Ace: Linguistic Mechanisms for Customizable Protocols

Mukund Raghavachari
Department of Computer Science
35 Olden Street
Princeton University
Princeton, NJ 08540
mrm@cs.princeton.edu

Anne Rogers
AT&T Labs-Research
600 Mountain Avenue
PO Box 636
Murray Hill, NJ 07974
amr@research.att.com

Abstract

Customizing the protocols used to manage accesses to different data structures within an application can improve the performance of shared-memory programs substantially [10, 21]. Existing systems for using customizable protocols are, however, 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.

We have designed a language, *Ace*, that integrates support for customizable protocols with minimal extensions to C. *Ace* applications are developed using the standard 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. In this paper, we isolate the issues involved in providing language support for using customizable protocols and present novel language abstractions for their easy use. We describe the implementation of a compiler and runtime system for *Ace*, and discuss the issues involved in their design. We also present measurements that demonstrate that *Ace* has good performance compared to an efficient software distributed shared-memory system.

1 Introduction

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. Most shared-memory systems, however, can 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 [10].

Previous work has shown that by tailoring the protocols used to handle accesses to data to the access patterns of the data structures, the performance of shared-memory programs can be improved substantially [10, 21]. It is important, therefore, for shared-memory systems to support *customizable protocols*, that is, allow programmers the ability to optimize communication performance by managing the protocols that are used for accesses to different data within an application. Existing systems for customizable protocols, such as Tempest [25], 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 [1, 3, 15, 18, 25], the development of intuitive abstractions for the use of customizable protocols is important.

We have designed a language, *Ace*, that integrates support for customizable protocols with minimal extensions to C. *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 [25]. 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.
- *High-level language abstractions for associating protocols with data structures:* Current systems require programmers to associate protocols with pages and to

PPoPP '97 Las Vegas, NV

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 implicitly set the granularity of coherence of data 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 a means of taking advantage of a bulk transfer mechanism.
- *Extensibility:* *Ace* offers a clean interface for adding new protocols to the system.

In this paper, we describe the design and implementation of *Ace*, and compare its performance with that of CRL [14], an efficient software distributed shared-memory system (DSM). We isolate the issues involved in providing language support, and present novel language abstractions for the easy use of customizable protocols (Section 2). We describe the implementation of a compiler and runtime system for *Ace*, and discuss the issues involved in their design (Sections 3 and 4). Finally, we present measurements, which demonstrate that *Ace* has good performance compared to an efficient software DSM (Section 5).

In addition to examining the issue of language and compiler support of customizable protocols, a secondary goal of this paper is to introduce a new method of implementing fine-grained shared memory on distributed-memory machines. Previous research has concentrated either on using virtual memory [4, 6, 18], or link-time object-code rewriting [28, 30] to implement shared memory. In contrast, we leverage compiler technology to achieve high performance with *Ace*. Without the use of customizable protocols, *Ace*'s performance is comparable to that of CRL. Using customizable protocols, *Ace* outperforms CRL significantly. *Ace* is portable to any system that supports an Active Messages [32] mechanism.

1.1 Related Work

Tempest [25] provides support for developing application-specific protocols, but 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* (the ability to specify protocols that execute on memory faults to data) hinder the creation of protocol libraries, and complicate application development (See Section 2.1).

Munin [6] provides a fixed set of protocols that programmers can associate with data structures, but provides no facility for customization of protocols. In addition, the large coherence unit of a page makes false-sharing and induced sharing patterns [12] primary concerns. Chandra *et al.* [7] study the performance of a hybrid protocol — a combination of a software DSM and hardware shared memory. They do not consider the use of customizable protocols in their system.

Several other shared-memory systems offer a single or few fixed protocols [4, 9, 18, 22, 14, 29, 28]. Shasta [28] supports fine-grain shared-memory by processing object files at link-time. However, much information that is available at the language level is lost by link-time. 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 link-time, 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.

Region-based systems, such as CRL [14], require programmers to insert annotations by hand. In CRL, shared variables all have the same type — casts must be inserted explicitly where appropriate. The presence of compile-time type checking makes *Ace* considerably easier to use. Shasta, CRL, and *Ace*, are similar in that they support user-specified granularity.

Teapot [8] is a system that facilitates the design of protocols. Teapot is complementary to *Ace*, and can be modified to be used by *Ace* protocol designers.

Alewife [1], FLASH [15], SHRIMP [5] and Typhoon [25] are architectures that provide some support for running shared-memory protocols in software. We discuss how *Ace* could be used on these architectures in Section 6.

2 Application-specific Protocols

In this section, we discuss various issues related to the use of customizable protocols and the design of language abstractions to support them.

2.1 Encapsulation of Communication Complexity

For ease of use of customizable protocols, complete separation of application and protocol code is essential. Encapsulation of protocols allows independent development and testing of applications and protocols, and the creation of protocol libraries. During the application development process, programmers can use a standard protocol, and then optimize performance by associating efficient protocols with data structures. With protocol libraries, common protocols such as update protocols, migratory protocols, etc. can be reused by many applications.

To allow the encapsulation of protocol code, the application/protocol interface must provide sufficient "hooks" to allow the implementation of most common protocols. The diverse nature of applications makes it difficult to predict when protocol actions may be performed. For example, an invalidate protocol requires invalidations to be performed *before* writes. 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.*[16], 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[9], must check that all outstanding memory operations have completed. Certain static update protocols propagate updates to remote cached copies of data at barriers.

Since protocols may have to be executed before and after accesses, and at synchronization points, *Ace* provides *full access control*, the ability to specify the protocol actions to be executed at each of these points. 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. To implement several common protocols with access-fault control, protocol designers must either violate the encapsulation of protocols or circumvent 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, 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.

2.2 Application/Protocol Interactions

To understand what abstractions allow programmers to manipulate protocols easily, we now consider issues relating to how applications use protocols:

Associating Protocols with Data:
Protocols do not always operate on each logical datum independently, but often handle a collection of data. For example, a static update protocol collects all writes to a particular data structure, and then, propagates these updates at barriers. One may also require separate instances of the same protocol to operate on different data structures (Section 3.3 provides an example). A language for a customizable protocols requires a mechanism for associating protocols with a collection of data, such as a data structure. We address this issue by providing a high-level abstraction, called *spaces*. A space manages a subset of the address space and handles all allocations, accesses and synchronization to data within it. To associate a protocol with a data structure, a programmer declares a space, associates a protocol with the space (the default is a sequentially consistent protocol), and then allocates all nodes of the data structure from that space.

Changing Protocols:
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 [31], the program alternates between phases where intra-processor and inter-processor calculations are made. We have found that shifting between a null protocol for the intra-processor phase, and an update protocol tailored to the communication pattern of the inter-processor phase has a speedup of two over a sequentially consistent execution (using *Ace*). Since each of these protocols make assumptions about the access patterns of their phases, neither could be used independently for the whole application.

Bennett *et al.* [3] 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. Since the space abstraction is an indirection for associating protocols with data structures, changing the protocol of a data structure in *Ace* is as simple as changing the protocol associated with its space. To our knowledge, no other system offers this capability.

2.3 Granularity

The issue of granularity in a DSM is, in a sense, orthogonal to the issue of using customizable protocols. The issues and solutions mentioned above apply equally to systems of page or fine-grained granularity. Fixed-size granularity, however, complicates the use of customizable protocols. Implicitly assigning data to fixed-size coherence units forces programmers to worry about about false sharing of data, and creates the problem of *false sharing of protocols*. False sharing of protocols can appear in two forms. First, data with different access patterns may reside in the same coherence unit. This problem can be solved by using the space abstraction to locate data with different access patterns in different coherence units.

Second, 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. However, 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. *Ace* alleviates this problem by operating at user-specified granularities. Data is shared using arbitrarily-sized *regions*. Maintaining coherence at the granularity of regions diminishes the problems, such as false sharing, of implicitly associating coherence at fixed size, and provides an intuitive mechanism for using bulk transfer.

2.4 Extensibility

In our experience, we have found producer-consumer protocols to be common. While systems such as Midway [4] and Munin handle these protocols in a fixed manner, 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 these 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 *Ace*, 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 [8], can be modified to be used with *Ace* to alleviate this burden.

3 The *Ace* Language

In the previous section, 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. We outline the programming model of the language in Section 3.1, discuss the protocol interface in Section 3.2, and describe a sample *Ace* program in Section 3.3.

Table 1: Sample *Ace* declarations

`shared int *x`	pointer to shared integer
`shared int A[10]`	array of ten shared integers
`shared int * shared A[10]`	array of ten shared pointers to shared integers

Table 2: *Ace* Library routines

Ace Library Routines	Description
`Ace_GMalloc(space, size)`	Allocate a new region from *space*
`Ace_NewSpace(protocol)`	Create a new space
`Ace_ChangeProtocol(space, protocol)`	Change the protocol of a space
`Ace_Barrier(space)`	Execute a barrier operation with the semantics specified by *space*
`Ace_Lock(region)`	Lock a region
`Ace_UnLock(region)`	Unlock a region

3.1 Programming Model

Ace is essentially C with minor modifications and a set of runtime libraries. The one additional construct in *Ace* is that programmers must annotate global data with the keyword **shared**. In addition, all global data must be allocated dynamically off the heap. *Ace* disallows arithmetic on pointers to shared data unless the result is dereferenced immediately. As a result, array references are allowed, but it is not possible to generate a pointer that points into the middle of a shared region (all pointers point to the base of shared regions). Table 1 gives a few examples of *Ace* declarations. Table 2 lists the *Ace* library routines.

The execution model supports a single user thread per processor (SPMD). Synchronization routines such as barriers and locks are provided by protocols, with default routines provided by the system. *Spaces* and *Protocols* are predefined types in *Ace*. Spaces are defined by calling `Ace_NewSpace`. The `Ace_GMalloc` routine is used to allocate shared data *regions* from a space. If desired, programmers can use the default space, which provides a sequentially consistent invalidation-based protocol.

The protocol associated with a space can be changed with the `Ace_ChangeProtocol` routine. The semantics of the change are defined by the old protocol of the space. This generally involves manipulating objects into a "base" state, and then calling the initialization routine of the new protocol. For example, changing from the default protocol to any other protocol results in all cached regions being flushed back to their home processors.

3.2 Protocol Interface

Ace supports *full access control*, the ability to specify protocols that execute before and after accesses and at synchronization points. A protocol designer writes functions for each of these points — before write, after write, before read, after read, barrier, lock, and unlock. If desired, null or default protocol routines may be specified for these routines. The protocol is then registered with the system by running a Tcl/Tk [23] script. The script generates a system configuration file that is used by the *Ace* compiler to determine the protocols available and the names of the functions used by the protocol. This scheme allows new protocols to be added to the system easily. Currently, protocol libraries must be explicitly linked in by the programmer. This process will be automated in the near future.

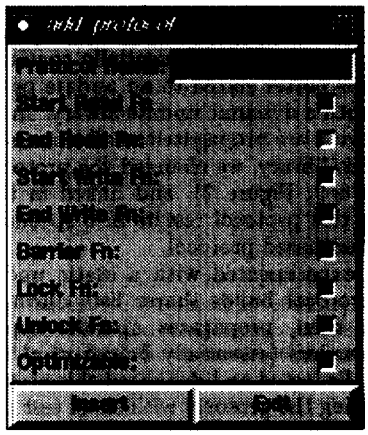

Figure 1: Adding a protocol

Figure 1 gives an example of protocol addition. The protocol writer specifies the name of the protocol, and at what access and synchronization points protocol routines must be invoked. The compiler derives the names of the protocol routines by concatenating the name of the protocol with the execution points (for example, `Update_StartRead`). The last field specifies whether calls to the protocol can be optimized (Section 4.2 discusses this parameter in greater detail).

3.3 Example — EM3D

EM3D [9], an application with static access patterns, models the propagation of electromagnetic waves through three dimensions. The data structure is a bipartite graph, with directed edges between the two sets of nodes, `E` and `H`. In each iteration, new `E` values are computed as a weighted sum of neighboring `H` nodes, after which, new `H` values are

```
1    #include "ace.h"
2    Space eval = Ace_NewSpace(SC);
3    Space hval = Ace_NewSpace(SC);
4
5    /* Allocate regions from spaces and create graph */
6    MakeGraph();
7
8    Ace_ChangeProtocol(hval, Update);
9    Ace_ChangeProtocol(eval, Update);
10
11   /* Begin computation - iterate over nodes */
12   for (t = 0; t < time_steps; t++) {
13       all_compute(e_nodes, my_e_nodes, hval);
14       Ace_Barrier(eval);
15       all_compute(h_nodes, my_h_nodes, eval);
16       Ace_Barrier(hval);
17   }
```

Figure 2: Sample code from EM3D

calculated from neighboring E nodes. Figure 2 shows code for this application.

In our implementation of EM3D, we allocate two spaces, one for E values, and one for H. The program was developed assuming the default sequentially consistent protocol. Since EM3D is better suited to an update protocol, we experimented with a dynamic update library. In this protocol, writes to a region are propagated to all sharers immediately. To plug in this library, we changed the protocol of the two spaces (lines 8–9, Figure 2), and linked in the update library. Using this protocol results in a speedup of 3.5 over the invalidation-based protocol.

We then experimented with a static update protocol. The static protocol builds sharer lists during the first iteration, and then, propagates updates appropriately at subsequent barriers (essentially Falsafi *et al.*'s protocol for EM3D[10]). We tested and debugged this protocol independently by linking the protocol with short test programs that exercised it. Again, the only changes necessary to use this library were to call `Ace_ChangeProtocol` and to link in the new library. Since the barriers specify the space they operate on, the underlying system invokes the static update barrier handler routine automatically. This protocol results in a speedup of about five over the invalidation-based protocol.

4 *Ace* Compiler and Runtime System

In this section, we describe the runtime system that is the target platform for the compiler. We then describe the design and implementation of the compiler and its optimization phases.

4.1 Runtime System

The programming model of the *Ace* runtime system is region-based [27], and is similar to that of CRL [14]. We use a system similar to CRL as the target of our compiler because the CRL programming model has been shown to be portable, and have good performance on distributed memory architectures [14]. We, therefore, expect the *Ace* runtime system to be easily portable as well.

In the *Ace* runtime system, each shared *region* (created by using `Ace_GMalloc`) is associated with a unique id. Before accessing any region, a processor must ensure that it is mapped. A map allocates memory for a cached copy of the region (if necessary), and returns a pointer to the region. Essentially, a map translates a unique *Ace* id into a local pointer for the associated region. Accesses to regions use this local pointer. `ACE_START_READ` and `ACE_START_WRITE` calls mark the beginning of reads and writes to a region. Similar calls exist to signal the termination of these accesses. Figure 3 summarizes the annotations required by the *Ace* runtime system, and Figure 4 gives an example of its use.

Ace Primitives	Description
`ACE_MAP`	Map a region into local address space
`ACE_UNMAP`	Unmap a mapped region
`ACE_START_READ`	Signal start of read on a region
`ACE_END_READ`	Signal end of read
`ACE_START_WRITE`	Signal start of write on a region
`ACE_END_WRITE`	Signal end of write

Figure 3: *Ace* runtime annotations.

```
/* Factor a column */
void FactorColumn (int colno) {
    address_t id = Matrix[colno]→nz_address;
    double *column, factor;
    int i;
    int column_length = Matrix[colno]→len;

    /* Map column and initiate write */
    column = ACE_MAP(id);
    ACE_START_WRITE(column);

    factor = sqrt(column[0]);
    for (i = 0; i < column_length; i++)
        column[i] = column[i] / factor;

    /* Terminate write and unmap column */
    ACE_END_WRITE(column);
    ACE_UNMAP(column);
}
```

Figure 4: An example of the use of the *Ace* runtime annotations.

The semantics of these primitives are, however, different from those in a single protocol system such as CRL. In *Ace*, invocation of these primitives on a region first determines the space of the region, and then dispatches the request to the appropriate protocol. The space of a region is determined by looking up the id of the region in a hash table. A space is implemented as a structure that holds pointers to the appropriate protocol's routines. The protocol pointers in a space are set (reset) whenever `Ace_NewSpace` (`Ace_ChangeProtocol`) is called. The structure also contains a pointer by which protocols may associate data with a space (for example, a static update protocol may wish to associate the sharer list for a particular data structure with its space).

4.2 *Ace* Compiler

We have implemented a compiler for *Ace* using the SUIF compiler toolkit [33]. The first stage in the compiler is to read a system configuration file that contains information

about the available protocols. The next step, which is relatively straightforward, is to translate an *Ace* program into the target runtime system format. Code for *Ace* annotations is inserted around accesses to shared variables in the intermediate representation of the program. Figure 5 delineates the process for loads (stores are handled in a similar fashion), and gives an example of code generated from a *Ace* code fragment.

As can be seen from the example, considerable overhead can be added for each access to shared memory. We have implemented three optimizations to reduce the overhead of calling *Ace* protocol routines. Note that the semantics of certain protocols, for example sequentially consistent protocols, do not allow code motion or other such optimizations [20]. We allow protocol writers to specify, when registering a protocol, whether a protocol's semantics allow optimizations. This flag is used by the compiler to determine whether an access can be optimized. A protocol is not optimizable if accesses in the protocol must appear atomic with respect to other accesses, or if the order of accesses between synchronization points is important.

Since not all protocols allow optimizations, before any optimizations can be performed on a program, it is necessary to determine, for each access, the set of spaces that are possibly associated with the data being accessed, and the set of possible protocols of each space at that access. To determine the set of spaces for each access, we perform global, interprocedural dataflow analysis. Information is generated at `Ace_GMalloc` calls and propagated to accesses. Concurrently, we propagate information about the protocols associated with spaces from `Ace_NewSpace` and `Ace_ChangeProtocol` calls. We compose the set of possible spaces for an access and the set of possible protocols for a space to derive the set of possible protocols for each access.

We now describe the three optimization phases in the *Ace* compiler. In all optimizations, code is never moved past synchronization calls.

Moving Calls out of Loops: Once the set of protocols associated with each access is determined, we perform loop invariance analysis (intra-procedural) on the arguments of calls to protocol routines to identify the calls that can be moved out of loops. `ACE_MAP` and `ACE_START_*` calls are moved above a loop, while `ACE_END_*` calls are moved below a loop. This optimization is performed only if all the possible protocols of an access are optimizable.

Merging Redundant Protocol Calls: We perform available expression analysis on each basic block on the arguments of `ACE_MAP` calls. Consider two `ACE_MAP` calls, M_1 and M_2. If the argument of M_1 is the same as that of M_2 and is available at M_2, then we remove M_2 and reuse the result of M_1. Furthermore, if the protocol actions associated with the two `ACE_MAP`'s are both reads or both writes, we use the highest `ACE_START_*`, and the lowest `ACE_END_*`, and remove the rest.[1] Figure 6 provides an example.

Avoiding Dispatching Overhead: If the compiler can determine that there is a unique protocol associated with an access, it replaces calls to *Ace* protocol dispatch routines (for example, `ACE_START_READ`) with direct calls to the appropriate protocol routine. The compiler obtains information about the name of the protocol routines from the system configuration file. In addition, if a protocol defines certain actions to be null, then calls to that protocol action can be removed.

[1] A possible optimization is to allow protocol designers to specify whether a protocol's semantics allow reads and writes to be merged.

Table 3: Benchmarks used in experiments.

Name	Inputs
Barnes-Hut [2]	16,384 bodies, 4 time-steps, tolerance = 1.0, eps = 0.5).
Sparse Cholesky(BSC) [26]	Tk15.O
EM3D [9]	1000 `E` and 1000 `H` vertices, 20%remote edges, degree 10, 100 steps
Traveling Salesman(TSP) [13]	12 cities
Water [31]	512 Molecules, 3 steps

5 Experiments

In this section, we use five benchmarks — Blocked Sparse Cholesky (BSC) [26], Barnes-Hut [2], EM3D [9], Traveling Salesman problem (TSP) [13], and Water [31] — to validate the performance of our system. We first compare *Ace*'s performance to CRL, an efficient software DSM. CRL has been shown to have low overhead and have good performance compared to Alewife, an architecture that supports shared memory in hardware [14]. Table 3 summarizes the benchmarks used in the paper. The code for Barnes-Hut, BSC, and Water are derived from the SPLASH [31] versions. EM3D is derived from the Split-C [9] version.

We then show the benefits of using customizable protocols for each of our benchmarks. Falsafi *et al.* [10] 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 DSMs. In addition, to our knowledge, we are the first to have studied application-specific protocols for BSC, TSP, and Water.

Lastly, we compare the performance of the *Ace* compiler-generated code with hand-written *Ace* runtime system code, and analyze the effects of the compiler optimizations. All experiments were run on a 32-processor Thinking Machines CM-5 (CMOST 7.3, Final 1 Rev 3) [17]. A processing node in a CM-5 is equipped with a 33Mhz SPARC microprocessor, a 64KB direct-mapped unified cache, and up to 128MB of memory.[2] Each application was compiled with `gcc -O2`.

5.1 *Ace* vs CRL

To perform a fair comparison of the *Ace* and CRL runtime systems, we use the *same* source files for *Ace* and CRL, Barnes-Hut, TSP, and Water were taken from the CRL 1.0 distribution and ported to *Ace* by replacing CRL primitives with the corresponding *Ace* calls. EM3D and BSC were written using the *Ace* runtime system and converted to CRL using the reverse process. We also *do not* use customized protocols in this experiment, both systems run a sequentially consistent invalidation-based protocol.

[2] We do not use the vector units on the CM-5.

Translation process for loads:

1. Find base of address to be loaded. For example, for *A[i]*, it is *A*.
2. Generate a ACE_MAP call on the base address and store the result in a tempory variable *t1*.
3. Generate a ACE_START_READ call on the temporary *t1*.
4. Perform the appropriate load on *t1* into another temporary *t2*. For example, *t2* = *t1[i]*.
5. Generate a ACE_END_READ call on the temporary *t1*.
6. Store *t2* into the original destination of the instruction.

```
shared struct hello {
    shared int *world;
} *x;

suif_tmp9 = ACE_MAP(x);
ACE_START_READ(suif_tmp9);
suif_tmp8 = *(int **)&suif_tmp9->world;
ACE_END_READ(suif_tmp9);
suif_tmp7 = 4;
suif_tmp11 = ACE_MAP(suif_tmp8);
ACE_START_WRITE(suif_tmp11);
*suif_tmp11 = suif_tmp7;
ACE_END_WRITE(suif_tmp11);
```

Figure 5: Compiler-inserted annotations for `*(x->world) = 4`

```
/* *x = y; */
suif_tmp8 = y;
suif_tmp9 = ACE_MAP(x);
ACE_START_WRITE(suif_tmp9);
*suif_tmp9 = suif_tmp8;
ACE_END_WRITE(suif_tmp9);
:
/* *x = 4; (x is live) */
suif_tmp7 = 4;
suif_tmp11 = ACE_MAP(suif_tmp8);
ACE_START_WRITE(suif_tmp11);
*suif_tmp11 = suif_tmp7;
ACE_END_WRITE(suif_tmp11);
```

```
suif_tmp8 = y;
suif_tmp9 = ACE_MAP(x);
ACE_START_WRITE(suif_tmp9);
*suif_tmp9 = suif_tmp8;
:
suif_tmp7 = 4;


*suif_tmp9 = suif_tmp7;
ACE_END_WRITE(suif_tmp9);
```

Figure 6: Example of removing redundant protocol calls.

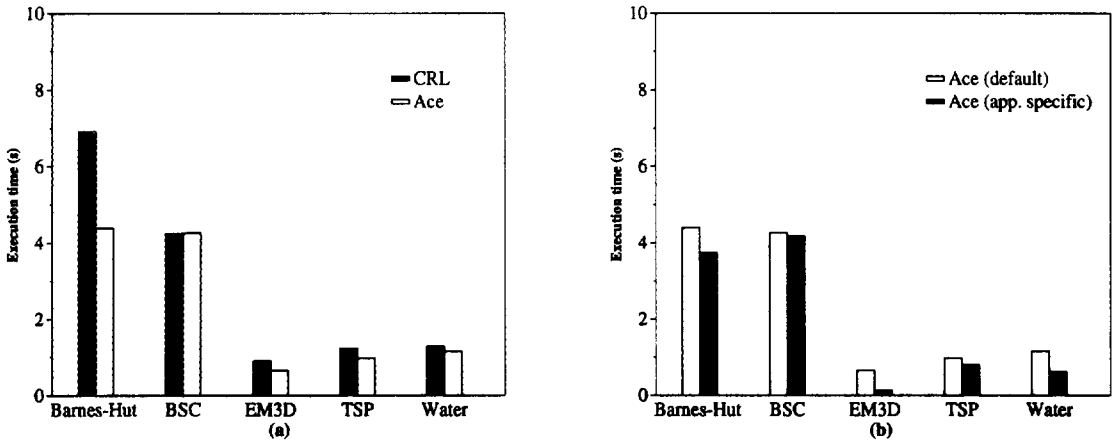

Figure 7: (a) *Ace* runtime system versus CRL (b) Comparison of using a single (sequentially consistent) protocol and application-specific protocols in *Ace*.

Figure 7a summarizes the results of our experiments. The times for Barnes-Hut, EM3D, and Water are average time per iteration. To avoid cold-start effects, we discard the results of the first iteration. Each execution time reported is the average of three runs. The reason for the performance difference between *Ace* and CRL is a careful redesign of the sequential consistency protocol and a more efficient mapping technique. The effects of the mapping optimization are most evident for fine-grained applications, such as Barnes-Hut and EM3D. For a coarse-grained application, such as BSC, the additional indirection in the dispatch of protocol calls in *Ace* nullifies the effects of the runtime system optimizations.

5.2 Application-Specific Protocols in *Ace*

In this experiment, we use versions of our benchmarks written directly for the *Ace* runtime system to study the benefits of using application-specific protocols with region-based DSMs (See Figure 7b). The speedups range from a factor of 1.02 to 5 (average speedup is approx. 2). Barnes-Hut uses a dynamic update protocol for bodies, and EM3D, a static update protocol, similar to those described in Falsafi *et al* [10]. In TSP, the improved performance is due to better management of accesses to a counter that is used to assign jobs to processors. In Water, we improve performance by pipelining writes to a molecule during the inter-molecular calculation phase, and using a null protocol during the intra-molecular calculation phase. For BSC, we take advantage of the fact that data are written only by the processors that created them. The performance improvement is marginal because, in BSC, the most important optimization is the use of bulk transfer for the transport of blocks between processors. Since the *Ace* runtime system supports user-specified granularity, the default protocol uses bulk transfer automatically. Optimizations to other accesses do not affect execution time greatly. Each of these protocols took from a couple of hours to a day to implement.

5.3 Evaluating the *Ace* compiler

We compare the execution times of compiler-generated code and code written directly for the runtime system for our benchmarks. For each application, we use the protocols that have the best performance. For the runtime system, we use the same code used in the previous two experiments, that is, code that an experienced programmer would write. In Table 4, for each application, we show the execution time of unoptimized compiler-generated code and the effect of each optimization. We also provide the time of the best implementation written for the runtime system.

As can be seen from the table, the best compiler versions are 1.1-1.3 times slower than the runtime system versions. Typically, the largest gains result from merging calls together. In Block Sparse Cholesky, however, there is a large improvement of the compiled code that can be attributed to the loop invariance optimization. This is largely due to the presence of several heavily used loops that calculate various matrix products. In EM3D, the static update protocol sets most of its handlers to be the null handler. During the direct dispatch optimization, the compiler removes dispatches to these handlers (most of which occur in a tight kernel) which results in a significant performance improvement.

One important parameter in the slowdown in the compiled code can be attributed to a greater number of `ACE_MAP` calls. For EM3D, the runtime system version performs `ACE_MAP` calls on each processor's data before entering the main computation loop, and uses the results inside the loop. The compiler was unable to determine that this optimization is possible. Further experiments revealed that the major component of the slowdown was a result of the extra `ACE_MAP` calls within the computation loop. On systems with a `mmap` call (the CM-5 does not support it), one could eliminate the use of `ACE_MAP` calls by ensuring that copies of data have the same address on every processor. This optimization would reduce the discrepancy between the compiler and runtime system performance significantly.

The current compiler is overly conservative in performing optimizations. For example, while alias information is available, the merging calls phase does not use this information while deciding when two calls can be merged. In addition, non-concurrency analysis [19] 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 could allow for more aggressive merging of protocol calls.

Table 4: Effects of compiler optimizations on benchmarks. All times in seconds.

Optimization	Barnes-Hut	BSC	EM3D	TSP	WATER
Base case	6.12	20.39	0.29	1.34	1.78
Loop Invariance (LI)	6.03	5.60	0.26	1.16	1.76
LI + Merging Calls (MC)	4.75	4.61	0.25	1.05	0.73
LI + MC + Direct Calls	4.60	4.50	0.17	1.05	0.71
Hand-optimized	3.74	4.18	0.13	0.80	0.63

6 Discussion

A goal in the design of *Ace* was to allow protocol designers the ability to take advantage of emerging architectures. For example, on Typhoon, which provides hardware support for access-fault control, protocol designers could implement certain protocols by registering null handlers with the *Ace* system and appropriate system handlers with Typhoon at protocol initialization time. Accesses to data structures associated with these protocols will be handled by the hardware mechanism automatically. Depending on the characteristics of the protocol, designers could choose to use the hardware or software mechanism. Similarly, on FLASH[15], one can use this mechanism to choose between custom and system protocols. Separating application and protocol views permits the use of hardware mechanisms by protocols, independent of application code.

In the past, the use of customizable protocols has mainly been studied in a pragmatic sense. However, a theoretical evaluation of the ramifications of mixing several consistency models is needed. For example, it is possible to have two spaces, where accesses to each of the spaces are sequentially consistent, but the combined set of accesses is not sequentially consistent [11]. While in our experience, the protocol to associate with a data structure is intuitive, a theoretical framework of correctness would be useful.

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, often several assertions can 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. Protocol development would also be facilitated by the creation of a library of protocol building blocks (for example, a routine for invalidating a cache block). We are currently attempting to isolate the primitives needed for such a library.

7 Conclusions

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 that of region-based systems, such as CRL. We have developed optimizations to improve the performance of the compiler, and found that *Ace* performs well compared to handwritten, region-based code. The performance of *Ace* applications is enhanced significantly by the use of customized protocols.

Acknowledgments

We wish to thank the Wisconsin Wind Tunnel(WWT) project, led by M. Hill, J. Larus, and D. Wood, for providing access to the CM-5 at the University of Wisconsin-Madison. We wish to thank the National Center for Supercomputing Applications at the University of Illinois (Urbana-Champaign) for the use of their CM-5. We would like to thank A. Appel, C. Dunworth, J. Reppy, and R. Samanta for their helpful comments on earlier drafts of the paper. Mukund Raghavachari was supported by a Princeton University Research Board Fellowship.

References

[1] A. Agarwal et al. The MIT Alewife machine. In *Proceedings of Workshop on Scalable Shared Memory Multiprocessors*, 1991.

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

[3] 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, 1990.

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

[5] 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.

[6] 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.

[7] 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*, 1994.

[8] 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.

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

[10] 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.

[11] 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.

[12] L. Iftode, J. P. Singh, and K. Li. Understanding application performance on shared virtual memory systems. In *Proceedings of the International Symposium on Computer Architecture*, pages 122–133, 1996.

[13] 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.

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

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

[16] 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, 1994.

[17] 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.

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

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

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

[21] 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*, 1995.

[22] 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.

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

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

[25] 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*, Apr. 1994.

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

[27] H. S. Sandhu, B. Gamsa, and S. Zhou. The Shared Regions approach to software cache coherence on multiprocessors. In *Symposium on Principles and Practice of Parallel Programming*, pages 229–238, May 1992.

[28] 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.

[29] 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*, Nov. 1994.

[30] 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.

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

[32] 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.

[33] 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.