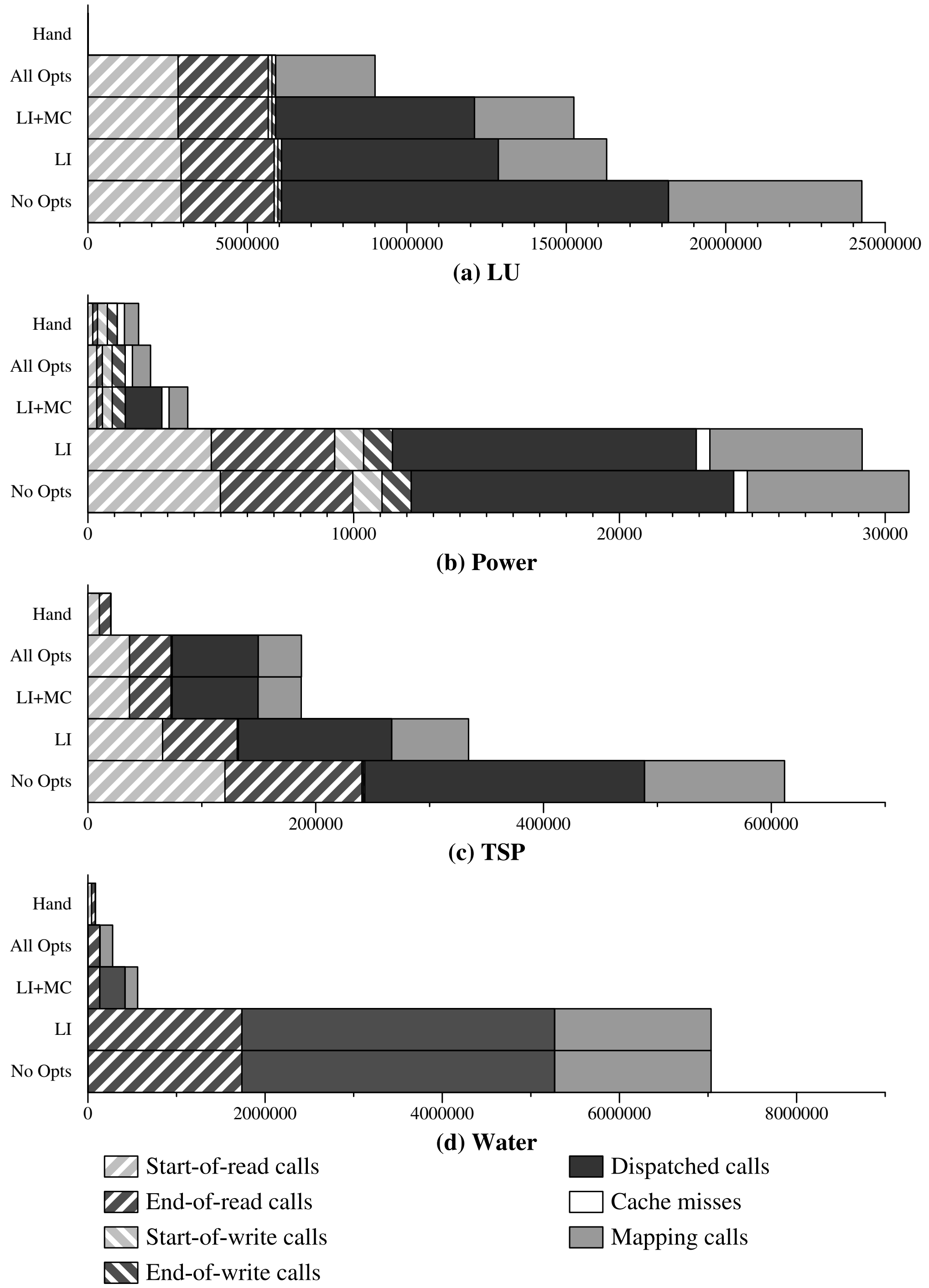


Figure 6.20: Profiling statistics for (a) LU (b) Power (c) TSP (d) Water. The X-axis has units of number of calls. LI = Loop Invariance, MC = Merging calls, All Opts = Loop Invariance, Merging Calls, and Direct Calls, Hand = Hand-optimized code.

the event handler call totals, the number of dispatches may be greater than the total number of event handler calls. The performance behavior of EM3D is similar to that of BSC. The fully optimized compiler version is reasonably similar to the hand-optimized code. The effects of the other optimizations, however, are more incremental than in BSC.

Gröbner profiling results reveal the non-determinism inherent in the algorithm. The "LI + MC" version actually performs more work than the "No optimizations" version. The results are skewed by the amount of work performed during each run, which is highly non-deterministic, and hence, do not reveal much about the quality of the compiled code. There is, however, a marked decrease in dispatches when the direct call optimization is performed. Note that there are many fewer $\mathcal{A}ce$ library calls invoked than in the previous two examples. Since the number of invocations is so low, we believe that the large difference in the execution times of compiler and hand-optimized code on the CM-5 is not a direct result of the overhead of $\mathcal{A}ce$ library calls. Instead, it appears that it is more a result of subtle differences in the order in which tasks are executed, which in Gröbner can have significant effects. On the T3E, the slower compiler-generated code actually results in a "better" ordering of the bases that are added to the set than the hand-optimized code. All the compiler versions are more efficient than the hand-optimized code!

The performance of LU factorization reveals an inadequacy in the $\mathcal{A}ce$ compiler. In the hand-optimized case, there are only 8000 calls to runtime library routine, and the bar for this value is too tiny to be visible in the profiling results! The best compiler-version has approximately 1000 times more runtime-system calls. In the application, there is a global array that stores the addresses of every block in the matrix. Once this array has been set, it is not written to again. In the hand-optimized version, the programmer can perform a map to this array, and once it has been initialized, initiate a read on the array. The compiler, however, does not optimize accesses across procedures or across synchronization points. As a result, in general, every access to this array is accompanied by $\mathcal{A}ce$ protocol calls. There are several other cases in this application where the hand-optimized version performs merges across procedures and synchronization to reduce overhead. Currently, the compiler is unable to duplicate any of these optimizations.

The number of runtime-system calls in Power, as in Gröbner, is relatively small. Power, unlike Gröbner, is not susceptible to minor perturbations, and as such, the runtime-system overhead does not play a significant role in determining execution time. The application is well-behaved, and the compiled code closely resembles the best hand-optimized code.

The performance of the optimized compiled version of TSP is reasonably close to the hand-optimized code. The inability of the compiler to to eliminate any dispatch calls in TSP is due to a specific implementation detail. The program stores the addresses of all shared variables in a global array, and each processor obtains the addresses of these variables through this array. The *Ace* compiler contracts arrays into a single object, whose properties are the union of those of its elements. Therefore, in TSP, the spaces of all elements of the global array are merged together, and the compiler cannot determine the space of any one element. If the implementation were to pass the variables's addresses to processors in a structure, or broadcast each of the addresses, this problem would not arise.

Finally, in Water, the compiler-optimized versions are similar to the hand-optimized versions. Without the merging of redundant calls, however, the performance drops severely. Notice that, in Water, as in most of the other applications, the number of cache misses is too low to be observed.

While we have not explicitly presented the effects of customizable protocols on compiled code, they can be inferred from the values in Table 6.14. Without customizable protocols, the compiler would assume that each access was handled by a sequential consistent protocol, and consequently, would not be able to perform any optimizations. The "No optimizations" results, therefore, place a lower bound on the execution times without customizable protocols. The benefits of customizable protocols are much greater than in the hand-optimized versions described in the previous section.

6.7 Discussion

As expected, the applications in our set show a wide range of access patterns. While some of these patterns could be handled by the protocols in our standard library (for example, Barnes-Hut), certain applications, such as TSP and Water, required highly specialized protocols. For Water none of the standard library protocols are appropriate for the pipelined access pattern. The ability to develop customized protocols in a clean and modular fashion is, therefore, an useful tool.

The protocols developed for our applications, generally, took a couple of hours to a day to develop and test. Customizable protocols were unnecessary for only two applications. In Power, communication plays a very small role in determining execution time, and as a result, customizable protocols would not add much to the efficiency of the application. While communication costs do play a role in BSC, the major optimization of using bulk transfer to transport blocks is available in the default version. Any further optimization does not have a significant effect. The advantage of the *Ace* abstractions are that for these applications, *Ace* may be used as any other distributed shared-memory system. When appropriate, however, a programmer has the ability to optimize performance in a clean and efficient manner.

Qualitatively, the performance benefits of customizable protocols are similar on both the T3E and the CM-5. The faster processors on the T3E and the higher bandwidth interconnect result in the T3E versions of the benchmarks outperforming the CM-5 versions significantly. The relative overheads of the compiler code on both machines are similar as well; the notable exception is Gröbner as explained in Section 6.6. The performance of the compiler-generated code with respect to hand-optimized code on the T3E is slightly better than that on the CM-5. One possible reason may be that the Alpha 21164 processor on the T3E can issue multiple instructions at a time. On the T3E, there may be more opportunities to overlap the overhead of *Ace* calls with the application's instructions. In Shasta, another software distributed shared-memory system that is implemented on a platform with Alpha processors, it has been shown that proper instruction scheduling can lower the overhead of shared-memory calls [83]. The older SPARC processor in the CM-5 offers no opportunities for such overlap.

There is room for improvement in the performance of the compiler with respect to hand-optimized code. An application developer can use semantic information about a program to deduce when accesses may be optimized across procedure calls and synchronizations. Whether a compiler can perform the same optimizations while ensuring the correctness of a program is an open question. We suspect that a deeper understanding of the interactions between protocols is necessary before such phases can be designed.

The insertion of extraneous runtime-system interface calls slows down programs by causing additional overhead. In addition, it also reduces the number of low-level optimizations (such as keeping values in registers) that our back-end can perform. Our back-end, the C compiler on the platforms, cannot distinguish *Ace* runtime-system interface calls from any other procedure calls. In general, the code for these routines is not available to the back-end. In performing analysis, the C compiler must assume that all information is killed across *Ace* runtime-system calls. Certain routines, such as the address translation call, are inlined by the C compiler, but the presence of indirections and branches in these routines reduces the C compiler's ability to perform optimizations. When the *Ace* compiler inserts extraneous calls between statements, in addition to runtime overhead, the quality of the code generated by the C compiler is hurt as well. This is especially true on the DEC Alpha 21164 based T3E, where sophisticated compiler techniques are necessary to take full advantage of the microprocessor's capabilities. A native back-end that understands the *Ace* programming model may be useful in solving this problem. Such a back-end may be able to inline protocol event handlers where appropriate, have a better understanding of the effects of *Ace* routines on registers, and may be able to schedule instructions better.

The current compiler is overly conservative in performing optimizations. For example, while interprocedural alias information is available, the merging calls phase does not use this information while deciding when an address is modified by a statement or a procedure. In addition, non-concurrency analysis [64] could be used to determine when more aggressive optimizations are possible. For example, this analysis could be used to derive more precise *kill* set information, which would allow for more aggressive merging of protocol calls. Finally, exposing more of a protocol's inner routines to a compiler may provide more opportunities for optimization. The appropriate form such a compiler/runtime-system interface should take is a subject for future study.

The difference between the *Ace* compiler-generated code and hand-optimized code is, in general, not too high. It is our belief that there are several opportunities for further optimizations and the performance gap between the compiler and hand-optimized code can be narrowed. An encouraging sign is that on the more efficient (and modern) Alpha-based Cray T3E, the compiler emulates the hand-optimized code more closely.

Chapter 7

Conclusions

The benefits of using application-specific protocols to tailor the communication mechanism to the access patterns of an application has been shown previously [34, 67]. Architectures have been designed that provide hardware support for customizable protocols [53, 76]. Yet, the viability of using customizable protocols on these architectures, and on other types of shared-memory systems, is predicated on the availability of a system that supports easy use of customizable protocols.

We have designed a language, $\mathcal{A}ce$, that provides language support for customizable protocols. A key benefit of $\mathcal{A}ce$ is that it allows programmers to separate the communication component of a program from the algorithmic part. Applications can be developed under a shared-memory abstraction, using a default, well understood protocol. Programmers can then optimize applications incrementally by experimenting with different protocols to handle accesses to data structures. The association of protocols with data structures is done using high-level language constructs. $\mathcal{A}ce$ programmers may use a set of standard protocols provided by the system, or extend the system easily by designing custom protocols. The design of the constructs in the language was driven by a detailed study of the issues involved in designing a language for customizable protocols.

We have implemented our language as a software distributed shared-memory system on a Thinking Machines CM-5 and a Cray T3E by building a compiler and supporting runtime systems. The runtime system performs most of the work in implementing the $\mathcal{A}ce$ programming model. It simulates a global shared memory through software routines that perform appropriate address translations and caching. It supplies routines for manipulating the states of objects and communicating that can be used by protocol developers. It also ensures that a program's accesses to shared memory are dispatched to the appropriate protocol automatically. The runtime system supports a well defined interface, *full access control*, for protocol development. A protocol can specify handlers that are executed before and after accesses and at synchronization points. The runtime system also defines an interface, based on full access control, that is used by the compiler to transform programs into code that can be linked with the runtime system and the protocol libraries to form an executable.

The compiler instruments *Ace* programs with runtime system calls. For *Ace* programs to be efficient, however, it is essential that the compiler perform optimizations. We have designed and implemented analyses and optimizations that allow us to reduce the overhead of software shared memory. Our compiler moves runtime system calls out of loops, merges redundant calls, and avoids the protocol dispatch process in the runtime system, where possible.

The performance of our system has been validated through careful experimentation. The costs of various runtime primitives were isolated using micro-benchmarks. A thorough analysis of the access patterns of eight benchmarks led to the design of several new custom protocols. Using customized protocols resulted in a speedup of 1.0 to 6.6 on the CM-5 and 1.0 to 8.2 on the T3E for our benchmarks. The speedup of any program using customized protocols, however, is highly application-specific and cannot be predicted accurately from the speedup range provided above. The strengths and weaknesses of the compiler was studied in detail through the use of profiling data.

Future Work

While *Ace* makes the task of developing and using customized protocols easier, a future goal is to automate the process of selecting protocols. One possible technique is to use runtime profiling information as a basis for suggesting new protocols. Alternatively, the compiler could determine when a protocol chosen by a programmer is inappropriate for an access pattern. These optimizations require a more thorough understanding of the structure of protocols and access patterns of applications. A taxonomy of protocols would be helpful in deciding if a particular access pattern is compatible with a protocol, or if changing from one protocol to another is correct. A more formal specification of *Ace* protocols would be necessary before a compiler could understand the semantics of protocols directly.

Another issue that requires further study is that of whether one can derive a basis set of protocols that is sufficient for all applications. FLASH [53] has hardware support for customizable protocols, but does not allow users to extend the system with new protocols. On the other hand, Typhoon [76] allows user-level specification of protocols, which though more flexible, may suffer from a host of protection problems. Which of the two is the right abstraction is an open question. If a single basis set is sufficient, what should the set of protocols be? For software shared-memory systems, the question is not as important, since it is easy to allow programmers to extend the system with new protocols. On hardware systems, issues such as protection must be taken into consideration.

Implementing *Ace* on systems with hardware support for customizable protocols [53, 76] would be an interesting avenue of further research. New techniques are required to ensure that the hardware and software protocols interact correctly. An implementation on a shared virtual memory platform is of interest as well. For an application, certain data structures may wish to use the large granularity shared virtual memory mechanism, while for others, a more fine-grained software shared-memory protocol may be appropriate. The tradeoffs in using the two approaches in the same application are not known. The issue of compiler support for explicitly parallel shared-memory systems is relatively unexplored. Developing techniques for propagating data flow more accurately is of great importance. A deeper understanding of the interactions among the access patterns in parallel applications may shed light on the types of techniques that could be used to optimize these applications correctly and efficiently.

Summary

We have designed a language that allows programmers to use customizable protocols intuitively. It provides complete separation of application and protocol code, high-level language abstractions for associating protocols with data structures, user-specified granularity, and can be extended easily with new protocols. We have implemented a compiler that automatically generates annotations similar to those of region-based systems, such as CRL.

We have developed optimizations to improve the performance of the compiler, and found that $\mathcal{A}ce$ performs respectably compared to handwritten code. The performance of $\mathcal{A}ce$ applications is enhanced significantly by the use of customized protocols. The $\mathcal{A}ce$ system represents a step towards understanding the abstractions necessary for easy implementation of parallel applications. The abstractions and techniques developed during the design and implementation of $\mathcal{A}ce$ should serve as a good basis for studies that attack other aspects, such as the task model, of the design space of efficient, easy-to-use, parallel systems.

Bibliography

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. Technical Report ECE-9512, Rice University, Sept. 1995.

[2] Y. Afek, G. Brown, and M. Merritt. Lazy caching. *ACM Transactions on Programming Languages and Systems*, 15(1):182–205, 1993.

[3] A. Agarwal, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B.-H. Lim, G. Maa, and D. Nussbaum. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In M. Dubois and S. S. Thakkar, editors, *Scalable Shared Memory Multiprocessors*, pages 239–261. Kluwer Academic Publishers, 1992.

[4] G. Agrawal, A. Sussman, and J. Saltz. An integrated compile-time and runtime approach for parallelizing structured and block structured applications. *IEEE Transactions on Parallel and Distributed Systems*, 6(7):747–754, July 1995.

[5] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[6] 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*, pages 18–28, Feb. 1996.

[7] E. Anderson, J. Brooks, and T. Hewitt. The benchmarker's guide to single-processor optimization for CRAY T3E systems. Benchmarking Group, Cray Research, 1997.

[8] T. Anderson, D. Culler, and D. Patterson. A case for NOW (network of workstations). *IEEE Micro*, 15(1):54–56, Feb. 1995.

[9] A. W. Appel and K. Li. Virtual memory primitives for user programs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems,* in *SIGPLAN Notices*, pages 96–107, Apr. 1991.

[10] R. H. Arpaci, D. E. Culler, A. Krishnamurthy, S. Steinberg, and K. Yelick. Empirical evaluation of the Cray-T3D: A compiler perspective. In *Proceedings of the International Symposium on Computer Architecture*, pages 320–331, 1995.

[11] J. Barnes and P. Hut. A hierarchical O(N log N) force calculation algorithm. *Nature*, pages 446–449, Dec. 1986.

[12] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Symposium on Principles and Practice of Parallel Programming*, pages 168–176, Mar. 1990.

[13] B. Bershad, M. Zekauskas, and W. A. Sawdon. The Midway distributed shared memory system. In *Proceedings of COMPCON'93*, pages 528–537, Feb. 1993.

[14] M. Blumrich et al. Virtual memory mapped network interface for the SHRIMP multicomputer. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 142–153, Apr. 1994.

[15] J. B. Carter. Design of the Munin distributed shared memory system. *Journal of Parallel and Distributed Computing*, 29(2):219–227, Sept. 1995.

[16] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Symposium on Operating Systems Principles*, pages 152–164, Oct. 1991.

[17] S. Chakrabarti, E. Deprit, E.-J. Im, J. Jones, A. Krishnamurthy, C.-P. Wen, and K. Yelick. Multipol: A distributed data structure library. Technical Report CSD-95-879, University of California, Berkeley, July 1995.

[18] S. Chakrabarti and K. Yelick. Implementing an irregular application on a distributed memory multiprocessor. In *Symposium on Principles and Practice of Parallel Programming*, pages 169–178, May 1993.

[19] R. Chandra, K. Gharachorloo, V. Soundararajan, and A. Gupta. Performance evaluation of hybrid hardware and software distributed shared memory protocols. In *International Conference on Supercomputing*, pages 274–288, July 1994.

[20] R. Chandra, A. Gupta, and J. Hennessy. Data locality and load balancing in COOL. In *Principles and Practice of parallel programming*, May 1993.

[21] S. Chandra, B. Richards, and J. Larus. Teapot: Language support for writing memory coherence protocols. In *Conference on Programming Language Design and Implementation*, pages 237–248, May 1996.

[22] M. Chandy and C. Kesselman. A declarative, concurrent object oriented programming notation. Technical Report CS-92-01, California Institute of Technology, Pasadena, Ca, 1992.

[23] D. Chase, M. Wegman, and K. Zadeck. Analysis of pointers and structures. In *Conference on Programming Language Design and Implementation*, pages 296–310, June 1990.

[24] D. Culler et al. LogP: Towards a realistic model of parallel computation. In *Symposium on Principles and Practice of Parallel Programming*, pages 1–12, May 1993.

[25] D. Culler et al. Parallel programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, Nov. 1993.

[26] D. Culler and J. P. S. with Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, To be published. Draft available online at http:://www.cs.berkeley.edu/ culler/book.alpha/index.html.

[27] F. Dahlgren. Boosting the performance of hybrid snooping cache protocols. In *Proceedings of the International Symposium on Computer Architecture*, June 1995.

[28] R. Das, M. Uysal, J. Saltz, and Y.-S. Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22:462–478, 1994.

[29] D. Dill, A. Drexler, A. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, 1992.

[30] J. Dongarra, S. Otto, M. Snir, and D. Walker. An introduction to the MPI standard. Oakridge National Laboratory, 1995.

[31] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenström. The detection and elimination of useless misses in multiprocessors. In *Proceedings of the International Symposium on Computer Architecture*, pages 88–97, June 1993.

[32] S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. An integrated compile-time/run-time software distributed shared memory system. In *Proceedings of Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 186–197, Oct. 1996.

[33] S. Eggers and R. Katz. A characterization of sharing in parallel programs and its application to coherency protocol evaluation. In *Proceedings of the International Symposium on Computer Architecture*, pages 373–383, 1988.

[34] B. Falsafi, A. Lebeck, S. Reinhardt, I. Schoinas, M. Hill, J. Larus, A. Rogers, and D. Wood. Application-specific protocols for user-level shared memory. In *Proceedings of Supercomputing '94*, pages 380–389, Nov. 1994.

[35] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Technical Report COMP-TR90-141, Dept. of Computer Science, Rice University, Houston, TX, Dec. 1990.

[36] C. Fraser and D. Hanson. *A Retargetable C Compiler: Design and Implementation*. The Benjamin/Cummings Publishing Company, Inc., first edition, 1995.

[37] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine – A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1995. Available online at http://www.netlib.org/pvm3/book/pvm-book.html.

[38] K. Gharachorloo, A. Gupta, and J. Hennessey. Hiding memory latency using dynamic scheduling in shared-memory multiprocessors. In *Proceedings of the International Symposium on Computer Architecture*, pages 22–33, May 1992.

[39] Greenberg. On the time complexity of broadcast communication schemes. In *STOC: ACM Symposium on Theory of Computing (STOC)*, 1982.

[40] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[41] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston, TX, May 1992.

[42] M. D. Hill, J. R. Larus, S. K. Reinhardt, and D. A. Wood. Cooperative shared memory: Software and hardware for scalable multiprocessors. *ACM Transactions on Computer Systems*, 11(4):300–318, Nov. 1993.

[43] Y.-S. Hwang, B. Moon, S. Sharma, R. Ponnuswamy, R. Das, and J. Saltz. Runtime and language support for compiling adaptive irregular programs on distributed memory machines. *Software — Practice and Experience*, 25(6):597–621, June 1995.

[44] IBM. SP-2 command and technical reference. IBM Corporation, Dec. 1994.

[45] T. E. Jeremiassen and S. J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *Symposium on Principles and Practice of Parallel Programming*, pages 179–188, July 1995.

[46] K. Johnson, J. Adler, and S. Gupta. *CRL 1.0 Software Distribution*, August 1995. Available on World Wide Web at www.pdos.lcs.mit.edu/crl.

[47] K. Johnson, F. Kaashoek, and D. Wallach. CRL: High-performance all-software distributed shared memory. In *Symposium on Operating Systems Principles*, Dec. 1995.

[48] M. F. Kaashoek, A. S. Tanenbaum, H. Flynn, and H. E. Bal. An Efficient Reliable Broadcast Protocol. *Operating Systems Review*, 23:5–19, Oct. 1989.

[49] R. M. Karp, A. Sahay, E. E. Santos, and K. E. Schauser. Optimal broadcast and summation in the LogP model. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 142–153, June 30–July 2, 1993.

[50] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the International Symposium on Computer Architecture*, pages 13–21, May 1992.

[51] R. Kessler and J. Schwarzmeier. Cray T3D: A new dimension in Cray Research. In *COMPCON Spring '93*, pages 176–182, Feb. 1993.

[52] A. Krishnamurthy. Optimizing explicitly parallel programs. Technical Report CSD-94-835, University of California, Berkeley, Ca, Sept. 1994.

[53] J. Kuskin et al. The Stanford FLASH multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, Apr. 1994.

[54] T. Kwan, B. Totty, and D. Reed. Communication and computation performance of the CM-5. In *Supercomputing '93*, Nov. 1993.

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

[56] J. R. Larus, B. Richards, and G. Viswanathan. LCM: Memory system support for parallel language implementations. In *Proceedings of Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 208–218, Nov. 1994.

[57] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA highly scalable server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 241–251, June 1997.

[58] A. Lebeck and D. Wood. Active memory: A new abstraction for memory system simulation. In *SIGMETRICS'95*, May 1995.

[59] C. Leiserson et al. The network architecture of the Connection Machine CM-5. In *Symposium on Parallel and Distributed Algorithms*, pages 272–285, June 1992.

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

[61] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford DASH multiprocessor. *IEEE Computer*, 25(3):63–79, Mar. 1992.

[62] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, Nov. 1989.

[63] S. Lumetta, L. Murphy, X. Li, D. Culler, and I. Khalil. Decentralized optimal power pricing: The development of a parallel program. In *Supercomputing '93*, pages 240–249, Nov. 1991.

[64] S. Masticola and B. Ryder. Non-concurrency analysis. In *Symposium on Principles and Practice of Parallel Programming*, pages 129–138, May 1993.

[65] P. K. McKinley, Y.-J. Tsai, and D. F. Robinson. Collective communication in wormhole-routed massively parallel computers. *Computer*, 28(12):39–50, Dec. 1995.

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

[67] S. Mukherjee, S. Sharma, M. Hill, J. Larus, A. Rogers, and J. Saltz. Efficient support for irregular applications on distributed-memory machines. In *Symposium on Principles and Practice of Parallel Programming*, volume 30(8), pages 68–79, Aug. 1995.

[68] R. Netzer. *Race Condition Detection for Debugging Shared-Memory Parallel Programs*. PhD thesis, Computer Sciences Departiment, University of Wisconsin – Madison, Aug. 1991.

[69] R. Netzer and B. Miller. What are race conditions? some issues and formalizations. *ACM Letters on Programming Languages and Systems*, 1(1):74–88, Mar. 1992.

[70] R. S. Nikhil. Cid: A parallel, shared-memory C for distributed-memory machines. In *Languages and Compilers for Parallel Computing*, pages 376–390, Aug. 1994.

[71] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Professional Computing Series. Addison-Wesley, 1994.

[72] M. Raghavachari and A. Rogers. Understanding language support for irregular parallelism. In T. Ito, R. H. Jr., and C. Queinnec, editors, *Parallel Symbolic Languages and Systems*, volume LNCS 1068, pages 174–189. Springer, Oct. 1995.

[73] M. Raghavachari and A. Rogers. A study of the effects of ordering, partitioning, and factorizations algorithms on distributed sparse cholesky factorization. Technical Report TR-505-96, Princeton University, Princeton, NJ, Jan 1996.

[74] M. Raghavachari and A. Rogers. Ace: Linguistic mechanisms for customizable protocols. In *Symposium on Principles and Practice of Parallel Programming*, pages 80–89, June 1997.

[75] S. Reinhardt. Tempest interface specification. Technical Report TR 1267, Computer Science Department, University of Wisconsin, Madison, WI, Feb. 1995.

[76] S. Reinhardt, J. Larus, and D. Wood. Tempest and Typhoon: user-level shared memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–337, Apr. 1994.

[77] B. Richards. *Memory Systems for Parallel Programming*. PhD thesis, Computer Sciences Departiment, University of Wisconsin – Madison, 1996.

[78] M. Rinard. *The design, implementation and evaluation of Jade: a portable, implicitly parallel programming language*. PhD thesis, Stanford University, Aug. 1994.

[79] A. Rogers, M. Carlisle, L. Hendren, and J. Reppy. Supporting dynamic data structures on distributed memory machines. *ACM Transactions on Programming Languages and Systems*, 17(2):233–263, Mar. 1995.

[80] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Conference on Programming Language Design and Implementation*, pages 69–80, 1989.

[81] E. Rothberg. *Exploiting the memory hierarchy in sequential and parallel sparse Cholesky factorization*. PhD thesis, Stanford University Department of Computer Science, Jan. 1993.

[82] E. Rothberg, J. P. Singh, and A. Gupta. Working sets, cache sizes and node granularity for large-scale multiprocessors. In *Proceedings of the International Symposium on Computer Architecture*, pages 189–199, June 1993.

[83] D. J. Scales, K. Gharachorloo, and C. Thekkath. Shasta: A low-overhead software-only approach for supporting fine-grain shared memory. In *Proceedings of Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 174–185, Oct. 1996.

[84] D. J. Scales and M. S. Lam. The design and evaluation of a shared object system for distributed memory machines. In *Proceedings of Symposium on Operating Systems Design and Implementation*, pages 101–114, Nov. 1994.

[85] I. Schoinas, B. Falsafi, A. Lebeck, S. Reinhardt, J. Larus, and D. Wood. Fine-grain access control for distributed shared memory. In *Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297–307, Nov. 1994.

[86] S. L. Scott. Synchronization and communication in the T3E multiprocessor. In *Proceedings of Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 26–36, Oct. 1996.

[87] J. P. Singh, J. L. Hennessy, and A. Gupta. Implications of hierarchical n-body methods for multiprocessor architecture. *ACM Transactions on Computer Systems*, 13(2):141–202, May 1995.

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

[89] B. Steensgaard. Points-to analysis by type inference of programs with structures and unions. In *International Conference on Compiler Construction*, volume 1060, pages 136–150. Springer, Apr. 1996.

[90] B. Steensgaard. Points-to analysis in almost linear time. In *Principles of Programming Languages*, pages 32–41, Jan. 1996.

[91] C. Tang. Cache design in the tightly coupled multiprocessor system. In *AFIPS Conference Proceedings, National Computer Conference*, pages 749–753, 1976.

[92] R. E. Tarjan. Data structures and network flow algorithms. *SIAM Regional Conference Series in Applied Mathematics*, CMBS 44, 1983.

[93] J. Torellas, M. Lam, and J. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Computer*, 43(6):651–663, June 1994.

[94] K.-P. Vo. Vmalloc: A general and efficient memory allocator. *Software — Practice and Experience*, 26(3):357–374, Mar. 1996.

[95] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active Messages: a mechanism for integrated commmunication and computing. In *Proceedings of the International Symposium on Computer Architecture*, pages 256–266, May 1992.

[96] W.-D. Weber. *Scalable Directories for Cache-Coherent Shared-Memory Multiprocessors*. PhD thesis, Stanford University, Jan. 1993.

[97] R. Wilson et al. The SUIF compiler system: a parallelizing and optimizing research compiler. Technical Report CSL-TR-94-620, Computer Systems Lab, Stanford University, Palo Alto, CA, 1994.

[98] W. C. Yen, D. W. Yen, and K.-S. Fu. Data coherence problem in a multicache system. *IEEE Transactions on Computers*, C-34(1):56–65, Jan. 1985.

[99] M. Zekauskas, W. Sawdon, and B. Bershad. Software write detection for a distributed shared memory. In *Proceedings of Symposium on Operating Systems Design and Implementation*, pages 87–100, Nov. 1994.