Multi-Level Shared State and Application Specific Coherence Models

by

DeQing Chen

Submitted in Partial Fulfillment

of the

Requirements for the Degree

Doctor of Philosophy

Supervised by

Sandhya Dwarkadas

Department of Computer Science
The College
Arts and Sciences

University of Rochester
Rochester, New York

2004
Curriculum Vitae

DeQing Chen was born in Xi'an, China on July 17th, 1969. He attended Shanghai JiaoTong University from 1987 to 1991, and graduated with a Bachelor of Engineering degree in 1991. He continued his education in SJTU from 1991 to 1994, and earned a Master of Engineering degree in 1994. He came to the University of Rochester in the Fall of 1998 and began graduate studies in Computer Science. He pursued his research in distributed systems under the direction of Professor Sandhya Dwarkadas and received the Master of Science degree from the University of Rochester in 2000.
Acknowledgments

I would like to thank my advisor Sandhya Dwarkadas, and committee members Michael Scott, Adam Frank, and Amy Murphy for their patient guidance and insightful advice. Sandhya Dwarkadas and Michael Scott have supervised me through the entire InterWeave project. I appreciate all of their thoughtful direction and encouragement.

InterWeave is the fruit of strong teamwork and collaboration by a number of people. Chunqiang Tang has put enormous effort and contributed to making InterWeave efficient in a heterogeneous environment. Eduardo Pinheiro wrote an early version of the InterWeave IDL compiler. This dissertation work benefited greatly from Rob Stets’ work on the Cashmere system and Srinivasan Parthasarathy’s work on the InterAct system. Brandon Sanders helped us use InterWeave to facilitate coordinating shared state in the field of visual recognition. Xiangchuan Chen, Xiaochen Du, Gautam Altekar, and Andrew Ross have helped design and implement various useful InterWeave applications and tools. Rajeev Garg and William Scherer III helped proofread an earlier draft. I owe many thanks to insightful discussions with Shenghuo Zhu, Grigoris Magklis, Rajeev Balasubramonian, Umit Rencuzogullari, William Scherer III, and Steven G. Dropsho.

I have found the Computer Science Department at University of Rochester to be a great place to study. I would like to thank all the faculty members, department staff, and fellow graduate students for their support and friendship, which have made life here a wonderful experience.
I would like to thank my parents, HuXun Chen and Mei Jiao, my parents-in-law, ShiXie Wang and AiXian Li, and my sister Fang Chen for their unwavering support and encouragement.

Finally, I am most grateful to my wife YuanYuan Wang. This dissertation would not be possible without your support through all these years of hard work. I love you.

This material is based upon work supported by grants from the National Science foundation under grant numbers CCR-9988361, ECS-0225413, CCR-9705594, EIA-0080124; the grant from Department of Defense under grant number DEF-925F19460; and equipment or financial grants from Compaq, IBM, Intel, and Sun. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors(s) and do not necessarily reflect the views of the above institutions or the University of Rochester.
Abstract

As a result of the rapid growth of the Internet, more and more applications are taking advantage of resources at distributed sites. Conceptually, coordination of these distributed processes requires some sort of distributed shared state—information that has relatively static structure but mutable content, and that is needed at more than one site. The management of the shared state over the Internet must efficiently handle low-bandwidth network links, machine heterogeneity, and different application languages.

In this dissertation, we study a middleware system that supports a shared memory programming model designed to facilitate the sharing of state in such applications. As a complement to Remote Procedure Call or Remote Method Invocation (RPC/RMI), the system, which we call InterWeave, allows processes to access shared state of strongly-typed, pointer-rich data structures with ordinary writes and reads.

InterWeave employs a variety of aggressive optimizations to improve the performance of distributed state in the Internet environment. In this dissertation, we focus on InterWeave’s support for high-level coherence specification, its multi-level coherence protocol, and its support for Java applications. The high-level coherence specification allow programmers to relax application coherence requirements in order to optimize communication traffic. The multi-level shared memory protocol helps accommodate and leverage hardware coherence and consistency within multiprocessors (level-1 sharing), software distributed shared memory (S-DSM)
within tightly coupled clusters (level-2 sharing), and version-based coherence and consistency across the Internet (level-3 sharing). Support for the Java language presents special implementation challenges but allows Java applications to use InterWeave to share information with other languages efficiently.

Experiments on microbenchmarks show that the current InterWeave implementation is efficient in supporting a range of different machine types and programming languages. We have experimented with InterWeave on a variety of applications, including interactive and incremental datamining, time-image sharing in an intelligent environment, web proxy caching, multi-player games, and remote scientific simulation and visualization. Our experience demonstrates the ease of use of the InterWeave programming model. Experiments with these applications reveal that various InterWeave optimizations bring significant performance benefits in low bandwidth high latency network environments like the Internet.
Table of Contents

Curriculum Vitae ii

Acknowledgments iii

Abstract v

List of Tables x

List of Figures xi

1 Introduction 1

1.1 Multi-Level Shared Memory Structure 4

1.2 High-Level Application-Specific Coherence Information 7

1.3 Java Support 11

1.4 Summary 13

1.5 Dissertation Outline 13

2 Background and Related Work 14

2.1 Coherence and Consistency 14

2.2 Software Distributed Shared Memory Systems 19

2.3 Persistent Distributed Shared State 27
2.4 Application-Level Coherence Specification ............... 31
2.5 Replication and Caching Management over the Internet .... 33
2.6 Java-based Distributed Systems ......................... 38
2.7 Remote Procedure Call Systems ........................... 39
2.8 Summary ............................................. 40

3 Efficient Distributed Shared State ........................... 42
   3.1 InterWeave Application Programming Interface .......... 43
   3.2 Framework of Implementation .......................... 48
   3.3 Evaluation ......................................... 58
   3.4 Summary ............................................. 67

4 High-Level Coherence Information ........................... 68
   4.1 Relaxed Coherence Models ............................. 69
   4.2 Dynamic Views .................................... 72
   4.3 High-Level Coherence Information Usage and Customization .. 74
   4.4 Implementation .................................... 79
   4.5 Micro-benchmark Evaluation .......................... 85
   4.6 Application Evaluation ............................. 94
   4.7 Summary ............................................. 108

5 Three-Level Shared Memory Structure ....................... 110
   5.1 InterWeave Programming on the Cashmere Cluster .......... 111
   5.2 3-level System Implementation on the Cashmere Cluster ... 114
   5.3 Evaluation ......................................... 120
   5.4 Summary ............................................. 124
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Java Support for InterWeave Applications</td>
<td>126</td>
</tr>
<tr>
<td></td>
<td>6.1 InterWeave Java Programming Interface</td>
<td>127</td>
</tr>
<tr>
<td></td>
<td>6.2 Implementation of Java Support</td>
<td>131</td>
</tr>
<tr>
<td></td>
<td>6.3 Evaluation</td>
<td>138</td>
</tr>
<tr>
<td></td>
<td>6.4 Summary</td>
<td>144</td>
</tr>
<tr>
<td>7</td>
<td>Conclusions</td>
<td>146</td>
</tr>
<tr>
<td></td>
<td>7.1 Contributions</td>
<td>146</td>
</tr>
<tr>
<td></td>
<td>7.2 Future Work</td>
<td>149</td>
</tr>
<tr>
<td></td>
<td>7.3 Recent Publications</td>
<td>150</td>
</tr>
</tbody>
</table>

Bibliography 151
List of Tables

3.1 InterWeave Implementation Decomposition. ..................... 58
3.2 Basic Operation Costs ........................................... 59
3.3 Basic InterWeave Translation Costs ........................... 62
3.4 Basic InterWeave Bandwidth Costs ............................ 63

4.1 InterWeave Relaxed Coherence Models ......................... 70
4.2 Simulation Results for ICP. ....................................... 102

6.1 Astroflow Java Visualization Client Performance ............. 144
# List of Figures

1.1 InterWeave's Multi-Level Structure ........................................... 6  
1.2 Sharing in an Intelligent Environment ..................................... 10  
2.1 Memory Access Categorization in Release Consistency. ............. 16  
3.1 InterWeave Client Data Structures ......................................... 53  
3.2 InterWeave Synchronization Costs on a Gigabit Ethernet .......... 61  
3.3 InterWeave Synchronization Costs on a 100Mbps Ethernet .......... 61  
3.4 Binary tree Update and Search Latency .................................. 64  
3.5 Binary tree Communication Traffic ......................................... 65  
4.1 Diff Collection with a View .................................................. 84  
4.2 Overhead for Acquiring Writer Locks ..................................... 86  
4.3 Overhead for Acquiring Reader Locks ..................................... 87  
4.4 Comparison of Full and Diff Coherence Models. ....................... 88  
4.5 Latency Reduction Using Non-Recursive Views ......................... 90  
4.6 Bandwidth Reduction Using Non-Recursive Views ...................... 91  
4.7 Bandwidth Reduction Using Recursive Views ............................. 92  
4.8 Latency Reduction Using Recursive Views ............................... 93
4.9 Bandwidth Cost in Datamining .................................. 96
4.10 Bandwidth Cost under Relaxed Coherence Models .......... 97
4.11 Using Recursive Views in Datamining ......................... 98
4.12 View Update Latency in Datamining ........................... 99
4.13 Server-side Diffing Cost in Datamining ...................... 99
4.14 ICP Directories .............................................. 103
4.15 ICP Directory Cost in InterWeave ............................ 104
4.16 Image Cubes and Views ...................................... 106
4.17 Bandwidth Consumption for Image Cubes and Views .......... 108

5.1 3-Level Synchronization on Top of Cashmere ................. 118
5.2 Barnes-Hut Overhead on a 3-Level System .................... 121
5.3 Astroflow Overhead on a 3-Level System ..................... 122
5.4 InterWeave Communication Overhead for Astroflow ........... 123

6.1 IWSegment Class. ............................................. 128
6.2 Block Structure in InterWeave Java Implementation .......... 133
6.3 Extending Kaffe Object Headers .............................. 135
6.4 Write Detection Overhead in the Kaffe Interpreter .......... 139
6.5 Write Detection Overhead in the Kaffe JIT3 Engine ........... 139
6.6 Translation Cost under JDK and InterWeave ................. 141
6.7 Translation Latency of a Partially Modified Java Double Array .. 142
6.8 Translation Bandwidth of a Partially Modified Java Double Array 143
1 Introduction

The rapid growth of the Internet has spurred new interest in moving applications to the wide-area networking environment. With multiple processes at multiple sites, these applications can take advantage of distributed resources to create or improve services, and provide increased scalability and availability. Example application domains include e-commerce, computer-supported collaborative work, multi-player games, intelligent environments, interactive data mining, and remote visualization and steering of real or simulated systems.

Conceptually, coordination of most of these applications in the distributed environment requires some sort of distributed shared state: information that has relatively static structure but mutable content, and that is needed at more than one site. There are two ways to access remote state—either by moving the access to where the data currently reside or by moving the data to the local storage. The first option corresponds to remote procedure call (RPC) or remote method invocation (RMI). It has become popular in Internet applications with the support of middleware systems such as CORBA [Obj96], .NET [Cor02], RPC [BN84], and Java RMI [Sun98].

RPC programmers reason about distributed coordination in terms of making procedure calls. Parameters and return values are automatically marshalled by
the underlying functions. Heterogeneity is accommodated with serialization of typed data items. An access to remote state usually involves making a procedure call that sends a message to the remote site; the callee returns the result by replying to the message. When making calls, the RPC runtime system marshals and unmarshals parameters and results between the local memory format and a wire format to allow communication between heterogeneous machines. Messages help distributed processes synchronize among themselves. With application-level knowledge, an experienced programmer can design arbitrarily complicated protocols for making procedure calls to minimize both synchronization and communication cost, and thus be very efficient.

Using RPC/RMI in the Internet is not without tradeoffs. Unlike local area networks, Internet and wide-area networks usually feature high latency and low bandwidth links. To allow applications to communicate efficiently, programmers typically attempt to increase locality through ad-hoc, application specific caching or replication protocols [CDFV00, KA96]. However, this usually adds additional complexity to the design of the application and requires considerable programming effort.

In addition, there is no trivial solution for handling pointer-rich shared data structures in RPC/RMI [KKM94]. Some RPC systems (such as Sun RPC) recursively copy the entire pointed-to data structure (deep copy), while some others (such as Java RMI and CORBA) pass remote references and require the callee to call back. Neither of these is ideal: deep copy requires significant bandwidth when the pointed-to data are large; passing remote references increases round trip communication overhead through another level of indirect accesses.

The second option we mentioned above—moving data to local storage—corresponds to a shared memory programming model. A shared memory system hides the underlying messaging mechanisms from the programmers by automatically caching remote state locally to allow access with normal reads and writes. Hardware
shared memory and software distributed shared memory (S-DSM) systems have seen wide-spread research and acceptance in the high performance and parallel computing domains, though they have rarely been extended and applied in wide-area networking environments. As a general rule, these systems assume that sharing processes are part of a single parallel program, running on homogeneous hardware, with communication latencies and bandwidth typical of modern local- or system-area networks, and with data lifetimes limited to that of the running program.

In this dissertation, we study a middleware system that tries to extend the shared memory programming paradigm to the wide-area Internet environment. This system, which we call InterWeave, deals specifically with the following research challenges: 1) interoperability of shared memory systems available at different hardware and software levels, 2) coherence management in low-bandwidth wide-area networks, 3) the sharing between popular Internet-wide Java applications and applications in other languages, such as C/C++, and 4) machine heterogeneity. This dissertation covers the first three challenges. Our thesis can be summarized as follows:

A shared state system across the Internet is feasible and can reap significant bandwidth benefits with high level application specific coherence management, while preserving high performance on tightly coupled clusters within a multi-level structure.

In the rest of this Chapter, we summarize the problems targeted by our thesis. After that, we give an outline of this dissertation.
1.1 Multi-Level Shared Memory Structure

Shared memory systems and cache coherence mechanisms can be built at multiple levels using hardware or software. At the hardware level, multiple microprocessors can share access to a single or distributed body of physical memory with hardware controlled coherence and consistency protocols. Specifically, symmetric multiprocessor machines (SMPs) that can be built to a modest scale have recently become popular. On the other hand, similar coherence mechanisms can also be implemented purely in software. Software distributed shared memory (S-DSM) systems such as TreadMarks [ACD+96] and Cashmere [SDH+97] use software methods to provide a global address space on clusters of workstations that are connected by locally available high bandwidth low latency system area networks.

Cashmere [SDH+97] is designed to exploit a two-level shared memory architecture on a cluster of SMP machines. The Cashmere system uses virtual memory mechanisms to provide a single memory address image across a cluster. However, processes inside each single SMP node use the hardware shared address space to map cluster-wide shared memory space. Cashmere could run on a variety of systems. Our experimental evaluations [Ste99] have shown that overall S-SDSM performance can benefit from leveraging the hardware provided coherence mechanism within the two-level architecture.

The Cashmere two-level architecture is not readily extensible to a wide area network environment: machines inside the cluster are homogeneous, and they are located within a geographically close area within which high speed local area network connections are available. In the Internet environment, machines have different hardware architectures and software platforms, they could spread over the entire world, and the communication links tend to be much higher in latency and much narrower in bandwidth.

Still, it is conceptually appealing and practically beneficial to extend the two-
level shared memory architecture to span the Internet. Many applications have the need to allow remote Internet processes to interface with high performance computing platform such as clusters. Examples include the online visualization of scientific simulations and object identification in an intelligent environment. In the first example, an S-DSM cluster running a simulator program needs to share the computation result with its remote visualization clients. In the second example, a group of strategically deployed cameras share images and leverage the computation power of a cluster to identify moving objects.

We have designed a multi-level structure to provide a systematic solution to help programmers in high performance distributed computing. Cashmere's two-level shared memory structure is extended to the Internet level, called the third level in InterWeave, to allow geographically dispersed and architecturally heterogeneous machines to share a common address space. InterWeave tries to map the level-3 shared data structure that allows heterogeneity to a level-2 shared memory region whenever possible so that it can leverage the existing level-2 consistency management to minimize overhead within the cluster.

An example of this multi-level environment is shown in Figure 1.1. A scientific simulator application called Astroflow [FDBH] (written in Fortran), runs on a Cashmere cluster with a two-level structure. The simulator is a computational fluid dynamics system for studying the evolution of stellar bodies. Connected to the cluster via the Internet are one or more satellite machines, located on the desktops of researchers physically distant from the cluster. These satellites run visualization and steering software written in Java and running on top of a JVM. The operating system and hardware architecture of these satellites are altogether different from those of the Cashmere cluster.

The distributed shared state in the above application is the data structure describing the simulated stellar bodies. On the Cashmere cluster, these data are distributed via Cashmere's global address space for high compute performance.
Figure 1.1: InterWeave's target environment. In the top half of the figure, we have a Cashmere cluster with 8 SMP nodes connected by a high performance point-to-point network. In the bottom half of the figure, remote satellites share simulation results with the cluster through the Internet.
To allow the Java visualization satellites to access the simulation state, state-of-the-art techniques would require the programmer to use a manually designed messaging protocol on top of something akin to CORBA or RPC to pass the data between the cluster and satellites. To reduce update bandwidth over the Internet, the application would need to employ caching while taking care of data consistency at both the cluster and the satellites.

On the other hand, within the InterWeave 3-level structure, the sharing among the simulator and the visualization clients can be handled seamlessly and efficiently by putting the shared state into the level-3 shared memory space. The InterWeave system automatically takes care of the coherence and consistency management and translation of the shared state, with negligible performance penalty on the simulator. With minimal programming effort, the programmer can optimize the communication across the Internet by specifying application level coherence information.

1.2 High-Level Application-Specific Coherence Information

In a shared memory system, processes read or write shared state located remotely in the same manner as when using local memory. To compensate for the overhead of remote operations, a shared memory system usually caches recently used state in local memory. If one sharer changes its cache, the modifications must be propagated to the other sharers in a coherent and consistent manner. Maintaining the coherence and consistency of shared data is essential for the correctness and efficiency of a shared memory system. It has been the subject of intensive research in the past decade and a half.

A Software Distributed Shared Memory (S-DSM) system aggressively utilizes
relaxed consistency models to reduce the cost of maintaining cache coherence and updates are delayed until the next use of the invalidated data. These “relaxed consistency” models, however, still require updates of the most recent copies of the accessed coherence units. For example, in the Cashmere protocol, the acquiring node has to fetch the most recent version of the entire required page to ensure the correctness of the protocol.

With the high latency and low bandwidth wide-area networking of the Internet environment, it is important to seek more aggressive methods to reduce the coherence overhead in shared memory applications. Just as an RPC programmer can use application-level knowledge to optimize messaging protocols, a shared memory programmer should be able to use application-level knowledge to optimize coherence updates. The goal, however, is to create an interface that is easy to use while allowing the runtime to deal with the complexities of heterogeneous communication and optimization.

In many applications we find that coherence requirements can be relaxed along two dimensions: the temporal dimension and the spatial dimension. In the temporal dimension, some applications can delay the update of cached state as long as the current cached version is “recent enough”, even though some more recent version of the state has been created. In the spatial dimension, applications that have knowledge of spatial access patterns can delay updates of currently unused portions of the data structure.

For example, in the Astroflow application, a temporal coherence relaxation will allow a remote satellite to avoid unnecessary coherence update requests. If a visualization satellite only needs to refresh its screen every $x$ seconds, it can actually relax its coherence requirement to retain its potentially outdated cache as long as the cache was updated within the last $x$ seconds\(^1\).

\(^1\)An analogy can be made here to the “max-age” cache-control directive in the HTTP/1.1 header protocol. In this example, a browser or a proxy cache can use a cached web object as
Spatial coherence requirements can be illustrated with an intelligent environment application, currently under development in our department [CTS+03, SNS02a, SNS02b]. In this application, cameras mounted throughout a living space monitor a common area from several vantage points. The cameras work cooperatively to discover objects by detecting events that are simultaneously observed by multiple cameras in the recent history. Figure 1.2 gives an example of discovering objects with three cameras. Each camera is served by a computing node, and the nodes are connected by a local-area network. Each node stores captured images locally as an image cube \((X \times Y \times t)\) and scans them for events of interest. When interpreting an event, a given node enhances its understanding of what occurred by scanning images captured by other nodes at the same time. Suppose that a yellow bowl moves into the monitored area. The monitoring system [SNS02a, SNS02b] wants to identify this object by comparing images at the same location, sampled at successive times.

Since events of interest occur in subregions of images, it suffices to share only the “interesting” image areas at any given time. While an entire image may logically be shared, the actual sharing varies dynamically according to the activity in the environment being observed. A coherence protocol that requires updates of entire images would be very inefficient and incur unnecessary communication traffic.

The general problem of the mismatch between coherence unit and the actual application sharing pattern is called “false sharing”. Page-based S-DSM systems ameliorate the problem but do not eliminate it. In the previous example, the application stores each image in a row major order where a row of pixels are stored together. An examination of one or multiple columns of pixels will force the system to retrieve the update for the pages containing these columns. Most of the updates are not used, and the corresponding bandwidth is wasted. The programmer could long as it is less than “max-age” old.
Figure 1.2: Three cameras monitor the same environment and cooperatively discover four objects by sharing relevant portions of their image cubes.
rearrange the image format to allow efficient sharing of column pixels. However, doing this incurs computation overhead and would disrupt sharing patterns in other image operations.

We try to solve the problem with a flexible method to allow a distributed application to specify high-level temporal and spatial coherence information. The method is independent of the underlying system implementation and should work in a heterogeneous environment. In InterWeave, specifically, we design relaxed coherence models for temporal coherence information and dynamic views for spatial coherence information. The temporal coherence information specifies when a cached shared state is still considered recent enough to be coherent by the application. It allows the runtime system to exploit the existing local cache to eliminate update overhead. On the other hand, a programmer uses spatial coherence information to specify the portion of the shared data structure that the application is going to use and need updated. Correspondingly, the runtime system can exploit the opportunity to optimize the coherence communication overhead by ignoring updates outside of the specified portion.

1.3 Java Support

In the Astroflow application we have described above, the satellites are written in the Java language. To exchange information with the Fortran simulator on the cluster, traditionally one would have to either use a standard RPC/RMI system as a common factor, or write a Fortran/Java interface at the satellite or simulator site. With the growing popularity of Java across the Internet and the in-diminishing popularity of Fortran and C/C++ in high-performance computing, we expect that the need for sharing among Java and legacy languages will only grow stronger.

Even for systems written entirely in Java, it will be helpful to be able to share
objects across heterogeneous JVMs. This is possible, of course, by using RMI or CORBA to access the remote site and using object serialization to pass shared objects. RMI has no built-in caching schemes to exploit localities in the programs. However, a number of Java middleware systems [AFT+00, PZ97, MKB00] have been developed to help automate the replication of RMI objects and allow object mobility across distributed JVMs. Our work is very different from these. The Java support in InterWeave provides an automatic method for applications to utilize caching among applications written in different languages. While it is possible to allow thread mobility on top of the InterWeave global address space, we do not explicitly address this problem in this dissertation.

Furthermore, RMI and many of the Java object replication schemes use Java object serialization to transmit shared objects. Object serialization deep copies Java references. Thus, when using object serialization to transmit a Java reference, the entire object cluster reachable from the reference has to be translated and sent to the remote sharer. To optimize such heavy weight object updates, we allow Java applications to obtain the benefit of the InterWeave global address space and efficient coherence management.

However, support for these coherence optimizations presents non-trivial challenges. For example, InterWeave's coherence protocol requires an efficient method to track modifications to locally cached Java objects. A pure Java implementation is possible by using byte-code rewriting techniques [CCK98]. However, this will bring unacceptable performance degradation and an extra recompilation pass of the Java applications and library classes. Our solution optimizes the tracking for InterWeave Java applications with a customized Java virtual machine. Inside the JVM, we reuse the InterWeave C implementation for efficient type matching and data translations.
1.4 Summary

InterWeave aims to provide programs with a global shared memory space that can be accessed with ordinary reads and writes, regardless of the machine architectures or programming languages used. As a complement to RPC, InterWeave serves to eliminate hand-written code that maintains the coherence and consistency of cached data. Our experience with real applications demonstrates that the InterWeave programming model is easy to use and can be applied in multiple languages. With a variety of optimization techniques integrated, an application can achieve high system efficiency that traditionally can be obtained only through aggressive programming effort. Our experiments show that these optimizations are especially useful over the Internet. We also find that the three-level structure effectively and efficiently allows applications to integrate a high-performance cluster and remote satellites connected with much lower bandwidth network links.

1.5 Dissertation Outline

We discuss background and related work in the next Chapter. Chapter 3 describes the application programming interface of the InterWeave system. We discuss coherence and consistency in Chapter 4. Chapter 5 discusses the design, implementation, and evaluation of InterWeave's three-level structure. Chapter 6 discusses issues related to the InterWeave Java implementation. We conclude and discuss future work in Chapter 7.
2 Background and Related Work

In this chapter, we provide necessary background and discuss related work. Section 2.1 describes some memory consistency models and their possible impact on system performance. Section 2.2 discusses related software distributed shared memory systems. Section 2.3 discusses support for persistent shared state storage in local area networks and across the Internet. Section 2.4 describes some systems that allow application-level coherence optimizations. Section 2.5 discusses replica consistency across the Internet. Section 2.6 discusses related work in sharing Java objects. Section 2.7 discusses related work in remote procedure call systems.

2.1 Coherence and Consistency

Reasoning about program behavior in a distributed and parallel environment would be difficult without a memory consistency model. Memory accesses on a sequential machine are intuitive and easily defined: a LOAD or read operation will return the value of the last STORE or write operation in program order. In a distributed system, the result of a LOAD operation is not immediately clear as there might be multiple competing STORE operations happening simultaneously. A memory consistency model thus provides semantic guarantees to allow a behavior as close as that on a sequential machine.
Sequential Consistency

The most straightforward and intuitive consistency model is Sequential Consistency, as proposed by Lamport [Lam78]. As its name suggests, in a sequential consistent system, memory accesses are executed as if they are running on a sequential machine:

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

However, sequential consistency is usually too strict to allow some common optimizations for improving system efficiency on many distributed memory systems. For example, with the high latency of remote memory accesses, the use of write buffers to accumulate remote memory writes can significantly reduce communication cost. Under sequential consistency, such optimizations become impossible because it will change the order of writes seen at different sites.

Relaxing the sequential requirement allows re-ordering of memory operations for various performance optimizations. On the other hand, non-sequential consistency models are usually harder for programmers to understand because they no longer behave exactly like a sequential machine. Programmers might see out-of-order executions of instructions. This will complicate application development and portability as programmers are forced to think of the details of lower-level architectural optimizations. In the rest of this section, we describe some relaxed consistency models that present applications with the illusion of the sequential consistency, while allowing lower-level re-ordering of memory operations.
Release Consistency

Release Consistency exploits application synchronization information to relax ordering constraints. Shared memory accesses are categorized in respect to their synchronization attributes, as shown in Figure 2.1. First they are divided into competing accesses and non-competing accesses. Two memory accesses are competing if they conflict with each other, i.e., they access the same memory location and at least one of them is a write. Competing accesses are further categorized into synchronization operations and non-synchronization operations. Non-competing accesses are provided to accommodate chaotic relaxation models. Synchronization operations are used for enforcing orderings and include acquires and releases. As their names imply, an acquire gains the access right to a set of shared data and a release relinquishes the previous acquired access right. They are usually paired and implemented with reader/writer locks.

Release Consistency (RC) [GLL+90] assumes all memory accesses in a parallel program are properly labeled (PL). Non-competing accesses are labeled as ordinary accesses and competing accesses are labeled as special accesses. Intuitively, a PL program has no data race condition (i.e. it is properly synchronized).

With the synchronization information, release consistency models thus re-
quire [GLL+90]:

“(A) before an ordinary LOAD or STORE access is allowed to perform with respect to any other processor, all previous acquire accesses must be performed, and (B) before a release access is allowed to perform with respect to any other processor, all previous ordinary LOAD and STORE accesses must be performed, and (C) special accesses are sequentially consistent with respect to one another.”

A release consistent system appears as a sequential consistent system as long as the program is properly synchronized. This would allow optimizations that reorder or even eliminate some ordinary memory operations.

Lazy Release Consistency

Lazy Release Consistency (LRC) [ACD+96] refines release consistency by considering the causal relationship [AH90] among special accesses. In RC, modifications are propagated at every release operation. LRC incurs no memory operations at release time. At acquire time, a process applies modifications that happens previously according to the happens-before relationship [Lam78].

Comparing with RC, LRC allows coherence messages to be aggregated at a processor’s acquire time. In addition, propagations that have no causal relationship with the acquiring processor are avoided. These optimizations are very important for software distributed shared memory systems (S-DSM) because their network latencies are considerably higher than those in hardware DSM systems.

Entry Consistency and Scope Consistency

Like RC, Entry Consistency (EC) [BZ91] also categorizes memory accesses with respect to the synchronization model. It goes further to explicitly associate each
shared object with a synchronization variable. Accesses to a set of shared objects must be guarded properly by their associated synchronization objects. An acquire of a synchronization variable begins a critical section and a release ends it. With these provisions, at the acquire time, the system only updates those shared objects that are associated with the synchronization variable.

EC improves system parallelism by allowing multiple critical sections guarded by different synchronization variables to execute concurrently without unnecessary message exchanges. Synchronization variables are refined into exclusive and non-exclusive types. Non-exclusive acquires can be granted simultaneously to multiple processors to allow potentially more parallelism.

Scope Consistency [ISL96] (ScC) inherits entry consistency’s idea of associating synchronization variables with shared data objects and applies it to a page-based S-DSM system. However, unlike entry consistency, ScC implicitly associates each critical section—a scope—with the synchronization operation that encloses the scope. A pair of lock acquire and release encloses a local scope; a barrier delimits a global scope. Unlike LRC, ScC does not consider the causalities among previous synchronizations. Thus, a correct program under LRC may become incorrect under ScC. Programmers have to either choose to redesign the application synchronizations or explicitly annotate scopes.

The idea of explicitly relating synchronization objects with shared data regions can be also found in Shared Region [SGZ93]. But it was used for software cache coherence management on multiprocessor shared memory machines.

Coherence

Coherence is a weakened form of sequential consistency [GLL+90, Mos93]. Although coherence and consistency are used interchangeably in some publications, they are different. Sequential consistency requires execution orderings be consistent on all processes to all memory accesses. Coherence, instead, only requires
consistent accesses to individual memory locations. It removes the ordering requirement for accesses to different memory locations. Thus, a coherent execution of memory accesses might be sequentially inconsistent when we consider inter-location relationships.

In InterWeave, the segment is the basic coherence unit. Accesses to individual segments are kept coherent under the InterWeave coherence models. On the other hand, consistency among multiple segments is enforced on an optional basis, using a causality-aware hash scheme. The solution strikes a balance between reducing consistency overhead and providing a reasonable programming model.

Other Consistency Models

Hybrid Consistency [Fri93] and Mixed Consistency [CGS96] are other efforts to categorize memory operations to weaken consistency and allow performance optimizations. Hybrid consistency separates memory operations into strong and weak operations. An example of a strong operation is an atomic read-and-modify operation, which can be used to implement synchronization operations. Mixed consistency considers synchronization operations and weak memory operations that satisfy Pipelined Random Access Memory Consistency [ABHN91] and Causal Memory Consistency [LS88].

2.2 Software Distributed Shared Memory Systems

Distributed shared memory can be implemented at both hardware and software layers. In comparison with hardware systems, S-DSM systems usually have much higher communication overhead for remote accesses. They also tend to have larger coherence units to make messaging optimizations possible. There have been many
research projects on efficient S-DSM implementations. A complete survey is not our intention here. Instead, we discuss some S-DSM systems that are representative in optimizing performance and most related to the InterWeave system.

2.2.1 Page-based One-level S-DSM

IVY

Page-based software distributed shared memory system was pioneered by Li’s work on the IVY system [LH89]. IVY implemented a shared virtual memory system on a collection of workstations connected by an Apollo ring. Each node keeps a page table to record access permissions for each shared page. The page access permission can be one of invalidate, read-only, and read-write. Each page has a designated owner node to coordinate access requests from sharers.

IVY implements a sequential consistency memory model. Virtual memory page faults are used to track accesses to shared pages. Initially, every page is invalidated. On a read fault, the node asks for read access permission from the owner node and obtains a current page replication. Then it upgrades the page permission to read-only. On a write fault, the node again asks the owner for write access permission. The owner sends back the page to the requester and a list of current sharers. The faulting node then sends explicit messages to the other sharers to invalidate the page in their page tables. After receiving confirmation from all sharers, the node upgrades its access permission to read-write and resumes as the new owner of the page.

IVY is a coarse-grained S-DSM system, i.e., its coherence unit (page) is usually much bigger than the application sharing unit. Coupled with the a sequential consistency model and a single-writer protocol, IVY can cause serious false sharing problems for parallel applications. For example, when two processors write simultaneously to two disjoint portions of a single page, IVY has to move the page
back-and-forth between the two sites. This "ping-pong" effect exacerbates communication and synchronization overhead on the application. Many later S-DSM systems subsequently try to optimize IVY's performance with relaxed consistency models, multiple writer protocols, fine-grained sharing techniques, and high performance networks.

**Munin**

Munin [CBZ91] tries to improve IVY's performance by using relaxed consistency models and enabling multiple writers. Release consistency based memory models allow distributed sharers to aggregate coherence messages until the next *Release* operation. To avoid false sharing at the page level, Munin assumes that applications follow the data-race-free-1 programming model [AH93] and allows multiple properly synchronized processes to write to one page simultaneously. On a write fault, the writer creates a pristine copy of the page, called a *twin*. At the next *Release* operation, a *diff* is computed and forwarded to the page owner. Thus modifications from different writers can be safely merged.

**TreadMarks**

TreadMarks [ACD+96] uses a Lazy Release Consistency model to further optimize consistency messages. TreadMarks divides process executions into consecutive intervals, delimited by corresponding synchronization operations. The causal relationships between intervals on different processors are maintained by keeping a completely distributed processor-specific time vector. Messages are only exchanged when strictly necessary at synchronization points. TreadMarks also uses twins and diffs to allow multiple writers, however, diffs are only generated when necessary. Because the coherence information is completely distributed, TreadMarks utilizes a garbage collection mechanism to recycle outdated diffs.
2.2.2 Fine-grained S-DSM

Blizzard

Another solution for the false-sharing problem is to allow fine-grain access control to shared memory regions. Blizzard [SFL+94] exploits three methods to implement fine-grain control in software. All these versions divide memory into 32-byte blocks. They all implement a single-writer, sequential consistency protocol. Blizzard-S instruments an application executable before every load/store instruction to detect possible accesses to individual blocks in the shared memory regions. Blizzard-E uses the ECC trap (error-code correction) to detect loads from invalid shared memory regions and uses page-based VM methods to detect stores. Blizzard-ES combines the above two methods, i.e., using executable instrumentation to detect stores and ECC to detect loads.

It is not clear how Blizzard compares to page-based S-DSM systems. In their paper [SFL+94], it was reported Blizzard-E actually outperformed Blizzard-ES, even though the former used a coarse-grained VM-based access control for writes, while the latter employed a finer-grained instrumentation-based access control. There are two possible reasons for that. First, a page-based system can actually batch writes and reduce invalidation overhead for applications with coarse-grained sharing patterns. Second, Blizzard instrumentation was rather preliminary, and had not been fully optimized.

Shasta

Shasta [SGT96] extends the Blizzard work on fine-grained access control in software shared memory. Like Blizzard, it is a single-writer protocol. Shared memory addresses are divided into lines of fixed sizes (64 or 128 bytes, larger than Blizzard blocks). A state table is employed to record shared state for each line. The coherence unit is a block, each of which can span multiple lines. Because blocks can be
of different sizes, Shasta supports different coherence granularity for applications with different sharing patterns.

Shasta exploits various techniques to optimize the software instrumentation cost. At the beginning of the instrumentation process, a static analysis procedure is used to find two unused registers to avoid saving and recovering registers. Instrumented code splices are scheduled to hide instruction latency on super-scalar machines. Shasta uses in-place flags to avoid checking state tables. When invalidating a line, Shasta stores a "flag" into the head of the line. By simply checking the flag, the instrumented code can quickly determine if the line is valid. Shasta makes further improvement by batching checks for read and write misses when the read/write addresses are indexed by using an unmodified base address register. This is especially useful when the compiler unrolls loops. With all these enhancements, Shasta shows much improved performance and scalability than Blizzard.

Millipage

Millipage [IS99] is a page-based DSM that attempts to simulate fine-grained control by a technique called MultiView. MultiView maps logical memory objects that are contiguous in physical memory into separate virtual memory regions. Each such object is assigned a virtual page frame, called millipage. A set of such millipage mappings form a view. MultiView thus allows individual applications to establish their own view of the shared data structures, and operates on sub-page coherent units.

Millipage, however, still uses page faults to detect writes and reads to memory objects. Having millipages could increase the opportunity that an application traps into the page fault handler and consequently increases coherence messages among sharers. To solve the problem, Millipage has to cluster small objects together to form larger millipages.
Millipage also stretches the underlying system by using a large portion of virtual address space. Subsequently it can increase the TLB miss rates and force in-memory page tables out of the second-level caches. All these contribute to serious performance degradation when the shared data size becomes large, which is typical for high-performance data-intensive parallel applications.

Millipage is not adequate for handling dynamic false sharing. Applications assign their view when they allocate and map the shared data objects. Once mapped, millipages are difficult to change because all sharers need to rearrange their virtual memory to physical memory mappings correspondingly.

2.2.3 Two-level S-DSM

Multigrain Shared Memory System

Multigrain Shared Memory System (MGS) [YKA96] was the first effort to build scalable shared memory systems from a cluster of SMP nodes. MGS hypothesizes that applications running on a two-level system exhibit multigrain locality—the sharing between processes on different SMP nodes uses a coarser grain than the sharing between processors inside individual SMP nodes. Unlike working set locality, Multigrain locality can only be exploited by systems that provide sharing at multiple granularities.

MGS is built on top of the Alewife distributed multiprocessor that supports shared memory in hardware. An Alewife machine consists of a number of processors connected in a 2-D mesh topology. A two-level system is simulated by grouping processors into smaller logical clusters (SMPs). Communication delay between SMPs is introduced by software to simulate longer inter-SMP communication overhead.

MGS’s software protocol at level-2 is based on the Munin multiple writer, invalidation based protocol. Memory mappings, however, follow a two-level de-
sign. For each page, besides the global sharing state maintained by the home node, each node holds a local page table. Thus, the protocol can avoid expensive inter-cluster communication as long as page update and invalidation can be accomplished locally.

The MGS experimentations show that many applications exhibit multigrain locality and benefit from the two-level system design. Since it was simulated on a multiprocessor system, it is hard to compare their results with other two level systems.

**Softflash**

SoftFLASH [ENCH96] implements the Stanford FLExible Architecture for Shared Memory (FLASH) [KOH+94] entirely in software. It is a two-level S-DSM system. In SoftFLASH, processes are clustered over SMP nodes so that intra-node sharers can leverage the underlying hardware coherence and consistency mechanism. SoftFLASH implements an optimized version of Release Consistency, called Delayed Invalidate Release Consistency (DIRC). Unlike RC, DIRC postpones the processing of release invalidation messages to the next acquire. However, it is a single writer protocol and does not use twins and diffs to propagate modifications as we have seen in the TreadMarks LRC protocol. SoftFLASH uses a 100Mbyte/sec HIPPI (High-Performance Parallel Interface) LAN for cluster communication.

In SoftFLASH, processes in a single node share a virtual memory page frame and a common page table. When a processor changes a page access permission, it uses TLB shootdown to synchronize with other processors. This proved to be a costly operation [ENCH96].
Cashmere

Cashmere [Ste99] is a page-based S-DSM system on a cluster of SMP nodes. It implements a two-level multiple writer, moderately lazy release consistency protocol. On the two-level system, applications benefit from reduced sharing overhead within a single node and sharply reduced protocol communications. Like Munin and TreadMarks, Cashmere uses twins and diffs to collect modifications and support multiple writers. Furthermore, a novel incoming differ technique avoids TLB shootdown in the two-level system, which can be expensive on many architecture platforms.

Cashmere takes advantage of the high performance networking provided by the Memory Channel [FG97]. Important Memory Channel features include remote write support, total message ordering, and low-cost broadcasting. The remote-write feature simplifies the maintenance of Cashmere meta data structures and application data. An interesting insight [SDK+00] from the Cashmere study is that the low-latency messaging is more beneficial than the remote write support.

The focus of this dissertation to extend Cashmere's two-level structure to the third level, allowing applications on high-performance clusters to share state seamlessly with remote sharers. At level-1 and level-2, InterWeave leverages the existing software Cashmere and hardware SMP consistency protocols. At level-3, InterWeave supports sharing among heterogeneous machine architectures and allows aggressive coherence optimizations with application-level information.

2.2.4 Heterogeneous S-DSM

Mermaid

The Mermaid [ZSLW92] system tries to extend the IVY [LH89] system to heterogeneous platforms. It adopts the IVY architecture for distributed shared memory,
i.e., the sequential consistency model and an SPMD programming model. Mermaid adds functionalities to convert data between heterogeneous computing nodes. To allow correct translation, Mermaid requires that data objects in a single page must have a uniform type. When receiving a page update, the receiver translates the page based on the pre-registered architecture information about the sender. Because of the limited support for different data format and their translations, Mermaid's applications are seriously limited.

Agora

Agora [BF87] clusters shared objects into collections. Each object is addressed by a unique string-based identifier. To access a collection, an application maps it into the local memory environment. Application specific translation functions are needed to translate objects among heterogeneous nodes. Agora defines translation procedures for primitive types to relieve some of the programming efforts. To simplify the caching scheme, all objects are immutable. Modifications require creations of new objects and obsolete objects are automatically garbage collected.

2.3 Persistent Distributed Shared State

Traditional S-DSM systems are optimized for parallel applications on tightly coupled clusters of workstations. Shared state is also prevalence in more loosely coupled systems. These applications usually involve more complex data structures and execute over network links with far less bandwidth and higher latency than the cluster environments. In this section, we discuss some of the projects that support distributed shared state over the wide area network and the Internet.
**PerDis**

PerDis [FSB+98] supports sharing of large volumes of complex objects across wide-area networks. Shared objects are organized into clusters. Every cluster has a designated home-site as its primary storage site. A cluster has a root object as the entry point and objects that are unreachable from the root are automatically garbage collected. PerDis supports pointers, including inter-cluster pointers. However, it does not support sharing across heterogeneous machine types.

An application fetches and caches a remote object either by explicitly issuing a *hold* message or by faulting on the corresponding pages. Reader or writer locks are required to accompany cache requests. The unit of hold can be a single object, or any continuous address space up to a cluster.

Accesses to PerDis objects are protected by transactions. PerDis allows both pessimistic transaction control and optimistic transaction control. PerDis does not support relaxed coherence, instead higher concurrency is achieved through the optimistic transaction control. In the pessimistic transaction method, the PerDis runtime takes a writer lock as soon as a cache hold has been issued. The lock is released when the transaction commits. Concurrency is thus limited by excluding conflicting transactions. The optimistic control only acquires a reader lock during the transaction process and allows greater concurrency. In the mean time, transactions are more likely to abort because of concurrent writers.

**Khazana**

Khazana [CRS98] provides an infrastructure for allocating and accessing distributed shared state. The shared data is addressed within a 128-bit global address space. The data are untyped, and subject to application interpretation. Clients can reserve, allocate, and access regions of contiguous addresses. Once access to
a region is obtained, the client can access the data by explicitly using \texttt{read()} or \texttt{write()} calls. Alternatively, the client can map the region into local memory address space.

Regions are managed by a global data structure, called address map. The address map maintains the meta data information of reserved and free regions within the global address space. The address map is itself stored in the Khazana address space, from a well known address of 0. The map is cached and kept relaxed consistent by the Khazana nodes. For scalable region management, nodes are grouped into clusters. Khazana allows user-defined consistency management for individual regions, although, at the time [CRS98], only the release consistency model was supported. Two possible applications were discussed. One is a distributed file system, and another is a distributed object system. It is not clear how heterogeneity is handled in Khazana.

\textbf{Stampede}

Stampede [RNH+99] is a system designed specifically with multimedia applications in mind. A data sharing abstraction called \textit{space-time memory} (STM) allows threads to access a time-sequenced collection of data items easily and efficiently across a cluster. The STM organizes data from different sources into separate channels. Threads can \texttt{put} or \texttt{get} data to or from channels by making connections. The Stampede runtime system automatically garbage collects data items marked by the consuming threads.

A later version of Stampede [APR02] extends it to allow remote devices to connect to the STM provided on a cluster. Limited heterogeneity is supported. Unlike InterWeave, STM does not provide a shared programming model. Interactions with STM by \texttt{put} and \texttt{get} are reminiscent of message-passing methods and of Linda[Gei85]. On the other hand, InterWeave attempts to provide semantics
similar to those of hardware shared memory, and therefore retains only the latest version of shared data.

Legion

The Legion [GW96] system is designed as a wide area operating system. Legion can run on unmodified existing systems. The underlying processors and resources are all abstracted as objects. An object is an abstraction of a system resource rather than shared state in InterWeave. Objects are globally addressed and managed by its associated Class Manager, which in turn is also a kind of meta object. Class managers are organized into domains. In Legion, objects are active and persistent. As an example, Legion's file system is simply a persistent storage object implemented with standard file operations. Parallel computing is supported in Legion more in a message passing style than in a shared memory style.

Linda systems

A Linda system [Gel85] organizes shared state into a tuple space. Applications access the tuple space by two basic operations—\texttt{out} and \texttt{in}. An application uses \texttt{out} to write a tuple into the tuple space, and uses \texttt{in} to take a tuple out of the tuple space. Synchronization is implicit through blocking on the \texttt{in} operations. Transactions are used to keep the tuple space coherent. The Linda system has many derivatives, e.g., JavaSpace [Sun99a], Lime [PMR99], and Scientific [sci99]. JavaSpace [Sun99a] extends JavaRMI and Jini [Sun99b] to provide a Linda-like global tuple space in the strongly typed Java language system. Lime [PMR99], which is also a Java system, uses the tuple space to support mobile computing. Scientific [sci99] is a Linda implementation designed for scientific parallel computing.
2.4 Application-Level Coherence Specification

Some S-DSM systems we have discussed above allow application specific coherence models, such as Agora [BF87], Munin [CBZ91], Legion [GW96], and Khazana [CRS98]. In this section, we discuss other distributed object systems that allows similar application-level coherence controls.

Globe

The Globe system [vSHT99] defines a uniform architecture for distributed shared object systems. Objects are addressed by using a path name that can be resolved at runtime (in a DNS like mechanism). Before using a shared object, an application needs to bind with the object and loads the object's interface implementation into the local node. A local object is divided into four local sub-objects: the communication object, the replication object, the semantic object, and the control object. The replication object decides whether the object's contents (i.e., internal state) needs to be replicated. If replication is necessary, consistency and coherency are dealt with by the communication sub-object. This design provides the flexibility for Globe to support customizable replication and coherence policies for different objects.

POOM

Problem-Oriented Object Memory (POOM) [KL95] is a distributed object model that allows exploitation of application-specific semantics. Applications with intensive shared data writing can achieve performance improvement by relaxing strict consistency requirements. Specifically, it allows multiple object replicas to be modified in parallel and uses a value "amalgamation" process to merge the state of diverged replicas to a single meaningful value. In comparison, InterWeave's relaxed coherence models serve mainly to improve performance for readers of shared
data. Views allow applications to define arbitrary portions of segments as the coherence unit.

Manta

Manta [MKB00] is a Java-based distributed object system. Its coherence model resembles that of Orca [TKB92], a predecessor of Globe [vSHT99]. Manta objects are replicated to exploit localities to reduce remote operation overhead. Read-only methods execute locally on replicas. Object update is accomplished by running the write function remotely at each replication site. To avoid cycles during this process, Manta defines object clusters that are closed sets of shared Java objects reachable and accessible only from a single root object. All objects in a cluster are replicated as a single unit. A writer can thus propagate modifications to a cluster safely and efficiently by broadcasting the operations to all of the replicas. InterWeave similarly follows pointers to expand a recursive view scope, but has no restrictions on the choice of root objects.

Object View

Object View [LPK99] uses programmer knowledge to reduce false sharing by allowing applications to specify the object sharing patterns in the computation process. As an example, VJava implements object views on a Java platform and uses a region-based DSM to allocate shared objects. It extends the Java language with two new keywords view and represents. A view subclasses an original Java object class and exposes only those fields that the application is interested in. A thread uses represents to declare fields that are visible in a specific view. At compile time, a translator computes for each view and classifies their fields with similar access patterns. Fields belonging to the same category should be allocated in the same region (DSM) to reduce communication overhead. VJava does not support view of arrays, and this could seriously limit its applicability in parallel
and distributed computing. In comparison with Object View, InterWeave views do not rely on language extensions, and can be composed dynamically.

2.5 Replication and Caching Management over the Internet

In this section, we discuss some data replication services over the Internet, including Bayou [TTP+95, PST+97], TACT [YV00, YV01, YV02], Deno [CKBF03], and Data GriD service [SSA+01]. We are interested in their consistency and coherence mechanisms that are relevant to our research.

Bayou

The Bayou system [TTP+95, PST+97] assumes a mobile computing environment where network connection is less than that ideal – disconnection is the common, rather than exceptional, case. It maintains a weakly consistent replicated storage system that allows each replica to be updated anywhere. Eventual consistency among replicas is achieved through a pair-wise anti-entropy protocol that is flexible enough to support arbitrary communication topologies. Replica update is incremental and can be supported over low-bandwidth networks. Bayou’s most relevant feature to this dissertation is its support for application-specific conflict detection and resolution.

In the Bayou system, an application can specify its notion of data conflict and provide its own procedure to resolve conflicts. At each Bayou write to a replica, the application provides two additional operators to the system besides the data to be updated. One is called dependency check that consists of an application supplied query and its expected result. Another operator is a merge procedure, which will be run by the Bayou server to try to resolve conflicts if the dependency
check fails. Combined, the Bayou applications can efficiently implement various degree of consistency enforcement based on application semantics.

TACT

TACT [YV00, YV02] explores the semantic space between strong consistency and optimistic consistency for replicated services. TACT considers a replica update as the merge of writes from multiple replication sites. Delaying the replica updates can reduce communication overhead and increase performance. On the other hand, unbounded inconsistencies could lead to semantic failures. TACT proposes to allow applications to define consistency requirements based on individual conits – each of which is a consistency unit. For each conit, a function maps its current consistency state into a numerical value, for example, using the number of weighted uncommitted writes into the replica. A conit’s (thus the replica’s) consistency requirement is bounded along three dimensions: Numeric error, Order error, and Staleness. Numeric error bounds the discrepancy between the locally observed numeric value of the conit and the ideal value if strong consistency is enforced. Order error measures the weighted out-of-order writes that affect a conit. Finally, staleness bounds the maximum age in real time of the oldest write to the conit that has not been seen locally. To reduce traffic, TACT executes updates through an anti-entropy process, transferring only the writes that have not been seen locally.

Accurately enforcing the above metrics requires globally consistent information, which is not available without incurring large communication overhead. TACT approximates estimation conservatively. Here we use TACT’s split-weight AE algorithm as an example. This algorithm is used to bound absolute numeric error. TACT keeps two numbers for this purpose. One number $x$ is the total weight of negatively weighted writes to the replica; another number $y$ is the total weight of the positively weighted writes to the replica. Thus, to guarantee $(n - 1)$
other replica not missing an $\alpha$ error bound, TACT keeps the invariable conditions $x \geq -\alpha/(n-1)$ and $y \leq \alpha/(n-1)$. If either condition is not held, TACT requires the replica to *push* the local updates to other remote replicas before committing a local write. This conservative estimate may cause replica servers to send unnecessary update communications.

The first TACT paper [YV00] does not fully address the question caused by increased inconsistencies. Later, TACT addresses the problem in the context of service availability in an unreliable networking environment [YV01]. When a network connection fails, a replica might not be able to provide service for the incoming write requests because the local replica has exceeded the inconsistency bounds, either in numeric error, order error, or staleness error. Their experiments show that relaxing consistency can increase service availability. However, the availability is best served with an aggressive update strategy, i.e., each replica aggressively pushes updates to other replicas to keep them strongly consistent. Subsequently when network failure occurs, a replica with relaxed consistency can provide a large "cushion" before the user's write requests have to fail.

**Deno**

Deno [CKBF03] addresses the object replication problem in a transactional weakly connected environment. Like TACT, local operations proceed without contacting remote peers, but transactions may abort because of conflicts discovered when servers exchange information in a pair-wise flow model. Unlike TACT, Deno does not bound the inconsistencies among servers. Instead, it tries to preserve weak consistency among committed transactions in a fully-decentralized manner.

Deno uses a *bounded weighted voting* scheme to allow a server to commit or abort transactions without *ever* having a whole system image. Replica servers participate in a group membership. Each server in the group is assigned an amount of voting currency and the total amount of currency in the group always adds to
1. A server casts its vote to a newly seen transaction that it executed locally or it learned from remote servers. It casts a "yes" vote (some nonzero value) if it finds no conflicts with existing locally committed transactions. Otherwise, it casts a "no" vote (0). Note, a local transaction can be blocked if it conflicts with other non-committed transactions. Servers exchange learned votes when they synchronize in a pair-wise fashion. A server can then commit a transaction if it has collected enough voting currency ($ \geq 0.5$) for the transaction. The base Deno voting protocol is proved to support weak consistency [CKBF03].

The base Deno protocol could be easily manipulated by malicious insiders. For example, a single server can stage a denial-of-service attack by refusing to vote on anything. Deno solves this with a currency revocation protocol. More interestingly, Deno looks into the problem that malicious servers can corrupt replica consistency by vote mis-representation. They extend the base Deno protocol to validate votes and discount invalidated votes during the commit stage. The extended protocol is still fully decentralized.

Both TACT and Deno deal with weakening consistency for replicas. InterWeave is looking at reducing overhead when programmers do want a coherent and consistent content. They can be complimentary. For example, replica consistency protocols can be used to replicate InterWeave servers to increase their scalability and availability, while the InterWeave techniques can be used for optimizing application accesses to replica servers and for managing replica meta data structures.

**Grid computing**

Grid computing is an international effort to build "coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations" [Fos01]. A core building block of Grid is the Globus Toolkit [FK97]. It is a set of middleware tools that is intended to be built into a meta computing environment. One
of the tools that have been widely deployed is the GridFTP. GridFTP improves upon FTP by supporting security, parallel data transfer, striped data transfer, partial file transfer, TCP window negotiation for wide-area network, etc.

One particular example of Grid applications and the most relevant to Inter-Weave is the Data Grid project [DMP03]. It was initiated by laboratories in the European Union. The motivation and the particular challenges are to share large amounts of experimental results among multiple research institutions over the Internet. Such experiments include high-energy physics experiments, biology and medical applications, and earth observation applications. For example, on the Data Grid web page [DMP03], it is claimed that “the European Space Agency missions involve the download, from space to ground, of about 100 Gigabytes of raw images per day”. Like other Grid systems, it combines many techniques to provide security, availability, and scalability [FVWZ02, RF02, SSA+01]. Here we only discuss its replication method.

In the general data grid architecture [SSA+01], replications are provided through three services: the storage manager service, the data mover service, and the replica catalog service. Experiment results are stored in database files. The storage manager provides mass storage service. The data mover uses the Globus GridFTP to transfer these database files. File replications are managed by a centralized replica catalog service, which provides the mapping from a file’s logical name to its physical location. Replicas are treated as read-only objects, simplifying the consistency management. A database file might contain millions of objects (events). It is observed that a GRID application usually filters through a large amount of files by a single object value. To avoid bandwidth waste to transfer all of these files, data grid supports object-level replication, at a finer-granularity than the file-level replication. At the beginning of a replication cycle, the needed objects are first identified and those that have not been replicated are copied from the remote site.
Data grid provides efficient sharing of large data files. InterWeave supports efficient sharing of typed data structure and is more appropriate for meta-data level sharing. Data grid does not incorporate various relaxed coherence models to improve bandwidth utilization, although object replication supports optimization by limiting the replication scope.

2.6 Java-based Distributed Systems

There are a couple of Java shared object systems that are built upon existing distributed shared memory (DSM) systems, such as Java/DSM [YC97], Hyperion [ABH+01], Jessica [MWLX99], and the aforementioned VJava [LPK99]. They create Java objects on the heap of an S-DSM shared address space and require the modification of the Java virtual machine. The JVMs rely on the virtual memory system to detect writes to the shared objects. This latter feature can possibly limit the JVM choice of better garbage collectors other than the “Mark-and-Sweep” because an object location has to be fixed after creation. Although the InterWeave 3-level Java implementation allocates Java block objects on the DSM system for the level-2 system, the level-3 system doesn’t rely on the existence of underlying DSM.

Hyperion and Jessica, along with MultiJav [CA98] take advantage of Java’s relaxed memory model. Hyperion compiles the Java bytecode into the native machine code so that high performance can be achieved. MultiJav modifies the JVM and transmits the modified objects in a “diff” format. Writes to shared objects are tracked by virtual memory in the JIT engine or by instrumenting code in the interpreter engine.

JOIE [CCK98] is developed for instrumenting the Java class file both dynamically and statically. The key idea is that a customized class loader can do the byte code transformation transparently. IVORY [BCC+99] uses JOIE to enable
the caching of both Java servlet and its data object so that the caching of dynamic web pages becomes possible at the network proxy site. Chen et al. [CMB+01] describe a technique of instrumenting the Java byte code interpretation in the JVM so that the it is possible to track memory errors inside the JVM. BIT [LZ97] is another Java instrumentation tool designed for analyzing dynamic behaviors of Java applications.

2.7 Remote Procedure Call Systems

Remote procedure call (RPC) is an elegant concept to conduct distributed computing tasks. The details are first figured out in the Courier system [Cor81]. Birrel and Nelson [BN84] describe the RPC system implementation in the Cedar system. They describe the interface description languages, the automatic generation of client stubs and server skeletons, the implementation of naming and locating services, and the transport protocol; all of which are essential to the modern-day RPC systems.

Corba [Vin97] and Microsoft's DCOM [Rog97] are both RPC systems in the guise of an object-oriented framework. Objects are defined in their specific IDL languages and are translated into the target language. Parameters and invocation results are marshaled during the RPC's execution. The standard CORBA and DCOM systems don't support object caching. Flex [KA96] is a CORBA compliant system that supports object caching. The system is organized into several frameworks: the consistency framework, the cached object framework, and the implementation framework that provides the actual system implementation. The cached object framework decides when and where to cache an object. The consistency framework provides multiple consistency protocols for inter-object consistencies (the paper only mentions strong consistency and causal consistency).
Smart RPC

Smart RPC [KKM94] discusses a method of passing pointer arguments in a RPC system without deep copying. It uses virtual memory to detect dereferences of remote pointers and brings pointed-to data on demand. When passing a pointer, the caller marshals the pointer into an extended, longer format. The callee unmarshals the longer pointer and reserves a memory space for the pointed-to data content. The callee then invalidates the memory space to detect reads into the area. Upon page faults, the callee makes a call-back to fetch data from the caller; and then enables user access. The callee caches previously fetched data in its local memory. However, the cache is only valid within individual sessions delimited by the caller. Unlike InterWeave, the cache update is page based and does not use diffing.

2.8 Summary

InterWeave supports distributed shared state with ordinary reads and writes. Traditional S-DSM systems [LH89, CBZ91, ACD+96, IS99, SGT96, SFL+94] are mostly developed on tightly coupled clusters consisting of machines of homogeneous types and connected with high speed system area networks. Heterogeneity support is rare, and tends to be inflexible and limited in functionality [BF87, ZSLW92, APR02]. InterWeave, however, extends two-level S-DSM systems [ENCH96, Ste99] to a three-level structure to encompass support for heterogeneous machines and multiple languages over the Internet.

InterWeave assumes a relaxed consistency model [GLL+90, ACD+96, BZ91, ISL96] and limits coherence and consistency updates to application synchronization points. InterWeave further allows the use of application-level coherence information to optimize communication performance along the temporal and spatial dimensions, in a manner more flexible and dynamic than previous systems [BF87,
vSHT99, CRS98, LPK99, GW96, MKB00, KL95]. Recent research in replica services [CKBF03, YV00, SSA+01] over the Internet has explored relaxing consistency conditions to improve performance and service availability. InterWeave can be complimentary to them in providing a “programmer-friendly” programming system and can be used for accessing distributed shared state. Unlike ad-hoc RPC caching schemes [KA96, KKM94], InterWeave automates coherence management and supports genuine memory object pointers to eliminate “deep-copy”.
3 Efficient Distributed Shared State

InterWeave serves as a middleware system to support efficient access to distributed shared state. InterWeave assumes a client/server model. Servers maintain persistent copies of shared data and coordinate sharing among clients. Servers are uniquely identified by their URLs. Similarly, every primitive data item inside a server has its own unique ID. With the two components combined, InterWeave allows shared state to be addressed uniformly across the Internet.

InterWeave applications are all clients of some InterWeave servers. The applications must be linked with a client library to be able to communicate with the servers. The library provides a programming interface that allows processes to create typed data structures at specified InterWeave server locations. Once a segment has been created, clients can map it into the local address space so that the data it contains can be accessed with ordinary reads and writes. The servers and client library automatically ensure the coherence and consistency of the shared data assuming the applications are properly synchronized.

The client/server model simplifies the design and implementation of the runtime system. However, applications do not assume any particular server structure. An InterWeave process can be a client of many different InterWeave servers and is ignorant of other peer processes of the same server. Competitive accesses to the
same segment are synchronized seamlessly by using segment locks.

In the rest of this Chapter, we first describe the InterWeave application programming interface, including data allocation, synchronization, and support for heterogeneity. We then describe the implementation framework for both the InterWeave server and the client library. We conduct experiments and present performance evaluations using microbenchmarks. Some applications are also presented to demonstrate the InterWeave programming features.

3.1 InterWeave Application Programming Interface

The specific form of InterWeave programming interfaces depends on individual programming languages. For example, in C/C++ they take the form of a set of InterWeave library functions. In the Java language, all of the functions are encapsulated in a specific InterWeave class. However, they all provide the same semantics and functionality. In this Chapter, we describe the InterWeave design in the context of a C/C++ programming environment. A more detailed description can be found in a separate InterWeave Programming Guide [CTDS04].

3.1.1 Data Allocation

InterWeave applications use *Segments* to organize and allocate shared state. The segment is the coherence unit of sharing in InterWeave. A segment is a self descriptive data structure, in which client processes can create strongly typed *blocks* of memory. A programmer can view a segment as a logical section of a shared heap in which blocks of contiguous addresses can be allocated. Because InterWeave is designed for sharing in a heterogeneous environment, a segment
and its blocks might take different local formats when being mapped in different machine architectures and thus might have different sizes.

Each segment is managed by an InterWeave server at the IP address corresponding to the segment's URL. Different segments may be managed by different servers, and may require different access rights from different clients. A segment is created by an InterWeave client by calling \texttt{IW\_create\_segment()}:

\begin{verbatim}
IW_handle sh = IW_create_segment(segment_url);
\end{verbatim}

Like all other InterWeave functions we are going to describe, \texttt{IW\_create\_segment()} is available assuming the client program is linked with an appropriate InterWeave client library. The client library will contact the InterWeave server located at \texttt{segment\_url} to create the meta data structure for the segment.

An existing segment can be opened by calling \texttt{IW\_open\_segment()}:

\begin{verbatim}
IW_handle sh = IW_open_segment(segment_url, iscreate);
\end{verbatim}

Both \texttt{IW\_create\_segment} and \texttt{IW\_open\_segment} return a segment handle whose content is opaque to the caller. It is used like a file handle in other segment related functions, such as allocating blocks, requiring segment locks, and setting segment coherence models.

A block is a contiguous section of memory allocated in a segment. Applications allocate blocks by calling \texttt{IW\_malloc}. \texttt{IW\_malloc} has two parameters, the handle to the segment and a type descriptor for the block. The latter provides necessary information for the InterWeave system to reserve memory addresses for the block and to translate the block between a machine dependent local format and the InterWeave machine independent wire-format.

Each block has a segment-wise unique serial number. The number is assigned during the \texttt{IW\_malloc()} call. A block can also be optionally named with a user-specified string by calling \texttt{IW\_named\_malloc()}. 
IW_wl_acquire(h);
my_type* p = (my_type*) IW_malloc(h, my_type_desc);
*p = ...
IW_wl_release(h);

Programmers use the InterWeave Interface Definition Language (IDL) to define types of blocks. The IDL is based on the SUN XDR notation and is rich enough to describe any arbitrarily complex combinations of primitive types, arrays, structures, pointers, and strings.

An IDL compiler is provided to translate a type definition into a format linkable to the used program. Each type in the file has a corresponding type descriptor. The descriptor provides detailed layout information for the types on the target machines. Prior to their use, they must be pre-registered with the local client library with IW_register XXX() calls, where XXX is the name of the IDL file. The InterWeave client library registers primitive type descriptors automatically at initialization time. User-specific type descriptors are registered depending on the data types used in the application.

### 3.1.2 Synchronization

InterWeave clients use reader/writer locks to coordinate access to shared segments. Each segment has its own reader/write lock. A client process uses IW_wl_acquire() and IW_r1_acquire() to acquire reader/writer locks, and releases them by IW_r1_release() and IW_wl_release(). All of these APIs take a parameter of the segment handle.

    IW_wl_acquire(sh);
    /* write critical section */
    ........
IW_wl_release(sh);
...
IW_rl_acquire(sh);
/* read critical section */
...
IW_rl_release(sh);

Appropriate use of segment locks is crucial in InterWeave programming. The InterWeave coherence and consistency models assume that segment sharers are properly synchronized with each other. In other words, there must be no race conditions in using InterWeave shared state: a critical section containing writes to a segment must be protected by the segment writer lock; and reads to the segment must be protected by the reader lock. For properly synchronized applications, the system will guarantee the coherence and consistency of touched segments.

Like most other shared memory systems, multiple sharers can own reader-locks simultaneously, but only one sharer can own the writer lock at a time. Unlike the traditional exclusive writer locks, a segment client can hold a relaxed form of reader lock while another client is locking the segment for write, depending on its own coherence model.

When locked, a segment’s content is mapped into the client’s local memory address space. In general, the InterWeave client library obtains recent segment updates from the segment server when the segment lock is acquired. However, when using relaxed coherence models the update can be delayed. Correspondingly, when using dynamic views, the scope of update can be limited. When a client finishes writing to a segment and releases the writer lock, the runtime system automatically collects the updates and sends it to the server.

InterWeave programmers use relaxed coherence models and dynamic views to provide the runtime system with application level information for aggressive communication optimizations. At any time, a client can change its current coherence
model by calling `IW_use_coherence()`. The client can call `IW_create_views()`, `IW_attach_view_unit()`, and `IW_activate_view()` to create, populate, and activate a view. Details of these API functions will be elaborated on in Chapter 4.

### 3.1.3 Pointer Translation

Every data element in an InterWeave segment can be uniquely addressed with an Internet URL like string, which is called a machine independent pointer (MIP). A MIP is composed of four components: the URL address of the segment, the name of the segment, the name or the serial number of the block, and an optional offset in terms of primitive data units—characters, strings, integers, etc.—instead of individual bytes. Both the block name and the offset are preceded by `#` signs. The general form is thus the form of "foo.org/segnamede#block#offset".

```c
/* char *mip = "foo.org/segnamede/1#0"; */
my_type *p = (my_type *)IW_mip_to_ptr(mip);
```

A MIP can be concatenated manually from the segment URL, block name or serial number, and element’s primitive offset inside the block. Or the user can ask the library to translate a local InterWeave pointer to its corresponding MIP format:

```c
/* my_type *p = ... */
char *mip = IW_ptr_to_mip(ptr);
```

The resulting MIP can be later sent to other segment sharers via network, file, or any other communication method.

### 3.1.4 Events

Many distributed applications require an event notification mechanism in order to respond to asynchronous operations by non-local processes. Our API provides an
IW_register_event() call that allows a process to specify a handler that will be invoked automatically when a specified segment or block has been modified and the locally cached copy is no longer recent enough.

3.2 Framework of Implementation

The InterWeave implementation requires efficient coordination of the client library and the segment server. Both the library and the server can be divided into the following relatively independent modules:

- Communication Management. This module handles the communications between the client and server.

- Content Management. This module manages data type and content on the client and server. When necessary it translates segment data between an InterWeave machine-independent format and the client local machine format.

- Coherence and consistency management. This module manages the coherence status of segment sharers and ensures an optimized scheme of segment update and modification propagation.

In this section, we describe the implementation details that are sufficient for understanding the basic InterWeave performance presented in this Chapter. Discussions of the implementation of the relaxed coherence models, dynamic views, multi-level architecture, and Java support are left to later chapters.

3.2.1 Communication Management

InterWeave clients use TCP-based connections to communicate with segment servers. While the use of UDP might help improve performance, TCP greatly
simplifies our implementation. InterWeave uses TCP_NODELAY to disable the Nagle Algorithm [Nag84] on the TCP sockets. The Nagle Algorithm is used to aggregate small messages for better congestion control and bandwidth utilization. However, it might delay small InterWeave protocol messages. InterWeave uses readv() and writev() to explicitly collect small packets into fewer larger packets to avoid bandwidth waste.

**Synchronous TCP Connection**

A client establishes a TCP connection to a segment server when it calls either IW.create_segment() or IW.open_segment(). A server listens to a well known port (8009) to accept connection requests from clients. During the connection process, the application can instruct the client library and the server to use their own pre-generated *identity keys* to authenticate each other.

The client uses this connection for synchronous communication with the server, sending the InterWeave operation requests and waiting for the reply. Potential client requests include segment closings, requesting and releasing reader/writer locks, setting coherence models, and creating and deleting views.

The server creates a dedicated thread for each segment to handle all client requests. Because a server can host multiple segments, multiple threads are created to exploit potential concurrency by overlapping the I/O operations and the computation processing for different segments.

Message handling in each segment thread is, however, event-driven and uses the system call select. Messages from different clients are processed in a round-robin manner to guarantee fairness of service. During our implementation process, we found that multithreading is not efficient in processing single segment requests. When there are many client requests, contention for the segment metadata structures results in frequent context switches, which is especially expensive on operating systems that only support "heavy threads" (kernel-level threads).
Additionally, there is little concurrency exploitable by multithreading when processing requests for a single segment. Thus, we adopt the event-driven method to eliminate context switch overhead and to allow the server to achieve better throughput.

The scalability of the segment service is not the focus of this work. Nonetheless, the current InterWeave server structure that combines the multithreading and event-driven models is similar with other research systems on scalable network services; for example, Flash [PDZ99], HACC [FGC+97], SEDA [WCB02], and Neptune [STYCO2], just to name a few.

**Asynchronous TCP Connection**

Occasionally, the server needs to initiate messages to the client. These messages include registered client events, heartbeat messages (explained in the following paragraph), adaptive polling notifications (to be described in Section 4.4.1), and notifications to quit exclusive-owner mode (to be described in Section 3.2.3). During the connection process described above, the client and the server open another TCP connection to handle these messages. The client marks the socket as an asynchronous connection.

The heartbeat message is necessary to prevent a segment from being locked forever by a dead client holding a writer lock. The server sends heartbeat messages periodically to the writer when there are other lock requests. The writer returns a short message to indicate its liveness. If the server does not receive a response for three consecutive heartbeats, it relinquishes the lock ownership and grants it to the next requester. If the client is falsely diagnosed, its next lock release request will fail. InterWeave currently requires the client check every synchronization operation for possible failures and handle them manually.\(^1\)

---

\(^1\)A recent InterWeave transactional system for automatic recovery is discussed briefly in Section 7.2.
3.2.2 Content Management

The InterWeave system manages and organizes segment content with the goal of optimizing frequent operations. The client keeps a cached copy of the segment in its local format. The server, to support heterogeneous clients, always keeps the segment content in a machine-independent wire-format. Most of the time, the server has to efficiently merge updates from the clients and compose updates to the clients. On the other hand, the client must efficiently apply updates from the server, collect modifications to send to the server, and translate between the wire-format and its local memory format.

Client-side Segment Management

As we have described before, a segment is versioned and consists of an unlimited number of blocks. Each block must have a well-defined type, but this type can be a recursively defined structure of arbitrary complexity. Each block has a unique serial number inside the segment, assigned when created by \texttt{IW.malloc()}. It may also have a symbolic name, specified as an extra parameter. Multiple types of blocks may be created in a single segment.

A client does not have the content of an opened segment mapped until the first time it acquires a lock on the segment. When opening, the client library simply makes a connection to the segment server and resets its available local version to -1. When closing a segment, the client sends a quit message to the server and closes all TCP sockets. It releases any local memory that has been used.

When locked, a segment has its contents mapped into the local memory address space, though it need not be a contiguous region. However, like memory blocks allocated through \texttt{malloc()}, individual segment blocks must be mapped to contiguous space to allow correct operation on local pointers. To balance the size of meta data structures and the ease of extending a segment, we organize
segment memory into user oblivious subsegments. Each subsegment contains multiple pages of contiguous space to support arbitrary sizes of blocks. Rather than relying on the standard C library malloc(), the client manages its own heap area for the subsegments.

Figure 3.1 illustrates the organization of the client memory into subsegments, blocks, and free space to support such conversions. The segment table has exactly one entry for each segment being cached by the client in local memory. It is organized as a hash table, keyed by segment name. In addition to the segment name, each entry in the table includes four pointers: one for the first subsegment that belongs to that segment (FirstSubSeg), one for the first free space in the segment (FreeList), and two for a pair of balanced trees containing the segment's blocks. One tree is sorted by block serial number (BlksByNum), the other by block symbolic name (BlksByName); together they support translation from MIPs to local pointers. An additional global tree contains the blocks of all segments, sorted by address (SubSegsByAddr); this tree supports translation from local pointers to MIPs.

Each subsegment uses a balanced tree sorted by address to keep pointers to blocks (BlkByAddr). A subsegment also keeps an array of pointers, called wordmap, for saving addresses of twin pages (Section 3.2.3). Each block in a subsegment begins with a header containing the size of the block, a pointer to a type descriptor, a serial number, an optional symbolic block name and several optimization flags. Free space within a segment is kept on a linked list, with a head pointer in the segment table. Allocation is currently first-fit. To allow a deallocated block to be coalesced with its neighbor(s), if free, all blocks have a footer (not shown in Figure 3.1) that indicates whether that block is free and, if it is, where it starts.
Figure 3.1: Simplified view of InterWeave client-side data structures: the segment table, subsegments, and blocks within segments. Type descriptors, pointers from balanced trees to blocks and subsegments, and footers of blocks and free space are not shown.
Translation

Segment clients and server all keep type descriptors to facilitate translation of segment data among different machine types. The system recognizes some predefined primitive types, including various integer and floating point types, strings, and MIP pointers. Arbitrary complex arrays and structures must be defined with the InterWeave IDL and compiled to generate a description of the data type. The description gives details of the number of data elements, their types, and offset in the data structure.

At the client, type descriptors must be preregistered with \texttt{IW.register.XXX()} before use so that there is enough information for the client library to translate user's shared data structure latter. When \texttt{IW.malloc()} is called, the type descriptor is added to the segment if it has not been used before. It is also assigned a segment-specific serial-number. Later, the type descriptor is sent to the server with other segment modifications.

Server-side Management

Each server maintains an up-to-date copy of each segment it serves. Since server and client always exchange segment data in the wire-format, the server keeps both type-descriptors and segment data in the wire-format to avoid an extra level of translation. Each segment has a version number which is increased whenever a new modification is propagated from a writing client. Blocks in a segment are organized into a balanced tree sorted by their serial numbers. Each block has a version number (when it was last modified), a pointer to a potential subblock version structure (to be described in the next paragraph), its serial number, a pointer to a type descriptor, a pointer to a wire-format block header, and a pointer to the data of the block, again in wire-format.

The body of a block is potentially divided into multiple subblocks, each con-
taining an equal number of contiguous primitive items. Its structure is invisible to the segment client, but helps the server track block modifications at a finer granularity. As mentioned above, subblocks also have versioned numbers, kept in an array. Subblocks offer a finer granularity for the server to compute update diffs when clients request them. Otherwise, any changes to a large block will always result in update of the whole block. Currently, each subblock has 16 primitive data items.

3.2.3 Coherence

InterWeave requires that client processes be properly synchronized. Read accesses must be protected by the reader lock of the specific segment, and write accesses protected by the writer lock. When a client first locks a segment, the client acquires a full copy of the current segment version from the server. The process then can access the segment content at local memory.

Lock requests for the relaxed reader locks can be readily granted without any waiting. Requests for the writer lock and the strict reader lock, however, have to wait for the current lock holder to release the lock. The server puts such blocked requests in a FIFO queue. The requests are serviced as soon as the segment is unlocked.

Two-way Differing

If the client acquires a writer lock, the InterWeave library employs the virtual memory system to write-protect all pages allocated to the segment. When a page is first written to, through IW::malloc() or direct writes, the InterWeave SIGSEGV handler, installed at initialization time, will create a pristine copy, or twin, and store the pointer in the segment meta data structure, and then restore write access to the page. When the client subsequently releases the writer lock, the
library looks through the twin pages created during the critical section. Guided by the type descriptors, the client finds out all of the primitive data elements in modified blocks and translates them into a wire-format diff to be sent to the server. With this process of client-side collecting diff, the propagation of modifications is optimized to only contain the minimal information. Receiving the diff, the server updates the segment version and applies the diff in a converse process to its own copy of the segment in the wire-format. This process is called server-side applying diff.

When a client acquires a lock and determines that its local cached copy of the segment needs update, it asks the server to build a diff in a process called the server-side collecting diff that describes the data that have changed between the client’s outdated copy and the master copy at the server. When the diff arrives the library uses its contents to update the local copy in a process called client-side applying diff, which is also guided by the type descriptors.

Exclusive-owner Lock

When a segment has only one client, the segment server allows the client to enter the exclusive-owner lock mode. The client can subsequently acquire and release locks locally for an arbitrary number of times without contacting the server. The exclusive-owner mode is especially useful in optimizing InterWeave performance on the three-level structure when the level-2 cluster is the single sharer (Chapter 5).

The server grants the exclusive-owner lock when it sees that the client has been the sole sharer in the last three lock requests and is acquiring a writer lock. Being the exclusive owner, the client does not contact the server when releasing or acquiring locks. Instead it just changes the local synchronization state. The client keeps the existing twins and continues to create new ones for pages that are written during critical sections.
The server sends an asynchronous message (Section 3.2.1) to the exclusive owner to quit the exclusive-owner mode when it receives a lock request from a new client. The server then waits for the exclusive-owner to reply. At this time, the exclusive owner might be in one of the following state: (1) unlocked; (2) in a read critical section; or (3) in a write critical section. For state (1) and (2), the client can safely quit from the exclusive owner mode and collect any diff updates by comparing the twins with the current segment content. It then sends the diff and its current lock status to the server. If the client is in state (3), it can not use the current twins to compute a diff. Its current local segment content might be in an incoherent state because of recent writes into the segment. The client has to wait until the next lock release point to collect diffs and update the server. Consequently the client just sends back a response indicating it is holding the writer lock and surrenders the exclusive owner mode.

After receiving the exclusive owner response, the server can process the lock requests from new clients as it would normally do: granting the writer lock request if there are no other writers; or granting the relaxed reader lock request immediately. During the process, the server could collect an update diff for this new client.

3.2.4 InterWeave Runtime System Implementation

At this moment, the InterWeave runtime system implementation contains about 46,000 lines of code in C, C++, and Java languages. Among them, about 13,700 lines are dedicated to the features discussed in this dissertation, including the support for the high-level coherence information specification, multi-level structure, and the support for Java language. A detail decomposition is given in Table 3.1. Other InterWeave components implement the support for heterogeneous shared state and transactional semantics, including about 2,800 lines for basic xdr operations, 22,300 for the client-side and server-side data management and translation,
4,600 lines for transactional support, and another 2,600 lines for other miscellaneous functionalities.

<table>
<thead>
<tr>
<th>InterWeave API</th>
<th>1,500</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client-Server protocol</td>
<td>4,400</td>
</tr>
<tr>
<td>Relaxed coherence models</td>
<td>3,000</td>
</tr>
<tr>
<td>Dynamic views</td>
<td>1,800</td>
</tr>
<tr>
<td>3-level support</td>
<td>1,000</td>
</tr>
<tr>
<td>Java support</td>
<td>2,000</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>13,700</strong></td>
</tr>
</tbody>
</table>

Table 3.1: InterWeave Implementation Decomposition.

### 3.3 Evaluation

#### 3.3.1 Experimental Platform

Throughout this dissertation, we use a collection of machines of different types running with different operating systems to evaluate the InterWeave system. The high-end cluster we use for our parallel applications is an AlphaServer system. Each node in the cluster is an AlphaServer 4100 5/600, with four 600 MHz 21164A processors, an 8 MB direct-mapped board-level cache with a 64-byte line size, and 2 GBytes of memory, running Tru64 Unix 4.0F. The nodes are connected by a Memory Channel 2 [FG97] system area network, which is used for tightly-coupled sharing. Connection to the local area network is via TCP/IP over Fast Ethernet. We also use Sun Ultra 5 workstations with 400 MHz Sparc v9 processors with 128 MB of memory, running SunOS 5.7; and 2GHz PCs with 512MB of memory, running Linux 7.2. Network connections among Sun workstations and PCs are configured to be either a 10Mbps, a 100Mbps, or a Gigabit Ethernet.
<table>
<thead>
<tr>
<th></th>
<th>Alpha</th>
<th>Sun</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>twin creation</td>
<td>64.6 (8K)</td>
<td>73.3 (8K)</td>
<td>8.7 (4K)</td>
</tr>
<tr>
<td>page fault</td>
<td>18.0</td>
<td>122</td>
<td>8.5</td>
</tr>
<tr>
<td>mprotect</td>
<td>3.79</td>
<td>11.6</td>
<td>1.54</td>
</tr>
<tr>
<td>IW write fault</td>
<td>88.6</td>
<td>209</td>
<td>26.73</td>
</tr>
</tbody>
</table>

Table 3.2: Basic operation costs (all times in μsecs).

3.3.2 Basic System Operation Cost

Table 3.2 provides statistics on the costs of various basic system operations on each machine type. Twin creation is the cost of copying an uncached page on each platform (the operating system (OS) page size is indicated in parentheses). The page fault time is the OS cost to transfer execution from the faulting instruction to the SIGSEGV handler and to return when the handler is complete; this cost is particularly high on SunOS. The mprotect cost is the average cost per page for changing access permissions. The IW write fault time is the total overhead incurred by the InterWeave system on the first write access after a write lock acquire to any page belonging to an InterWeave segment. This time includes the page fault, twin creation, and mprotect times, as well as some time to search and update InterWeave metadata.

3.3.3 Basic InterWeave Synchronization Operation Cost

Figure 3.2 and Figure 3.3 show the breakdown costs of acquiring and releasing locks on the Linux PC machines. The overhead of a lock acquisition includes the total communication cost to transmit the client request and the server response, the server-side computation cost to assemble a wire-format diffing update ("server collect diff" in the figure), and the client-side computation cost to translate the update from the wire-format to the local machine format ("client apply diff"). The
lock release overhead includes the client-side diffing overhead to find modifications and assemble them into a wire-format update packet ("client collect diff"), the communication overhead to send the packet to the server, and the server-side cost to integrate the update into its segment master copy ("server apply diff").

The segment in study contains an array of 256K integers all of which were initialized to zero. All of the experiment and measurement were collected on 2GHz Pentium IV machines with 512MB memory, running Linux 2.4. The two figures represent different experiments over two different Ethernet connections, a Gigabit Ethernet and a 100Mbps Ethernet. We have also experimented with two different modification sizes. The left two bars in each figure represent the overhead when all integers were modified between acquiring and releasing the writer locks. The right two bars represent the overhead when only 1/16 of all integers in the segment were modified.

The figures show that communication cost is much higher than the InterWeave translation cost. The translation cost at both the client-side and server-side becomes negligible on 100Mbps Ethernet. When the segment was partially modified, sending diffs cuts the communication cost proportionally to the size of the modifications. Diffing also helped reduce client-side and server-side translation cost since there is less data for InterWeave to translate.

The server has generally less cost in collecting and applying diffs than the client has. This is because the segment server always keeps segment data in the wire-format and thus skips any translation cost. The client has higher cost in collecting diffs than applying diffs because collecting requires comparing the twins with the current content to find modifications.
Figure 3.2: Costs of acquiring and releasing locks. The client and server were connected with a Gigabit Ethernet. The segment contained an array of 256K integers. In two different experiments, before acquiring and releasing locks, the integers were either all changed, or only 1/16 of all them were changed. All measurements were in millisecond.

Figure 3.3: Costs of acquiring and releasing writer locks. In these experiments, the client and server were connected with a 100Mbps Fast Ethernet.
3.3.4 Basic Interweave Translation Costs

Table 3.3 compares InterWeave translation costs with those of the RPC parameter marshaling functions generated with the standard Linux rpcgen tool.\(^2\) Besides the integer array we used above, we also tested a double array and a pointer array. The pointer array contains pointers all referencing a single integer variable. In each case, we arranged for the total amount of data to equal 1MB. In the table we also present the InterWeave costs when 1/16 of the arrays are modified.

<table>
<thead>
<tr>
<th></th>
<th>RPC encode</th>
<th>IW collect diff</th>
<th>IW collect diff (1/16)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer array</td>
<td>3.11</td>
<td>3.89</td>
<td>0.238</td>
</tr>
<tr>
<td>Double array</td>
<td>4.86</td>
<td>3.87</td>
<td>0.250</td>
</tr>
<tr>
<td>Pointer array</td>
<td>28.79</td>
<td>22.4</td>
<td>1.41</td>
</tr>
</tbody>
</table>

Table 3.3: Basic InterWeave translation costs (all times in msecs).

Generally speaking, InterWeave overhead is comparable to that of RPC. InterWeave has 25% higher cost in the collect-diff process that generates a wire-format diff for the integer array than RPC in the corresponding process that marshals the array. In the mean time, the InterWeave costs are lower on the double array and the pointer array by 20% and 22% respectively. When the arrays are partially modified, the table shows the InterWeave costs are reduced proportionally.

Table 3.4 presents the size of the resulting wire-format data from both the InterWeave collect-diff process and the RPC marshaling process. We can see InterWeave and RPC generate almost equal amount of bandwidth for the integer and double arrays. The InterWeave format is slightly higher due to more meta data information. However, InterWeave wire-format for the pointer array is half of the RPC's. When RPC marshals a pointer, the deep copy semantics requires that the pointed-to data, the integer value in this experiment, be marshaled along with

\(^2\)In our experiments, we found the RPC unmarshaling costs to be roughly identical.
the pointer. InterWeave does not deep copy the integer, instead it only translates the pointer into a four-byte-long intra-segment offset and thus requires less traffic.

<table>
<thead>
<tr>
<th></th>
<th>RPC wire-format</th>
<th>IW wire-format</th>
<th>IW wire-format (1/16)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer array</td>
<td>1048576</td>
<td>1048760</td>
<td>65700</td>
</tr>
<tr>
<td>Double array</td>
<td>1048576</td>
<td>1050788</td>
<td>65828</td>
</tr>
<tr>
<td>Pointer array</td>
<td>2097152</td>
<td>1048740</td>
<td>65700</td>
</tr>
</tbody>
</table>

Table 3.4: Basic InterWeave bandwidth costs (all traffic in MByte).

A more comprehensive evaluation of InterWeave heterogeneity support can be found in another InterWeave technical report [TCDS02].

### 3.3.5 Parameter Sharing in RPC

We use an application kernel consisting of searching and updating a binary tree to demonstrate the benefit of using InterWeave to facilitate the sharing of pointer-rich data structures between an RPC client and server. The application also benefits from two-way-diffing in reducing communication overhead. In this application, a client maintains a binary tree keyed by ASCII strings. The client reads words from a text file and inserts new words into the tree. Periodically, the client makes an RPC call to the server with one string as parameter. The server uses the string as a keyword to search the binary tree, does some application-specific computation using the tree content, and returns the result.

To avoid requiring the client to pass the entire tree in every call, the server caches the tree in its local memory. We compare three methods by which the client may propagate updates to the server. In the first method, the client performs a deep copy of the entire tree and sends it to the server on every update. This method is obviously costly if updates happen frequently. In the second method, the server invokes a callback function to obtain updates when necessary. Specif-
ically, it calls back to the client when (and only when) it is searching the tree and cannot find the desired keyword. The client will respond to the callback by sending the subtree needed to finish the search. The third method is to share the binary tree between the client and server in the global store provided by Inter-Weave, thereby benefiting automatically from caching and efficient updates via wire-format diffing.

Figures 3.4 and 3.5 compare the above three solutions. The text file we use is from *Hamlet Act III* and has about 1,600 words. The client updates the server after reading a certain number of words from the text file. The frequency of updates is varied from every 16 words to every 320 words and is represented by the X axis in both graphs. After each update, the client takes every other word read since the previous update and asks the server to search for these words in the tree. Under these conventions the total number of calls is proportional to the update frequency. The Y axis shows the total amount of time and bandwidth, respectively, for the client to finish the experiment.

![Graph of time to update and search a binary tree](image)

Figure 3.4: Total time to update and search a binary tree.

As updates become more frequent (moving, for example, from every 80 words
Figure 3.5: Total size of communication to update and search a binary tree.

to every 64 words), the deep-copy update method consumes more and more total bandwidth and needs more time to finish. The call-back and InterWeave methods keep the bandwidth requirement almost constant, since only the differences between versions are transferred. However, the call-back method requires significantly more programmer effort, to keep track of the updates manually. Due to the extra round-trip message needed by the call-back method, and InterWeave's efficiency in wire format translation, InterWeave achieves the best performance among the three.

This example is inspired by the one used in [KKM94] (Section 2.7), where data can be shared between the client and server during a single critical section (RPC call). In our version of the experiment, InterWeave allows the client and server to share the data structure across multiple critical sections and with a more intuitive programming interface.
3.3.6 API Ease of Use

We further analyze the ease of use of the InterWeave API by comparing the InterWeave version and the RPC version of the benchmark used in the above section. In our experimentation, we intentionally shared as much source code as possible between the two versions, such as data initialization and binary tree operations. We used rpcgen to generate client-side RPC stubs and skeletons for both versions automatically. When calling the server for a binary tree service, the InterWeave version passes a machine independent pointer — a string — to the server instead of a pointer for deep-copying in the RPC version.

The two versions differ in the way the binary tree is shared between the client and server. In the InterWeave version, we use about 10 lines of code in C at the client and server to initiate the InterWeave runtime at the server and client (IW.init(), IW_register_type.xxx(), and IW_open_segment()). There are Another 4 lines of code in each version for setting up coherence models (IW_use_coherence()). When calling the server, the client code uses one line to extract the machine independent address of the root of the binary tree (IW_ptr_to_mip()), and two lines of code to acquire and release the InterWeave segment lock (IW_xx.acquire() and IW_xx.release()). The server code, correspondingly, uses two lines to acquire and release the lock, and one line to translate the MIP into its local memory address (IW_mip_to_ptr()).

In the RPC version, the server makes a call back to the client to obtain updates on the binary tree. This requires the extra effort of setting up an RPC service at the client side and extra programming effort correspondingly. Besides, the service code must use extra caution to check the binary tree during the search and make callbacks to the client if missing updates are encountered. There are 72 lines of server code for checking and making callbacks alone.

As our evaluation shows, the InterWeave version is more efficient than the
RPC-callback version. The above analysis further demonstrates that the InterWeave version is much more straightforward than the RPC-callback version. Generally, compared with RPC or other message passing based systems, in this example and our experience with other applications, we find that InterWeave programming is much more intuitive and straightforward.

3.4 Summary

In this Chapter, we have described the InterWeave application programming interface and basic performance results. InterWeave supports a shared memory programming model for managing distributed shared state. Applications use reader/writer locks to synchronize conflicting accesses. With user-provided type descriptors, the InterWeave runtime system can automatically translate data among heterogeneous machine types.

Our experiments demonstrate the efficiency of the InterWeave implementation. The translation costs are comparable to existing highly-optimized RPC implementations. Two-way diffing reduces application bandwidth requirement to be proportional to the size of modifications.

Through our experiences, we found that the InterWeave programming model is easy to use. It helps significantly in automating the management of distributed shared state. Programmers can thus better focus on application logic.

Our experiments also show that InterWeave can be used seamlessly with RPC systems [CDP+00, PCDS00, CTC+02, TCDS03, CTS+03]. InterWeave helps simplify and optimize the caching of remote state. Pointers can be used conveniently with the support of InterWeave's global address space, eliminating the need for deep-copying message parameters.
4 High-Level Coherence Information

Bandwidth is a valuable resource in wide-area networks. However, distributed shared memory systems often display under-optimized bandwidth consumption by keeping shared data conservatively coherent without application level information. In the previous chapter, we described the two-way differencing technique to keep coherence update costs proportional to the size of modifications. In this chapter, we discuss techniques allowing more aggressive bandwidth optimization with high-level application-specific coherence information. We first describe relaxed coherence models in Section 4.1. Relaxed coherence models provide a method to specify temporal coherence information so that updates can be safely postponed. In Section 4.2 we describe dynamic views, which support spatial coherence specifications so that InterWeave can reduce bandwidth by limiting the coverage of updates. We discuss implications of using and customizing these coherence models in Section 4.3 and their implementation in Section 4.4. We evaluate the implementation efficiency in Section 4.5 using microbenchmarks and in Section 4.6 using real-world applications. We conclude with a brief summary in Section 4.7.
4.1 Relaxed Coherence Models

A shared memory system must provide some sort of "recency" guarantees. The sequential consistency (Section 2.1) model imposes the strictest guarantee of recency and consequently has the highest communication overhead. Various relaxed consistency models, such as Release Consistency and Entry Consistency, relax recency guarantees to the synchronization points. However, many distributed applications can often accept a more relaxed coherence requirement. These applications usually possess the knowledge to allow the underlying middle-ware system to postpone updating the locally cached state as long as they are "recent enough".

4.1.1 Relaxed Coherence Models

InterWeave applications use relaxed coherence models to specify the recency requirements when acquiring reader locks. In the InterWeave system, when writing to a segment, a process must have exclusive write access to the most recent version (we do not support branching histories). When reading a segment, however, the most recent version may not be required. We currently admit six different definitions of cache recentness (recent enough), summarized in Table 4.1.

Among the current models, Strict coherence has the most stringent recency requirement, and is exclusive with any concurrent writer. A client using Null coherence has no recency requirement and will succeed immediately on read lock acquires. Full coherence always requires the most recent update available at the server, but allows the segment being locked for write by another client simultaneously.

Other models can be parameterized with runtime information. The client library uses these parameters to decide if the current local segment copy is "recent enough". If not, it obtains an update from the server. The server collects a
<table>
<thead>
<tr>
<th>Coherence Model</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strict</td>
<td>None</td>
<td>Requires the most recent version and excludes any concurrent writer</td>
</tr>
<tr>
<td>Full</td>
<td>None</td>
<td>Requires the most recent version but does not exclude the potential writer</td>
</tr>
<tr>
<td>Temporal</td>
<td>$x$ seconds</td>
<td>Accepts the local cache if the segment has been updated within last $x$ seconds</td>
</tr>
<tr>
<td>Delta</td>
<td>$x$</td>
<td>Accepts the local cache if it is within $x$ versions of the server’s master copy</td>
</tr>
<tr>
<td>Diff</td>
<td>$x%$</td>
<td>Accepts the local cache if it has less than $x%$ difference from server’s master copy</td>
</tr>
<tr>
<td>NULL</td>
<td>None</td>
<td>Always accept the local copy</td>
</tr>
</tbody>
</table>

Table 4.1: InterWeave Relaxed Coherence Models

wire-format diff for the data newer than the current client version, just like that described in the previous chapter.

Among these models, *Temporal* coherence uses the timestamp of the most recent update at the server to decide recentness of the local cache recentness.*Delta* coherence uses the differences between local cache’s version and the server’s master version. In *Diff* coherence, the local cache is “recent enough” if there is less than a user-specified percentage difference with the server’s master copy.

Unless otherwise specified, newly-created segments employ *Full* coherence. The creator of a segment can specify an alternative default if desired. An individual process may also establish its own default for its own lock operations, and may override this default for individual critical sections. For example,

```c
IW_handle_t sh = IW_open_segment(seg_url, is_open);

... ...

IW_coherence_model_t cm;
```
\[ \ldots \ldots \]

```c
cm.model = IW_COHERENCE_NULL;
/* change coherence model */
IW_use_coherence(sh, &cm);
\[ \ldots \ldots \]
IW_rl_acquire(sh);
```

The new coherence model takes effect at the next reader lock acquisition.

### 4.1.2 Consistency

Without additional mechanisms, in the face of multi-version relaxed coherence, the versions of segments currently visible to a process might not be mutually consistent. Specifically, let \( A_j \) refer to version \( j \) of segment \( A \). If \( B_k \) was created using information found in \( A_j \), then previous versions of \( A \) are causally incompatible with \( B_k \); a process that wants to use \( B_k \) (and that wants to respect causality) should invalidate any cached segment version \( A_i, i < j \).

To support this invalidation process, we would ideally like to tag each segment version, automatically, with the names of all segment versions on which it depends. Then whenever a process acquired a lock on a segment the library would check to see whether that segment depends on newer versions of any other segments currently locally cached. If so, the library would invalidate those segments. The problem with this scheme, of course, is that the number of segments in the system—and hence the size of tags—is unbounded. In Section 4.4.2 we describe a mechanism based on hashing that achieves the same effect in bounded space, at modest additional cost.

To support operations on groups of segments, we allow their locks to be acquired and released together. Locks that are acquired together are acquired in the alphabetical order of their segment addresses (server URL concatenated with
segment name) to avoid deadlock. Write locks released together make each new segment version appear to be in the logical past of the other, ensuring that a process that acquires the locks together will never obtain the new version of one without the other. To enhance the performance of the most relaxed applications, we allow an individual process to “opt out” of causality on a segment-by-segment basis.

4.2 Dynamic Views

A segment serves as the logical unit for organizing shared state. However, individual clients could use the same segment in many different ways. For large segments, always updating clients with full segment modifications can incur much unnecessary communication traffic. Views allow an InterWeave client to specify the portion of a segment in which it is currently interested, thereby avoiding the use of bandwidth for updating the rest of the segment. A view is constructed dynamically and can evolve over time.

4.2.1 Defining Views

Each InterWeave view is explicitly associated with a segment and may contain an arbitrary number of view units. Each view unit is a contiguous portion of a block. The view may be specified by a pair of MIPs that refer to its start and end, respectively, or by equivalent local pointers, if the segment is locally cached. Like using relaxed coherence models, sharers of a single segment can have different views. Where coherence models (Full, Temporal, Delta, etc.) address the temporal dimension of application-level coherence information, views address the spatial dimension. As with relaxed coherence models, it is the programmer’s responsibility to define views correctly and to touch only the data covered by the current view.
A process can create an empty view given a segment handle by calling `IW_create_a_view()` (h is a segment handle):

```c
IW_view_t v = IW_create_a_view(h);
```

Once created, the process can augment it by attaching *view units*:

```c
IW_mip_t start, end;
bool recursive;
IW_attach_view_unit(v, start, end, recursive);
```

It can also detach view units using `IW_detach_view_unit()` correspondingly.

The `recursive` parameter indicates whether or not the view unit should be recursively expanded. If the `recursive` flag is set, the runtime includes in the view all other data that are reachable by following intra-segment pointers that originate inside the view unit. (Data reachable by following a pointer chain out of the segment and back in again will not be included.) Each contiguous region of additional data becomes an additional (implicitly specified) unit in the view.

A view unit can be part of an array or multiple contiguous fields of a structure. We provide APIs for creating frequently used view structures, such as slices of multi-dimensional arrays. Such structures consist of multiple view units, and can be attached to a view as a group. Recursive views are especially convenient for pointer-rich dynamic data structures, such as a subtree rooted at a given node or a linked list starting from a header node.

After a view has attached some units, it can be activated for use by a call to `IW_activate_view()`, which transmits the view definition to the InterWeave server. At any given time, a single process can have at most one active view on a given segment. Once a process activates a view, future lock acquisitions will maintain coherence only for the portion of the segment covered by the view. As pointers inside a recursive view change, the view will be updated automatically by
the runtime. New view units, whether recursively reachable or explicitly attached, will become effective (i.e., actually cached) at the time of the next lock acquire. A view remains in effect until it is disabled using `IW_deactivate_view()`. A view that is no longer needed can be destroyed using `IW_delete_view()`.

### 4.3 High-Level Coherence Information Usage and Customization

The high-level coherence information serves as hints from an application on its acceptable limits to the staleness of locally cached state. Then the InterWeave runtime can exploit various opportunities to optimize the communication overhead of segment updates. In this section, we discuss the general guidelines concerning the uses of different coherence models and dynamic views, and their performance implications. At the end of the section, we discuss potential future work on user customizable coherence models.

#### 4.3.1 Relaxed Coherence Models

The *Null* and *Temporal* coherence models allow the client to decide the cache “recentness” locally without contacting the segment servers, and thus has the minimum runtime overhead. The former is appropriate for applications that require a one time cache update. Temporal coherence, as its name indicates, is most useful when applications refresh state periodically. For example, it can help a remote visualization client avoid unnecessary updates of the visualized state by setting the temporal coherence parameter equal to the refresh rate of the display frame buffer.

The *Full*, *Delta*, and *Diff* coherence models all require server participation and have higher overhead than the temporal coherence model (InterWeave uses an
adaptive protocol to reduce the overhead. Please see Section 4.4.1). However, they allow programmers to make trade-offs more precisely between cache recentness and performance. Full coherence is usually the default, ensuring the most recent update while avoiding needless client-server communications. If the programmer understands the shared state recentness in relation to its update frequency or the manner the content is changed, she can obtain further optimization by using Delta coherence or the Diff coherence. Delta coherence bounds the frequency with which the local cache is updated with respect to the rate the segment is written by other sharers. Diff coherence, on the other hand, bounds the cache staleness by the degree the segment content has changed so that updates can be avoided when there are very small changes in the segment.

4.3.2 Dynamic Views

Dynamic views are powerful tools when a logical data structure is shared with disparate spatial patterns by multiple distributed processes. For example, individual processes that share a two-dimensional matrix segment might only want to access some particular columns, rows, or blocks of the matrix. They can subsequently use views to specify their individual interest. Like the relaxed coherence models, views provide the InterWeave runtime with hints to optimize the communication overhead in the spatial dimension. An alternative solution to the above problem is to break the matrix segment into smaller pieces. However, this is not appropriate under the InterWeave programming model. The application will become more complex and a process will have to acquire multiple locks to have access to multiple pieces.

View users must exert discretion with pointers. In a non-recursive view, pointers are coherent with other values as long as they are in the same view. However, the datum pointed to might fall outside the view and thus might contain inconsistent values. If an application wants to chase pointers in a pointer-rich segment, it
should instead use recursive views as much as possible. The InterWeave runtime will expand the recursive view for the application automatically, although at the expense of a larger computation overhead (Section 4.5.2).

4.3.3 Mixing High-level Coherence Information

Different processes (and different fragments of code within a given process) may use different coherence models for the same segment. A process can also specify multiple views for a single segment. The underlying InterWeave runtime will keep track of the actual spatial coverage of the views. They are all entirely compatible because all user-visible shared state will be coherent and consistent, provided the processes are properly synchronized.

The InterWeave system maintains two invariants during any read/write critical sections. First, after each lock acquisition, the InterWeave runtime guarantees that the values in the segment or its current view are coherent. Second, the values are consistent with those in any other locally locked segments, by the use of the hash-based consistency mechanism. Consequently, the client local caches are always consistently updated.

4.3.4 Customizing Relaxed Coherence Models

Occasionally, applications would benefit from customizing coherence models that could be more flexible than the current InterWeave coherence models and dynamic views. Programmers could exert better control over coherence updates so that the runtime system could further optimize the communication traffic. For example, an application might only want a segment to be updated every 5 times a second or when there is more than 10% changes in the segment content. However, the customization of coherence models is not immediately possible in the current InterWeave system. Here, we briefly discuss a possible API design that would
allow programmers to define and measure their own notion of "recent enough" so that customizing coherence models would become possible.

Coherence Distance

To have a customized coherence model, the user must be able to use his own notion of whether a cache is "recent enough". Cache "recentness" can in fact be defined as the coherence distance between the local cache and segment server's master copy. The user can compute Coherence distance using segment information available at the client and server side, such as the timestamp of recent updates, how many times the segment has been updated, the modifications made at other clients, etc. As an example, the temporal coherence model can be defined using the local cache timestamp as the coherence distance.

We can design an API to let programmers to define customized coherence model by providing a coherence handler to measure the coherence distance:

```c
IW_coherence_handler_t chHandler;
...
...
IW_use_coherence(seg_handle, NULL, chHandler);
```

A segment coherence handler is an application-provided function that has access to the current coherence status of the local cache. When acquiring a lock for the application, the InterWeave runtime system calls the handler to find if the local coherence distance is large enough to require a cache update.

Measuring Coherence Distance

Coherence handlers need server-side segment information to compute the coherence distance. To have access to such information, a coherence handler can be registered either as a client-side handler or a server-side handler. The client-side handler is registered by calling `IW_register_client_handler()`.
bool coherence_handler(seg_coherence_state_t *stat) {
    
    . . . .

    return update_needed;
}

. . . .

IW_coherence_handler_t ch =

    IW_register_client_handler(h, coherence_handler).

When a coherence handler is registered for a segment, the client library will first get the segment coherence state from the segment server and pass it to the handler when acquiring locks. The coherence state passed to the handler includes various coherence information available at the server side, such as the most recent segment version, the last update time, and the percentage of modifications since the client local cache. The handler returns a boolean value to instruct the client library whether to update the local segment content.

The server-side handler is a coherence handling script passed to the segment server. The script can be based on an interpreted programming language, such as Java, so that it can be executed on servers running on different types of machines or operating systems. The handler would be allowed to access specific segment state, or even values at specific segment locations addressed by machine independent pointers. In the following example, the client registers a server-side coherence handler by calling IW_register_server_handler().

char *coherence_handler = ‘‘if . . . .’’;

. . . .

IW_coherence_handler_t ch =

    IW_register_server_handler(h, coherence_handler);
The client side approach requires extra communication time to fetch segment state from the server. Its access to segment coherence information is also more restricted than the server side approach. On the other hand, the server side approach is more flexible but adds more computation overhead on the segment server than the client side approach. Ultimately it will have negative effect on server throughput. The exact trade-off between the two approaches and their implementation is out of the scope of this work and we leave it for future work.

4.4 Implementation

Many of the meta data structures and functions devised for two-way diffing can be reused to support relaxed coherence models and dynamic views. In this section, we describe some specific implementation details in cache recency, hash-based inter-segment consistency, and view maintenance.

4.4.1 Deciding Recency

When acquiring a reader lock under a relaxed coherence model, the client has to decide whether the local cached copy is still recent enough. For some coherence models, including Temporal and Null, the recency can be resolved with local information immediately. For the Strict coherence model, the client always contacts the server to be sure that there is no exclusive writer. However, Full, Delta, and Diff models require server involvement. To avoid extra round trip message exchanges, the client sends the current cached segment version to the server to compute its recentness. Computation for the Full and Delta models is straightforward; it is just the difference from the version number of the server’s master copy.

To support Diff coherence, the server needs extra information beyond the
version numbers associated with the segment, blocks, and subblocks (see Section 3.2.2). For each client using Diff coherence, the server must track the percentage of the segment that has been modified since the last update sent to the client. To minimize the cost of this tracking, the server conservatively assumes that all updates are to independent portions of the segment. It adds the sizes of these updates into a single counter. When the counter exceeds the specified fraction of the total size of the segment (which the server also tracks), the server concludes that the client's copy is no longer recent enough.

**Adaptive protocol.** We use an adaptive protocol to further reduce the communication required by Full, Delta, and Diff coherence to determine cache recency. A client that finds that its local copy of a segment is usually recent enough will enter a mode in which it stops asking the server for updates. Specifically, every locally cached segment begins in polling mode: the client will check with the server on every read lock acquire to see if it needs an update (temporal coherence provides an exception to this rule: no poll is needed if the window has yet to close). If three successive polls fail to uncover the need for an update, the client and server will switch to notification mode. Now it is the server's responsibility to inform the client when an update is required (it need only inform it once, not after every new version is created). If three successive lock acquisition operations find notifications already waiting, the client and server will revert to polling mode. The server uses the asynchronous TCP channel to send the notification messages (Section 3.2.1).

### 4.4.2 Hash-Based Consistency

To ensure inter-segment consistency, we use a simple hash function to compress the dependence history of segments. Specifically, we tag each segment version $S_i$ with an $n$-slot vector timestamp, and choose a global hash function $h$ that maps
segment identifiers into the range $[0...n-1]$. Slot $j$ in the vector indicates the maximum, over all segments $P$ whose identifiers hash to $j$, of the most recent version of $P$ on which $S_i$ depends. When acquiring a lock on $S_i$, a process checks each of its cached segment versions $Q_k$ to see whether $k$ is less than the value in slot $h(Q)$ of $S_i$'s vector timestamp. If so, the process invalidates $Q_k$.

To support the creation of segment timestamps, each client maintains a local master timestamp. When the client acquires a lock on any segment (read or write) that forces it to obtain a new version of a segment from a server, the library updates the master timestamp with any newer values found in corresponding slots of the timestamp on the newly obtained segment version. When releasing a write lock (thereby creating a new segment version), the process increments the version number of the segment itself, updates its local timestamp to reflect that number, and attaches this new timestamp to the newly-created segment version. To handle timestamp roll-overs, we assume local clocks at servers and clients are loosely synchronized and can be used for timestamp validations.

### 4.4.3 Maintaining View Coverage

**Client Side View Management**

At the client side, a view is represented by a hash table indexed by block serial number. Each entry in the table contains a list of the view units contained in a given block. The client library uses the table to track view changes made by `IW_attach_view_unit()` and `IW_detach_view_unit()`. When the process calls `IW_activate_view()` and then attempts to lock the segment, the runtime passes the hash table or its changes in the table to the server.

So long as a client process keeps its promise to touch only the data covered by its view, the existing modification detection and wire format translation routines
in InterWeave correctly collect any changes made by the client, and pass them to the server.

**Server Side View Management**

When a server receives a view definition from a client, it creates its own hash table, indexed by block serial number, to store the list of view units. For each large block in the table it also stores a bit vector indicating which subblocks are in the view.

For recursive view units, the server traverses the segment metadata to determine the full extent (*scope*) of the view. The traversal is driven by the type descriptors already maintained for the segment. For each view unit encountered, the server searches the block’s type descriptor to find the locations of pointers. It then builds a new view unit for each (strongly typed) pointed-to datum (not the entire block containing the datum) and adds these view units to the view. The traversal procedure stops when no more view units can be added.

When a client informs the server that it has added or deleted view units, the server updates its description of the view accordingly. When a block is deleted from a segment (by any client), the server automatically removes any view units contained in that block from all known client views. (The server also informs each client of the deleted block as part of the normal update mechanism when it next updates the client’s cached version of the segment.)

With recursive view units, the scope of a view can change dynamically as pointers are reassigned. Before sending diffs to a client, any view with recursive view units needs (at least conceptually) to be re-expanded by recursively following pointers. To avoid this expensive operation, the server actually updates view scopes lazily and conservatively. Assisted by block version numbers, the server searches only subblocks that have changed since the last update sent to the client. For each pointer in such a subblock, the server adds the pointed-to view units
into the view if it is not present yet. To avoid accidentally dropping useful view units, the server conservatively keeps the old pointed-to view unit in the view. With this strategy, a client may receive some view units that should already have been dropped. Although this is harmless semantically, it may waste bandwidth. As a trade-off, we set a threshold for the number of changed versions. When the threshold is exceeded, the server traverses all views and re-builds the view scopes.

Server Side Diff Collection

A server keeps the most recent version of the segments for which it is responsible. For each modest-sized block in each segment, and for each subblock of a larger block, the server remembers the version number of the segment in which the content of the block or subblock was most recently modified. Without views, the server can compute diffs for a client using the version number of the segment cached at the client and the version number associated with the server’s master copy. The server simply updates the client with blocks or subblocks whose version is larger than the client’s version. When views are in use, the process is similar except that the server now must consider the blocks or subblocks covered by the views and the version of these blocks or subblocks cached at the client. In this case, a single version number for each client no longer suffices.

Consider a client that activates a view of some segment \( S \) at version \( V_1 \) and obtains its first update at version \( V_2 \). At this point the data in the view have been updated to version \( V_2 \), but the data outside the view have not. Now suppose the client advances to a new version \( V_3 \), and then decides to add a new view unit \( v \) into the view (\( v \) might also be added by the runtime automatically as a result of pointer change in a recursive view). Since the data \( d \) in \( v \) were not in the view before, the client-cached copy of \( d \) is at version \( V_1 \). If the server only maintains a single version number for the view, it will be unaware that \( d \) missed the updates between version \( V_1 \) and \( V_2 \). Simply put, while the server will know that a newly
added view unit needs to be brought up to date, it won't know how out-of-date
that view unit was before. Similar problems exist when dropped view units are
added back into a view.

To address these problems, the server maintains some additional information
for each client that has activated a view: (a) a pre-view version—the segment
version number when the view was activated; (b) a view version—the segment
version number when the client was most recently updated; and (c) a view version
table—a hash table that tracks the version of each client-cached view unit, even
if the view unit has been detached.

![Diagram of view units and versions]

Figure 4.1: An InterWeave server collects a diff to update a client with an active
view. Here blk#4 and blk#6 were added to the view after the last update to the
client. blk#4 was once in the view but was dropped after the client updated its
version to v8. Since then blk#4 has not been modified. blk#6 was not in the view
before. Using its view version, pre-view version, and view version table, the server
constructs a diff containing the necessary updates for blk#1 and blk#6, but not
blk#4.

To collect a diff for a view unit covered by the current active view, the server
must know which version of the view unit is cached by the client. For views whose
scope has not changed since the last update to the client, the server knows that the
client-cached view units have been updated to view version. This is the common
case, and is handled efficiently by InterWeave. For view units added to the view
since the last update, there are two cases. In one case, the corresponding view
units are found in the view version table, so the server knows exactly which version
is cached by the client. In the other case, the server can infer that the client’s
copy must be the pre-view version. Once the server determines a version for the
client-cached copy, it computes the difference between the client’s version and the
current version, using the normal diff collection process already implemented in
InterWeave. The server sends the diff to the client and updates the view version
table to reflect the new version cached at the client. Figure 4.1 gives an example
of the process described above.

While data not in the current view are not updated when acquiring a lock, the
server must still inform the client of any changes to segment metadata, including:
(a) added or deleted type descriptors; (b) the serial number of added or deleted
blocks; and (c) the serial numbers of type descriptors of added blocks. This
information is needed for management of segment memory space, and for swizzling
pointers in a view that points to data outside the view. In the original InterWeave
implementation, the data and metadata of a newly created block were sent to the
client together. The ability to send the metadata by itself was the only change to
InterWeave’s wire format required to accommodate views.

4.5 Micro-benchmark Evaluation

In this section, we use microbenchmarks to evaluate the potential benefit and
implementation overhead of the relaxed coherence models and dynamic views.
4.5.1 Relaxed Coherence Models

We first demonstrate how relaxed coherence models help increase critical section scalability, especially during reader lock sections. In these experiments, we run an InterWeave server on a Sun Ultra 5 workstation and all clients on different processors of the AlphaServer 4100 5/600 cluster (Section 3.3.1). Communication is via a 100Mbps Ethernet.

Figure 4.2 shows the time per write lock acquire/release as the number of clients of a segment increases. In the lower curve only one word of data is actually modified; in the upper curve each client modifies every element of a 1024-word integer array. Because writer critical sections are serialized, the server is able to accommodate additional clients at little additional cost, and the time per critical section (calculated as the total execution time divided by the total number of acquires performed) stays relatively constant. The jump in time from 1 to 2 clients reflects the loss of exclusive-owner lock mode, which avoids communication with the server in the absence of sharing.

![Figure 4.2: Sequential cost of acquiring a writer lock.](image)
Figure 4.3 shows the time per read lock acquire assuming Full coherence. In this experiment, a single client acquires a write lock and modifies 1, 10, or 1024 integers. All clients then acquire a read lock on the data. The time per reader critical section (calculated as the total execution time, minus the time spent in extraneous code, divided by the number of read lock acquires per client) increases with the number of clients, reflecting serialization of service to clients by the (uniprocessor) server. The use of multicast, where available, might produce a flatter curve.

![Diagram showing the relationship between number of clients and total critical section time for different integer updates.](image)

Figure 4.3: Cost for a given number of clients to acquire a read lock concurrently.

Figure 4.4 demonstrates the effectiveness of using Diff coherence with 16 clients. Here, after each update all 16 clients acquire a read lock under Diff coherence, specifying different thresholds for tolerable percentage of data changed. The y-axis represents the corresponding time to acquire and release each read lock, calculated in the same manner as for Figure 4.3. The lower three curves show significant reductions in communication overhead and, consequently, average acquire/release overhead.

The differences among these last three curves illustrate the benefit of the adap-
Figure 4.4: Comparison of Full and Diff Coherence Models.

tive protocol described in Section 4.4.1. When the ratio of read acquires that do not require data communication to write acquires is $\geq 1$, the use of notification mode avoids the poll messages that would otherwise be sent to determine if the segment needs to be updated at the client. When the ratio of read acquires that do not require data communication to write acquires is $< 1$, it is desirable to switch back to polling mode, especially with a large number of clients, in order to avoid extra messages and unnecessary overhead at the server. With Diff coherence, as the diff parameter is increased, the number of read lock acquires that result in no data being transmitted between server and client increases. Dynamically switching among the polling and notification allows a general-purpose protocol to closely approximate the performance of pure notification. In this experiment there is no measurable difference among polling, notification, and adaptive variants of Full coherence; they are all represented by the “Full-adaptive” curve.
4.5.2 Dynamic Views

Views require much more meta-data management than relaxed coherence models. In all these experiments, all clients and servers run on 2GHz Linux 7.2 PCs with 512 MB of memory. To factor in the network bandwidth, we experiment with two different connections between the reader processes and the InterWeave server, i.e. 100Mbps or 10Mbps Ethernet.

Non-recursive Views

The first experiment compares the time required for an InterWeave client to receive updates with or without non-recursive views. We arrange for two InterWeave client processes to share a segment consisting of 1000 blocks. Each block is a 160-element integer array. One process functions as a writer and updates every integer in the segment. The second process functions as a reader and uses Full coherence to read the segment after each update. We measure the latency for the reader process to acquire a reader lock. The latency is broken down into the communication time to transmit the client request and server update, the diff construction time on the server, and the translation time to apply the diff on the client.

The results are shown in Figure 4.5. As can be seen from the figure, the communication time is significantly reduced due to the reduction in traffic by using views. The absolute reduction is more dramatic with the 10Mbps network (note the different scales on the two y axes). “Subblk. View” has slightly higher computation and communication overhead than “Blk. View” for both the client and server. On the server, when an entire block is in a view, the server only needs to collect changes in the block. When only a portion of a block is in the view, the server has to perform extra work to locate the portion of the block that is covered by the view. The resulting diff is also larger because more blocks, and
Figure 4.5: The effect of using non-recursive views. (a) 100Mbps network. (b) 10Mbps network. The x axis shows the increasing coverage of the view from 10% to 100%. The left-most bar in each graph is the baseline performance when no view is used. For cases involving views, we vary the scope of the view from a coverage of 10% to 100% of the segment. For each view coverage, the left bar ("Blk. View") uses a view consisting of x% of all blocks; the right bar ("Subblk. View") includes x% contiguous region of each block in the view.
thus more metadata, are in the diff. The larger diffs, increased metadata, and scattered changes increase the client's translation cost correspondingly.

We plot the communication traffic (bytes transferred) for the "Blk. View" and "Subblk. View" in Figure 4.6. The bandwidth consumption is directly proportional to the percentage of total data contained in the current view.

![Diagram](image)

Figure 4.6: Communication traffic for "Blk. View" and "Subblk. View", normalized to the communication traffic without views.

**Recursive Views**

Our second experiment measures the time required for the server to maintain views with recursive view units. We arrange for one writer and one reader to share a segment that contains 50 doubly linked lists with header nodes. Besides pointers to the previous and next items in the list, each item contains 16 integers as a payload. We start with each list containing 1,000 items. The writer inserts 100 items at the end of each list and the reader locks the segment to get the updates. The reader includes a list into its view by adding the header of the list as a recursive view unit. Thus the items inserted into those lists will be
automatically added to the view by InterWeave. We vary the coverage of views from 10% to all of the 50 lists. Again, we conduct experiments on both 100Mbps and 10Mbps Ethernet.

Figure 4.7: Communication traffic with recursive views, normalized to the communication traffic without views.

Figure 4.7 shows the reduction in communication traffic with recursive views. As described in Section 4.4.3, the InterWeave server always updates the client with full segment metadata. Because some of this metadata corresponds to data not included in the view, communication traffic is higher than that in Figure 4.6.

In Figure 4.8, we compare the latency of client updates with and without views, using networks with different bandwidth. The latency breakdown in this figure includes four items: the client application cost (Client Appl.), the differencing cost for views at both the client and server side (View), the cost of recursively computing the view scope on the server (Server Recur. View), and the communication cost.

Unsurprisingly, there is a higher overhead associated with recursive views. This cost grows linearly as the view coverage increases. Combined together, the lower bandwidth reduction (see Figure 4.7) and the larger overhead of view main-
Figure 4.8: The effect of using recursive views. (a) 100Mbps network. (b) 10Mbps network. The x axis shows the increasing coverage of the view from 10% to 100%. The left-most bar in each graph is the baseline performance when no view is used. For each view coverage, a total of x% blocks are included after the view is recursively expanded.
tenance actually cause the performance of view coverage over 60% to become worse than that without views for the fast 100Mbps network. However, with a slower 10Mbps network, using views continues to be beneficial until it reaches 100% coverage. We expect Internet applications to benefit from views in InterWeave in most cases since the network conditions (e.g., bandwidth and latency) in the Internet are typically worse than that of a congestion-free 10Mbps local-area network.

4.6 Application Evaluation

We have applied InterWeave relaxed coherence models and dynamic views on a variety of applications. Three of them will be discussed in this section, including an incremental and interactive datamining application, a web proxy caching protocol, and an intelligent environment application. We examine their high-level sharing patterns and how the information can be translated into a performance advantage on the InterWeave system.

4.6.1 Incremental and Interactive Datamining

The data mining [PZOD99] application we used performs incremental sequence mining on a remotely located database of transactions (e.g., retail purchases). Each transaction in the database (not to be confused with transactions on the database) comprises a set of items, such as goods that were purchased together. Transactions are ordered with respect to each other in time. The goal is to find sequences of items that are commonly purchased by each customer in sequential order over time.

In our experimental setup, the database server (itself an InterWeave client) reads from an active database whose content continues to grow. As updates arrive
the server incrementally maintains a summary data structure (a lattice of item sequences) that is used by mining queries. Each node in the lattice represents a sequence that has been found with a frequency above a specified threshold. The lattice is represented by a single InterWeave segment; each node is a block in that segment. A sample lattice is shown in Figure 4.11. Each data mining client, representing a distributed, interactive interface to the mining system, is also an InterWeave client. It executes a loop containing a reader critical section in which it performs queries that need search over the summary structure.

Our sample database is generated by tools from IBM research [SA94]. It includes 100,000 customers and 1000 different items, with an average of 1.25 transactions per customers and a total of 5000 item sequence patterns of average length 4. The database size is 20 MBytes. The summary structure is initially generated using half this database. The server then repeatedly updates the structure using an additional 1% of the database each time.

Relaxed Coherence Models

We first consider the scenario where users perform simple queries. A simple query is to find a set of sequences that are supported by more than a certain percentage of customers in the database. The result can be obtained by a breadth-first traversal over the structure.

Because the summary structure is large, and changes slowly over time, it makes sense for each client to keep a local cached copy of the structure and to update only the modified data as the database evolves. Moreover, since the data in the summary are statistical in nature, their values change slowly over time, and clients do not need to see each incremental change. Delta or Diff coherence will suffice, and can dramatically reduce communication overhead. To illustrate these effects, we measure the network bandwidth required by each client for summary data.
structure updates as the database grows and the database server finds additional sequences.

In Figure 4.9, the upper curve represents the network bandwidth required to send a complete copy of the summary data structure to one client after every incremental update (the experiment was set up to ensure a read lock acquire after every modification). The lower curve represents the bandwidth to send only diffs. The average savings over time is about 78%.

![Bandwidth required per read lock acquire](image)

Figure 4.9: Bandwidth required per read lock acquire.

Figure 4.10 shows the total bandwidth requirement as the client relaxes its coherence model. The leftmost bar represents the bandwidth requirement if the client uses a full coherence model, corresponding to the integral of the lower curve in the previous graph. The other four bars show the bandwidth requirements if the client uses Diff and Delta coherence with different thresholds. Using Diff coherence with a threshold of 20% changes in the summary structure, we see an additional savings of almost 75%.
Figure 4.10: Bandwidth required under different coherence models. Diff-10 uses the Diff coherence model and sets the diff parameter to 10%; Diff-20 sets the parameter to 20%. Delta-2 uses the Delta coherence model and sets the version delta parameter to 2; Delta-4 sets the parameter to 4.

Recursive Views

In the next experiment, the mining clients execute including queries over the summary structure, returning all sequences containing the query items. For example, in the example shown in Figure 4.11, an including query concerning item $B$ will return the nodes highlighted in the figure (i.e., $B$, $AB$, $B\rightarrow B$ and $A\rightarrow AB$). Because each sequence node contains pointers to every node for which it is a subsequence, we can process an including query by starting from the nodes that have items that are required and then traversing the descendants of those nodes. If a client process is only interested in including queries on a certain set of items, it can save communication bandwidth by updating only the substructure rooted at those items. For our tests we selected four items for which including queries would produce a relatively large number of sequences from the lattice.

Figure 4.12 compares the average update latency and bandwidth consumption
Figure 4.11: Including queries on a summary structure. Each node represents a meaningful data mining sequence. Each node has pointers to all other nodes for which it is a subsequence. Here the result for an including query concerning item B can be obtained by traversing the substructure rooted at node B.

seen at the client side using different view configurations. "View X" means X of our four selected items are added as recursive view units to the view. When the client acquires a lock on the segment, only the substructures rooted at those items are updated. Three different metrics are measured: update latency, update traffic, and the number of nodes updated. The figure demonstrates that recursive views tremendously reduce the update traffic and latency.

With fewer nodes to update, the server and the client also reduce the processing time on generating diffs. Figure 4.13 shows the time spent on constructing diffs and maintaining recursive views on the server. Recursive view maintenance overhead increases as the number of nodes added to the view increases. However, this overhead is adequately compensated for by the reduction in diff construction time, not to mention the reduced communication time due to lower traffic.
Figure 4.12: Update latency with views normalized to the cost without views in the datamining application.

Figure 4.13: Server side overhead for constructing wire-format diffs for updates in the datamining application.
4.6.2 Scalable Sharing of Metadata in ICP

With the rapid growth of the Internet, hierarchical and cooperative proxy caching proves to be effective in reducing bandwidth and improving client side HTTP access latency [FCAB00, CDN+96]. The Internet Caching Protocol (ICP), designed by the Harvard Harvest group [CDN+96], is perhaps the most popular sharing protocol, and is widely deployed. In ICP, cooperative proxies are organized into a hierarchical structure. Each proxy can have parents and siblings, all of which are referred to as peer proxies in this paper. When a proxy misses in its own local cache, it probes its peers for possible remote hits. If all of its peers miss, the proxy must go to the original web server on its own or ask its parent to fetch the object.

Unfortunately, ICP does not scale well as the size of caches or the number of cooperating proxies increases. On a local miss, the proxy sends query messages to its peers asking for the missed object. The number of these messages is quadratic in the number of peer proxies. To solve this problem, Summary Cache [FCAB00] and Cache Digest [RW98] have independently proposed similar solutions, in which each proxy keeps a compact directory of URLs recently cached at every other proxy. Now when a proxy misses in its local cache, it consults its peer directories before sending out ICP query messages. Queries are sent only to those proxies indicated by the directory as having a recent version of the page (URL). Both solutions use Bloom filters [Blo70] to represent the peer directories. A Bloom filter is a succinct randomized data structure that supports efficient membership queries.

There is a basic trade-off in the implementation of directories: frequent updates consume communication bandwidth, while outdated directories may introduce both false hits and false misses. False hits occur when the directory falsely indicates that a URL exists in another proxy’s cache. False misses occur when the directory does not list any proxy as containing the URL in its cache, even though
the URL is actually cached at a peer. Use of a Bloom filter also entails a small but controllable false hit rate [Blo70, FCAB00].

Summary Cache [FCAB00] proposes a broadcasting update scheme. Each proxy broadcasts to its peers an update of its directory after a fixed percentage of changes have occurred (for example, broadcasting every 1% of its local changes). To reduce the size of a broadcast message, each time only the difference since the last update is broadcast. Cache Digest [RW98] uses an on-demand update scheme. Each proxy decides how often it needs an update from other proxies. A proxy can piggy-back update requests on query messages to its peers. Likewise, a proxy can inform its peers that a new directory update is available by piggy-backing the information on its responses to query messages.

Neither of the above schemes is ideal. Broadcasting requires that all peers be updated at a fixed rate. If one proxy requires more accurate information, each of its peers must also receive more frequent updates. More importantly, efficient and robust broadcasting support is usually not available in wide-area networks. While update on demand is more flexible, it consumes more bandwidth than broadcasting because it always transmits the entire directory (a Bloom filter) on each update. Since the Bloom filter is a randomized data structure, it is difficult to be compressed [Mit01].

To evaluate the overhead incurred by updating peer directories on demand, we conduct an experiment on an ICP simulator, proxycizer, from the Crispy Squid Group at Duke University [Dep]. Since the original proxycizer implements neither Summary Cache nor Cache Digest, we augmented it with a directory implementation based on Cache Digest [RW98]. We use a trace file from the IRCache organization [IRC]. The trace records one day of requests to one of its proxies at Pittsburgh, Pennsylvania. The trace comprises 1.4 million HTTP requests with an average object size of 11.4KB. We first run the trace through proxycizer's ICP simulator with 4 simulated proxies, each with a disk cache of about 1.2GB (this
allows space for approximately 100,000 objects). The trace is fed to the proxies in a round-robin manner. Table 4.2 summarizes the results.

<table>
<thead>
<tr>
<th>Number of requests</th>
<th>1295000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of ICP query messages</td>
<td>2187207</td>
</tr>
<tr>
<td>Hits in local cache</td>
<td>28.4%</td>
</tr>
<tr>
<td>Hits in remote peer cache</td>
<td>9.7%</td>
</tr>
<tr>
<td>Aggregate size of remote hit objects</td>
<td>1860.89MB</td>
</tr>
</tbody>
</table>

Table 4.2: Simulation Results for ICP.

Each proxy caches 3 Bloom filters to summarize the contents cached at each of its peer proxies. Each Bloom filter is 200KB long. ¹ For each of its cached Bloom filters, a proxy requires an update once the proxy that is modeled by the Bloom filter has changed its content beyond a certain threshold. Figure 4.14 shows the number of query messages and the total communication traffic required to update the directories as we vary the update threshold from 0.1% to 2%. In this figure, we also show the percentage of remote hits lost due to the imprecise information in the directories.

The figure shows that using cached directories indeed significantly reduces the number of ICP query messages. However, the proxies voraciously consume bandwidth to update the directories. At the update threshold of 0.6%, the communication traffic to update the directories equals almost half the traffic for transmitting the objects themselves from hits in remote peers. The directory update traffic can be reduced by increasing the update threshold, with the trade-off of increasing lost remote hits.

We propose using InterWeave to automate the sharing of the directories among peer proxies. We can then reduce the traffic to update directories using Inter-

¹We use a Bloom filter longer than that specified in [RW98] in order to achieve a reasonable remote cache hit rate.
Figure 4.14: Using directories to reduce ICP query messages. The number of ICP queries is normalized to that when a directory is not used (i.e., standard ICP protocol). The directory update traffic is normalized to the traffic required to transmit the objects as a result of remote hits. The value of lost remote hits is normalized to the number of cache hits when using the standard ICP protocol.
Weave's relaxed coherence models and differencing. To evaluate this idea, we add
an InterWeave simulator into the modified proxycizer. In the simulator, each proxy
stores its own directory in a segment shared by other peer proxies. Each proxy up-
dates its directory segment whenever 0.1% of its local disk cache changes. Other
proxies access the directory by acquiring a reader lock on the segment.

![Bar chart](image)

**Figure 4.15:** Using InterWeave segments to share peer directories. The $x$ axis
represents the parameter for the *Delta* coherence model. The directory update
traffic is normalized to the traffic required transmit remote hit objects. The value
of lost remote hits is normalized to the number of cache hits when using the
standard ICP protocol. Note that the scale of the $y$ axis is one order less than
that in Figure 4.14.

We again run the previously described experiments, this time on the InterWeave-
augmented simulator. To control the update frequency of cached segments of peer
directories, we use the *Delta* coherence model with parameter $x$ ranging from 1 to
20. These cause InterWeave to update the client's local segment cache whenever
it is $x$ versions older than the server's master version. Since each proxy updates
its own directory segment when 0.1% of the cache changes (and thus creates a
new version at that time), the parameters we used effectively correspond to the previously used update thresholds of 0.1% to 2%. The results are shown in Figure 4.15.

Comparing the InterWeave results with those in Figure 4.14, we see that InterWeave significantly reduces the communication traffic required to keep the peer directories up to date. Each time a proxy locks a peer directory segment for update, the segment server computes and transmits a diff capturing the difference between the directory's newest version and the proxy's cached version.

Although Summary Cache [FCAB00] also broadcasts the differences between consecutive versions to reduce message size (which has not been compared here), InterWeave provides a much more flexible method for doing so without requiring complex coherence management code within the application. With InterWeave, the application (and in fact, each proxy) can change the update frequency simply by tuning the parameter for Delta coherence. In addition, cooperating peer proxies no longer have to communicate in lock step in order to coordinate with each other to update their directories.

In Figure 4.15, the choice of Delta coherence parameter exhibits a correlation with the rate of lost remote cache hits. As we relaxed the coherence models, the locally cached directories for remote proxies are less often updated. The local proxy will miss more URLs that are recently cached at remote sites because they are not in the outdated local directories. In Figure 4.15, this is represented by higher remote cache hit rates. Each proxy can make individual decisions about how often it wants to get updates for each of its peers’ directories by choosing appropriate coherence parameters, keeping the lost remote hit rate at a satisfactory level.
Figure 4.16: Finding and interpreting events in a shared image cube. The application samples images in the shared cube to detect interesting events. Later it examines the subcube containing that event.

4.6.3 Views in an Intelligent Environment Application

In Chapter 1, we described a distributed object discovery application in an intelligent environment [SNS02a, SNS02b], where multiple nodes share their "image cubes". Each image cube is an array of images captured recently by the node's camera. As explained in Section 1.2, different nodes at different times access different portions of the cubes. Straightforward sharing of these cubes using InterWeave segments would be very inefficient, wasting large amounts of communication bandwidth if a node only wants to access a small portion of the cube. Similarly, splitting up the cube into multiple segments is cumbersome and difficult, especially given that the portion of the cube accessed by any node changes over time. In the following, we describe how we solve the problem using InterWeave views.

Since a full-fledged intelligent environment is still under development, we use an application kernel to evaluate InterWeave. As shown in Figure 4.16, each
camera node collects an image cube and stores it in an InterWeave segment shared by other nodes. Each image is stored as a separate block. An application running on a remote machine samples the images and executes a series of operations to find and analyze events in the cube. It first looks for events in the cube starting from the first image and coarsely sampling images through time. Once evidence of an event is detected in a sample image \( t_0 \), the application locates a minimal rectangular region that contains the event. To verify and interpret the event, it accesses the same region within \( d \) time steps before and after the event—the sub-cube from \( t_0 - d \) to \( t_0 + (d - 1) \).

Without using views, whenever a remote image is accessed, the entire image must be brought in, even though only a small part of it will ever be examined. Our solution is to use views to specify the portion of the image that will be scanned for events. Once an event is located within a sample image, we add the surrounding sub-cube to the view to more closely examine the evidence for the event.

Figure 4.17 shows the reduction in communication traffic achieved by using views. In this experiment, the image cube contains 100 recently collected images, each of size \( 320 \times 240 \) pixels. The cube is sampled at two different rates, every 5th image (“Sample 5”) or every 10th image (“Sample 10”). We assume the discovery of an event is in the middle of the sampling process (i.e. at the 50th of 100 images in the cube). The application then examines the remote sub-cube centered around the event point. The cross section area of the sub-cube is \( 80 \times 60 \) pixels and the length of the sub-cube varies from 10 to 90 images as indicated by the \( x \) axis in the figure.

As we can see from this figure, the communication traffic is significantly reduced. While the traffic increases as the length of the sub-cube increases for both “Sample 10” and “Sample 5”, a closer comparison reveals that the former increases slightly faster than the latter. This is because the smaller sampling interval causes more images to be accessed and hence cached by the application. When an event
Figure 4.17: Bandwidth consumption with views under different sampling intervals, normalized to the bandwidth consumption without views. The $x$ axis indicates the number of images intersected by the sub-cube.

is detected, the application needs a smaller amount of update data to construct the sub-cube containing the event.

### 4.7 Summary

In this Chapter, we have studied the effect of using high-level application specific coherence information to optimize shared state efficiency [CTC+02, CTS+03]. Specifically, we have discussed and evaluated the following optimization techniques.

- We described a set of relaxed coherence models to specify temporal coherence information. We evaluated an implementation of relaxed coherence models on the InterWeave system and found that the system communication overhead can be greatly reduced by postponing unnecessary update traffic. In
a distributed data mining service, using a *diff* relaxed coherence model can save communication traffic by 75%.

- Second, we described InterWeave *dynamic views* to specify spatial coherence information. We discussed a runtime implementation that automatically track the scope of user-specified views. On the same data mining service mentioned above, our experimentation shows that using views can further reduce communication traffic by more than 80%, without any loss of service quality.

- We have discussed some implementation optimization techniques and evaluated their benefit. We use an adaptive protocol to determine local cache recency to reduce communication overhead when acquiring locks. The protocol switches between a client-polling and a server-notification mode based on past segment update frequencies. We use a vector-based technique to ensure inter-segment consistency. Here, hashing is employed to allow the technique to scale to large numbers of related segments.

In comparison with using a message-passing based programming system, using InterWeave to optimize distributed shared state is more "friendly" to programmers: it allows programmers to reason at the level of application synchronization and logical data structures, and thus relieves them from mulling over the lower level details of messaging protocols and data representation.
5 Three-Level Shared Memory Structure

The InterWeave multi-level structure addresses the problem of seamless coordination among application components on tightly-coupled locally available software distributed shared memory (S-DSM) clusters, and remotely distributed satellites. We use the term "level-1 shared memory" for hardware supported shared memory. Examples of level-1 systems include popular symmetric multiprocessor machines (SMPs) that have multiple CPUs inside a single computing node. A level-2 shared memory system is a typical S-DSM system on a cluster of SMPs, where a cluster-wide address space is supported entirely in software. A 2-level S-DSM system such as Cashmere-2L [SDH+97] leverages the level-1 shared memory support and can achieve a competitive ratio of performance/price. However, such systems are not readily extensible to coordination over the Internet. Except in rare instances, S-DSMs do not and need not handle hardware heterogeneity. To exchange shared state with remote processes, programmers commonly resort to some Remote Procedure Call (RPC) or Remote Method Invocation based (RMI) middleware solutions, such as RPC [BN84], CORBA [Obj96], .NET [Cor02], and Java RMI [Sun98].

The 3-level system allows an application to map InterWeave shared state directly onto the S-DSM's 2-level shared memory region. Thus the application can leverage the level-2 system to exploit parallelism on the cluster. Interaction with
remote processors is realized through appropriate synchronization of mapped InterWeave data structures. The InterWeave runtime system automatically takes care of data translation, caching, and coherence maintenance. Correspondingly, we use the term "level-3" to denote the InterWeave shared state.

In this Chapter, we use the Cashmere system as the experimental level-2 S-DSM. As described in Section 2.2.3, Cashmere integrates intra-SMP hardware cache coherence with cluster-level VM-based lazy release consistency. In particular, it employs two-way diffing for processes to avoid the need for TLB shutdown when synchronizing across nodes [SDH+97], and relies on low-latency user-level messages for efficient synchronization, directory management, and write-notice propagation [SDK+00]. Although the use of Cashmere limits the generality of our study, we believe many of our results are applicable to other S-DSM systems.

In the rest of this Chapter, we first discuss programming issues on the InterWeave 3-level system. Implementation over the Cashmere system is described in Section 5.2. We evaluate the implementation on a SPLASH-2 benchmark application and the Astroflow application [FDBH] in Section 5.3.

5.1 InterWeave Programming on the Cashmere Cluster

InterWeave programming on Cashmere is straight-forward. Making a Cashmere application accessible to the level-3 state normally requires only a small amount of code changes. Mostly, it involves level-3 data allocation and synchronization. Finally, the application has to be linked with an InterWeave-enabled Cashmere library.
5.1.1 Accessing InterWeave Segments on Cashmere

Like other Cashmere applications, an InterWeave application spawns across the cluster as many processes as necessary to exploit intra-cluster parallelism. Any one of these processes can call \texttt{IW.open.segment()} to gain access to an InterWeave segment. However, they all appear as one \textit{single InterWeave client} to the segment server. This important design choice allows us to exploit Cashmere shared memory support and simplifies the implementation. The single client image is achieved by maintaining a cluster-wide InterWeave meta data structure.

In order to take advantage of the existing high-performance Cashmere implementation, the InterWeave client library uses the Cashmere shared memory region to allocate both meta data and shared data structures. For example, when a process uses \texttt{IW.mip.to.ptr()} to obtain a pointer to a segment block, the returned pointer will point to a level-2 Cashmere shared memory region and is valid cluster wide. A process can also use \texttt{IW.malloc()} to allocate new blocks in a segment, assuming that it has acquired the writer lock for the segment.

5.1.2 Synchronization

As with any level-3 application, a process on a Cashmere cluster uses InterWeave locks to synchronize with other level-3 sharers. However, because of the single-client semantics, each process inside the cluster behaves like a thread in a multi-threaded InterWeave application. The client library maintains an internal counter to track the number of lock holders. If the counter is 0 when a process acquires a lock, the cluster contacts the segment server to acquire the lock by calling \texttt{IW.xx.acquire}. Otherwise, InterWeave simply increments the lock counter and returns to the requester. When releasing a lock, InterWeave decrements the lock counter; if the counter reaches 0, the segment server is contacted to release the
lock (IW_r1_release or IW_w1_release()). ¹

Multiple processes on the Cashmere cluster can acquire a segment writer lock simultaneously under the single-client semantics. This is slightly different from the pure level-3 exclusive writer lock we have discussed in the previous chapters. However, having multiple writers is necessary to fully leverage the high computing performance on the Cashmere cluster.

In the following example, a Cashmere process first opens an InterWeave segment. It then locks the segment for write and allocates a new block inside the segment.

```c
h = IW_open_segment(segurl);

.......

double *bar = (double *)IW_mip_to_ptr(bar_mip);
    // get the local pointer

.......

IW_wl_acquire(h); // level-3 synchronization
csm_barrier(0);   // level-2 synchronization

.......

char *p = IW_malloc(h, foo_type, ‘foo’);

...

bar[i] = ...;

.......

csm_barrier(0);  // level-2 synchronization
IW_wl_release(h); // level-3 synchronization
```

The previous example also shows that the processes must use the Cashmere (level-2) synchronization primitives to synchronize among themselves properly on

¹If the client is in the exclusive owner mode, the manager node will not send messages to the segment server
the cluster. Here, `csm_barrier()` is used to allow multiple processes to write to the segment memory simultaneously. In other words, the level-2 system is not entirely transparent to the level-3 InterWeave applications. An alternative design would require enforcing InterWeave single writer on the cluster. Since the multi-writer protocol contribute to the performance of the Cashmere system, we did not adopt this design.

5.2 3-level System Implementation on the Cashmere Cluster

The 3-level system implementation is based on the 2-level Cashmere implementation on the AlphaServer cluster. When an InterWeave client runs on this cluster, the client library uses Cashmere shared memory regions to manage and map level-3 segment meta data and shared content. One cluster node is assigned as the manager node to coordinate coherence communications with InterWeave segment servers. To minimize overhead on the level-2 system (the Cashmere system in our case), our level-3 implementation leverages the level-2 protocol to detect writes into the level-3 pages and facilitate the generation of the wire-format diffs.

In this section, we first briefly summarize the features of the Cashmere implementation. After that, we discuss specific issues of the 3-level implementation, including manager node, meta data management, and synchronization.

5.2.1 Overview of the Cashmere Implementation

The Cashmere implementation used in our experimentation is built on a cluster of Compaq AlphaServer 4100 5/600 computers [Ste99]. Each AlphaServer is an SMP machine and is equipped with four 21164A processors operating at 600 MHZ and with 2GB of shared memory. The nodes are connected by a PCI-based
Memory Channel interface, which provides a peak point to point bandwidth of 83Mbytes/sec and supports remote memory write.

Cashmere implements a modified Lazy Release Consistency (LRC) model. The writer node sends invalidation messages (called write notices) at lock release points; but the message receiver nodes delay the invalidation processing until the next lock acquire points. Like TreadMarks and Munin, twins and diffs are used to collect local modifications and allow multiple writers to a single page. Each page has a single home node, which merges modifications from other nodes into the page's master copy. Cashmere uses a two-level page directory to maintain the global page state information. The protocol takes advantage of the special features of Memory Channel for efficient intra-cluster communication and meta data management.

Cashmere introduces various techniques to optimize system efficiency. Pages at the home node are not twinned because the home node always has the master copies. A page's home node can migrate according to the application's sharing pattern. Incoming diffs are used to avoid costly TLB shutdown that is necessary for synchronizing processors on the same node. Furthermore, Cashmere allows a sole owner of a shared page to enter into an Exclusive mode to eliminate its coherence overhead on accessing the page. More details on the Cashmere design and implementation can be found in Stets et al. [Ste99].

5.2.2 InterWeave Cluster-wide Manager Node

When started, the InterWeave client library assigns one node in the cluster to be the manager node. The manager node assures the single-client semantic. All cluster communications with segment servers are through this manager node. Currently, there is only one cluster-wide manager node for all of the opened segments. Although it is not prohibitively difficult for us to assign manager nodes for indi-
individual segments, we have not encountered the need to do so now.

The manager node maintains the segment coherence information for the entire cluster. All level-3 segment synchronization and coherence operation requests within the cluster are first sent to the manager node. It maintains a segment state table recording the coherence state for each locally opened segment at every cluster node. The manager node receives updates from segment servers and updates the corresponding local segment caches mapped onto the level-2 Cashmere shared memory region. When releasing writer locks, it collects modifications and generates a diff to be sent to the server (Section 5.2.4).

The manager node uses the Cashmere messaging system to communicate with other nodes in the cluster. Intra-cluster messages are necessary for coordinating communications with segment servers. These messages include node-level lock requests and releases, high-level coherence information specifications, and segment openings and closings. Like the Cashmere messages, messages to the manager node are put into a node-level shared memory queue and can be processed by any process on the SMP node. This guarantees maximal responsiveness and parallelism from the manager.

5.2.3 InterWeave Cluster-wide Meta data

The InterWeave client library puts the level-3 meta data in the Cashmere shared memory region so that every cluster process can access it efficiently. When initiated, the client library pre-registers a region of level-2 shared memory address space for future use by the InterWeave system. We utilize an unused bit in the Cashmere *global page table* to mark the reserved InterWeave memory regions.

Out of this reserved level-2 shared memory, InterWeave allocates—from the manager node—a segment table accessible to all processes in the cluster. The table contains meta data information for each locally opened segment as described in
Section 3.2.2. The table is augmented with the per-node meta data information, for example, the per-node level-3 synchronization state. Processes need these meta data for efficient segment lookup, pointer translation, and data allocation and deallocation.

Because the segment table and the segment meta data are stored in the level-2 shared memory region, accesses to them are synchronized by level-2 locks. We use one level-2 lock for each segment to protect the table. For example, when a process calls \texttt{IW.malloc()} to allocate a block, the client library acquires the level-2 lock to gain the right to modify the segment meta data structure. After the allocation, the client library releases the lock to allow the changes to be seen by other processes. Similarly, when another process calls \texttt{IW.mip_to_ptr()}, the InterWeave library acquires the lock to access the correct segment address information.

5.2.4 Level-3 Synchronization

The level-3 lock operations coordinate cluster processes with other remote InterWeave sharers. As explained above, we designate a single \textit{manager} node within the cluster for all communications with the segment servers. Upon level-3 lock acquisition, the manager node first write-locks the segment within the cluster (with the level-2 lock) and asks the server for possible updates. After applying any updates from the server, the manager releases the level-2 lock for the segment. Because processes have to synchronize themselves within the cluster, the updated content will correctly propagate to other processors. Similarly, upon level-3 write lock release, the manager node ensures that modifications made visible by previous level-2 operations can be identified through the use of twins and diffs.

Figure 5.1 visually depicts a sequence of actions performed on the Cashmere system. The time lines in the figure flow from left to right, and represent three processors within a tightly coupled cluster. P0 is the manager node in this case.
All interactions between the level-2 system and the segment’s InterWeave server go through the manager node. The manager node interacts with the server to obtain level-3 updates and also send back modifications on the cluster back to the server. In the mean time, processes within the cluster protect their accesses to mapped segment content with level-2 synchronization operations.

![Diagram of coherence actions at levels 2 and 3.](image)

Figure 5.1: Coherence actions at levels 2 and 3.

The manager node has to generate diffs at an InterWeave release efficiently. In the pure level-3 system described in previous chapters, this is implemented by creating twins in the page fault handler and comparing them with the real page copy at the time of a write release. On the Cashmere cluster, we cannot simply harness the Cashmere page fault handler for level-3 purposes. First, only the manager node needs to do the differencing. Second, the manager node might not catch an update to a level-3 page by a Cashmere page fault handler in the following scenarios: 1) the manager node is the level-2 home node to a level-3 page and will always set the page writable; 2) the home node could collect level-2 diffs from other nodes to update its master copy. 3) the manager could receive updates for a level-3 page from its home node before writing to the itself.

Our solution for these scenarios is illustrated in the Figure 5.1. First, as illustrated on the *P0* time line, the manager node creates a twin for a page if it
experiences a write fault. If the manager is not the level-2 home node for the page, then this twin is used for both level-2 and level-3 modification detection purposes. If the manager node is the level-2 home node, then this twin is needed for level-3 only. Thus, the manager node sets all pages not-writable (either Invalid or ReadOnly) after obtaining a level-3 writer lock for the segment. We modify the Cashmere page fault handler to handle this. Second, as illustrated with page b, the manager creates a level-3 twin if it receives a write notice from another node in the cluster (P2) and must invalidate the page. Third, as illustrated with page c, the manager creates a twin for level-3 purposes (only) if it receives a level-2 diff from another node in the cluster (P1).

On a level-3 release, the manager node compares any level-3 twins to the current content of the corresponding pages in order to create diffs for the InterWeave server. Overhead is thus incurred only for those pages that are modified; in practice, the number of additional twins created is fairly low.

**Exclusive-owner Lock**

As described in Section 3.2.3, the exclusive-owner mode is an optimization that allows the sole sharer to acquire and release level-3 locks locally. It is particularly useful on the Cashmere cluster. First, the frequent writer on the cluster can eliminates level-3 communications by entering the exclusive owner mode and not contacting the segment server at all. Second, as long as the cluster is in the exclusive-owner mode, the manager node need not collect modifications from other nodes in the cluster to compute level-3 diffs. Correspondingly, the level-3 subsystem avoids the intra-cluster InterWeave messages described in the previous subsection entirely.
5.3 Evaluation

To illustrate the interaction between InterWeave shared state managed across the Internet, and software distributed shared memory running on a tightly coupled cluster, we collected performance measurements for two different applications: a remote visualization of the Splash-2 [WOT+95] Barnes-Hut simulation and the Astroflow application [DDFS00]. Both of the simulations run on the Cashmere cluster, using 4 nodes, 16 processors. The Barnes-Hut simulation repeatedly computes new positions for 16,384 bodies, while the Astroflow simulator computes on a $256 \times 256$ grid. These positions or grid values are potentially shared with a remote visualization satellite via an InterWeave segment. The simulator uses a write lock to update the shared segment. In the case of Barnes-Hut, the satellite uses a relaxed read lock with temporal coherence to obtain an effective frame rate of 15 frames per second. Under human direction, the visualization satellite can also steer the application by acquiring a write lock and changing a body's data. In the case of Astroflow, since the rate at which the data is modified is very low, the satellite uses full coherence.

5.3.1 System Efficiency

When we combine high performance level-2 shared memory (Cashmere) with level-3 shared memory (InterWeave), it would be ideal if there were no degradation in the performance of the level-2 system. To see how closely we approach this ideal, we linked the application with the InterWeave library, but ran it without connecting to a visualization satellite. Communication with the server running on another Alpha node was via TCP/IP over Fast Ethernet. Relatively little communication occurs in the absence of a satellite, due to the exclusive mode optimization described in Section 5.2.4 and Section 3.2.3.

Execution times for Barnes-Hut in the no-satellite experiment appear in Fig-
Figure 5.2. Each bar gives aggregate wall-clock time for ten iteration steps. Each pair of bars is for a different number of processors (the configuration `processors: nodes` specifies the number of processors per node and the number of nodes used, implying that `processors \times nodes` processors were used), with the bar on the left for the standard Cashmere system with no external communication and the bar on the right for Cashmere linked with the InterWeave library and communicating with a server. The right-hand bars are subdivided to identify the overhead due to running the level-3 protocol code.

![Graph showing time versus configurations](image)

**Figure 5.2:** Overhead of InterWeave library for the Barnes-Hut application, without a visualization satellite.

We also measured the simulator’s performance when communicating with a single satellite. Specifically, we compared execution times using InterWeave to those obtained by augmenting user-level code with explicit TCP/IP messages to communicate with the satellite (directly, without a server), and then running the result on the standard Cashmere system. Figure 5.3 presents the resulting execution time for Barnes-Hut. In all cases, the satellite was running on another Alpha node, communicating with the cluster and server, if any, via TCP/IP over Fast
Figure 5.3: Overhead of InterWeave library and communication during Barnes-Hut remote visualization.

Ethernet. We have again subdivided execution time, this time to separate out both communication and (for the right-hand bars) InterWeave protocol overhead and wire format translation overhead. The overhead of the InterWeave protocol itself remains relatively small. For this particular sharing scenario, much of the shared data is modified in every interval. InterWeave therefore switches, automatically, to no-diff mode to minimize the cost of tracking modifications. Meta data communication in InterWeave adds less than 1% to the total data communicated.

Figure 5.4 shows the corresponding results for the Astroflow program. The three bars show the execution time of the application running under Cashmere alone, InterWeave without a visualization satellite (representing overheads in exclusive mode), and InterWeave with a satellite. Since the modification rate for this application is low, the protocol adapts into notification mode, and the communication overhead is quite low. The perturbation in simulation time when using a remote satellite can be seen to be small.
Figure 5.4: Overhead of InterWeave library and communication with and without remote visualization for Astroflow.

5.3.2 Programming Experience

Our programming experiences with these two applications demonstrate a key advantage of using InterWeave in a multi-level shared memory system: bringing high performance by using an intuitive and easy programming interface. Both applications need minimal InterWeave related changes to bring the simulators and visualization clients together. Where code changes are necessary, they are limited to places where the processes need to protect their accesses to the shared state, by using InterWeave locks. Unlike the versions that use hand-written message passing, the InterWeave versions need not be aware of the number of satellites or the frequency of sharing.

Besides easy programming, the versions that use the InterWeave 3-level structure are also very efficient. At the visualization satellites, the InterWeave runtime system automatically caches shared simulation results and keeps the data coherent. The communication traffic is optimized by using two-way differ and application-level coherence information. At the simulator sites, InterWeave auto-
matically maps shared data structures onto the Cashmere shared memory region with little performance penalty on the computation processes, as we have seen above.

Of course, the efficiency of the InterWeave applications can be challenged by well designed and implemented applications that use message passing. However, as we have argued before, it would requires programmers to commit significant more effort in managing memory caches and optimizing communication traffic. In this scenario, the programmer must explicitly deal with the additional issues of coordinating with the Cashmere system efficiently. In comparison, InterWeave offers programmers a choice to produce efficient systems with much less effort.

5.4 Summary

The 3-level distributed shared memory structure is an important feature of InterWeave. It extends the Cashmere [SDH+97] 2-level structure to allow parallel applications running on tightly coupled clusters to coordinate seamlessly with remotely distributed satellites. In this Chapter, we have We have discussed the InterWeave programming model on the Cashmere system, with particular emphasis on the synchronization model under the level-2 Cashmere and the level-3 InterWeave system. Our experience indicates that the extra programming effort to migrate Cashmere applications to InterWeave is minimal and straightforward.

The InterWeave 3-level implementation leverages the coherence and consistency support at the Cashmere level. The level-2 shared memory region is used to hold the level-3 InterWeave shared state and meta-data structures[CDP+00, CTC+02]. It utilizes the Cashmere high-performance Memory Channel messaging system [FG97] for fast intra-cluster communications.

We have experimented with the 3-level implementation on remote scientific simulation applications. We found that InterWeave added a small performance
overhead to the level-2 parallel application, less than 1%. Communication overhead with a single remote satellite is comparable to that of a hand-coded TCP-based satellite. Moreover, the InterWeave overhead can be easily amortized over multiple satellites with no extra programming effort. Programmers can apply InterWeave high-level coherence specifications to further optimize communication traffic. We also found that the exclusive mode on the Cashmere cluster is very effective in eliminating InterWeave overhead when there are no remote sharers.
6 Java Support for InterWeave Applications

In this Chapter we describe InterWeave support for Java applications. With this support, Java programs can obtain access to the InterWeave global address space and coordinate naturally with applications written in other languages, especially those parallel applications that have been traditionally written in Fortran 77, Fortran 90, and C/C++. Java applications can automatically cache remote state and take advantage of InterWeave coherence optimizations.

We want our InterWeave support to be as non-intrusive as possible on programming practice. Java applications should be able to read/write InterWeave shared state like normal Java objects without compromising the privacy and security of the Java runtime environment. A Java object reference to an InterWeave block must be safely dereferenced. The Java virtual machine must not inadvertently garbage collect InterWeave objects or leave dangling references.

We first describe the Java programming interface for the InterWeave system in Section 6.1. Section 6.2 discusses our implementation platform and various implementation subtleties. We compare the performance of InterWeave to Java RMI in Section 6.3 and evaluate the performance of Astroflow's Java remote satellite.
6.1 InterWeave Java Programming Interface

The InterWeave Java programming interface allows Java applications to access the InterWeave shared state with ordinary object reads and writes. Because InterWeave assumes a client/server model, the interface provides a client-side Java library to access InterWeave functionalities. The InterWeave server always keeps segments in the InterWeave wire-format and is oblivious of both client architectures and programming environments. Consequently, there is no need to change the server to support a client of a specific language environment.

6.1.1 InterWeave Segment Class

The basic InterWeave functionalities described in the previous Chapters are provided to Java applications with the IWSegment class (see Figure 6.1). InterWeave's segment-wise operations (e.g. lock acquire and release) are defined as public methods. System-wide operations (e.g. segment creation or dereference of a MIP) are static member functions. All are native functions, implemented in a JNI library.

To create and initialize a segment in Java, one can execute the following calls similar to those in C programs:

```java
String url = "\.\";
IWSegment seg = new IWSegment(url);
```

Following the basic InterWeave design, the segment is the basic coherence unit in a Java InterWeave process. The application obtains a connection to an InterWeave segment by creating a corresponding IWSegment object. The segment object is a container for a collection of block objects, each of which can be of different classes. These objects are called InterWeave block objects to be distinguished from ordinary Java objects. Block objects are always associated with a
public class IWSegment {
    public IWSegment(String URL,
            Boolean iscreate);

    public native static
        int RegisterClass(Class type);

    public native static
        Object mip_to_ptr(String mip);

    public native static
        String ptr_to_mip(IWObject Object obj);

    ................

    public native int wl_acquire();

    ................

    public native int setCoherenceModel(CoherenceModel model);
    public native IWView createView();
    public native IWView addViewUnit(IWView view, IWObject obj);
    ................

}

Figure 6.1: IWSegment Class.
segment and can be individually addressed by their serial numbers or optional names.

6.1.2 Allocate InterWeave Objects

Like other Java objects, an InterWeave object can be created by invoking the `new` operator given its class and an associated segment object:

```java
seg.wl_acquire();
MyType myobj =
    new MyType(seg, blkname);
myobj.field = ... ...
seg.wl_release();
```

Like block types in the C/C++ programming interface, the Java block classes are created automatically by an InterWeave Java IDL from the corresponding InterWeave Interface Definition Language (IDL) specification. These classes must be registered by using `IWSegment.RegisterClass()` when the process starts. Each such class is derived from a root InterWeave class `IW.Object`, with a default constructor that takes two parameters, a segment object, and an optional block name string. The corresponding class object serves as the type descriptor for the block object.

6.1.3 Generating Block Classes

The Java IDL compiler must generate type descriptors without ambiguity so that the InterWeave objects can be correctly translated when communicating with the segment servers. In many cases the mapping from an IDL specification to a Java class is straightforward: an IDL struct is mapped to a Java class, primitive fields
in an IDL struct are mapped to primitive fields in a corresponding Java class, and
pointers to references.

Some data types are tricky. To be able to obtain Java references to objects
of primitive data types, we have to map these types into their corresponding
wrapped up Java classes. For example, we map int to the java.lang.Integer
class, float to the java.lang.Float class, etc.

An InterWeave array type is always associated with a length, while a Java
array class is not. Simply mapping an InterWeave array to a Java array creates
ambiguities that will make object translation difficult. Instead, we map an Inter-
Weave array type into a wrapped-up Java class. The class has a constant integer
field representing the array length and a corresponding array field. The wrapped-
up class always derives from a predefined IW.Array class, which derives from the
IW.Object class. The generated constructor initializes the array field with the
specified length when the array object is created using new.

6.1.4 Reference Translation

Programmers can use IWSegment.mip.to.ptr() and IWSegment.ptr.to.mip() to
convert between an InterWeave object reference and its corresponding machine in-
dependent address. However, we do not support certain translations that are valid
in C/C++ languages but are prohibited by Java language semantics. Examples
include obtaining a reference to a primitive field inside an object or obtaining
references into the middle of a primitive array.

6.1.5 Synchronization and High-Level Coherence Information

The IWSegment class provides functions for segment synchronizations and for high
level coherence information specifications. They are similar to those in the C lan-

guage, except for setting view units. Because of the reference translation restrictions explained in the previous subsection, we only allow specifying entire objects as a unit, and disallow units that start or end at the middle points of objects.

6.2 Implementation of Java Support

As a basic design principle, we have tried to reuse as much code as possible in our development. Consequently, we have implemented the InterWeave Java support using the InterWeave C library and used a customized Java virtual machine based on the Kaffe virtual machine. The JVM customization is necessary to map InterWeave blocks created and translated by the InterWeave C library to genuine Java objects. Kaffe is chosen because it is an open source project and its Just-In-Time (JIT) bytecode compiler provided good performance compared with other contemporary open source JVMs when we started the project [Kaf].

The InterWeave Java implementation thus comprises three interacting components: the implementation of the IWSegment class and its Java native interface (JNI) library, the Kaffe component, and the existing InterWeave C library. The IWSegment class functions as the interface between the InterWeave Java applications and the underlying Kaffe and C InterWeave library. The Kaffe component modifies the original Kaffe bytecode execution engine and garbage collector to cooperate with the InterWeave objects. The InterWeave C library implements the InterWeave functionalities.

Our decision to implement InterWeave support directly in the JVM clearly reduces the generality of our work. A more portable approach would implement InterWeave support for segment management and wire-format translation in Java libraries. This portability would come, however, at what we consider an unacceptable price in performance. Because InterWeave employs a clearly defined internal wire format and communication protocol, it is at least possible in principle for
support to be incorporated into other JVMs.

6.2.1 JNI Library for IWSegment Class

The native library for the IWSegment class serves as an intermediary between Kaffe and the C InterWeave library. Programmer-visible objects that reside within the IWSegment library are managed in such a way that they look like ordinary Java objects.

As in any JNI implementation, each native method has a corresponding C function that implements its functionality. Most of these Java functions simply translate their parameters into C format and call corresponding functions in the C InterWeave API. However, InterWeave object creation and their type translations need special explanation.

Mapping Blocks to Java Objects

Like ordinary Java objects, InterWeave objects in Java are created by “new” operators. In Kaffe, the “new” operator is implemented directly by the bytecode execution engine. It requires a class object on the parameter stack. We modified this implementation to call an internal function newBlock in the JNI library when the class object is an InterWeave type descriptor. newBlock calls the InterWeave C library to allocate an InterWeave block from the segment heap instead of the Kaffe object heap. Before returning the allocated block back to the “new” operator, newBlock initializes the block to be manipulated correctly by Kaffe.

In Kaffe, each Java object allocated from the Kaffe heap has an object header. This header contains a pointer to the object class and a pointer to its own monitor. Since C InterWeave already assumes that every block has a header (it makes no assumption about the contiguity of separate blocks), we put the Kaffe header at
the beginning of what C InterWeave considers the body of the block. A correctly initialized Java InterWeave object is shown in Figure 6.2.

![Diagram of block structure in InterWeave Java Implementation](image)

Figure 6.2: Block Structure in InterWeave Java Implementation.

After returning from `newBlock`, the Kaffe engine calls the class constructor and executes any user customized operations.

Java Class Objects

Before any use of a class in a Java InterWeave segment, including the creation of an InterWeave object of the type, the class object must be first registered with `RegisterClass`. This is because the underlying InterWeave C library is not able to use class objects directly to translate the InterWeave blocks between the local format and the wire-format. `RegisterClass` converts the class object into a type descriptor form that the library understands (Section 3.2.2) and passes it to the library.

The registered class objects and their corresponding C type descriptors are placed in a hashtable. The `newBlock` later uses this hashtable to convert a class object into a shadow C type descriptor. The class object is available on the parameter stack of the `new` operator. The C library uses the descriptor to allocate InterWeave blocks so that the library has the information to translate back and forth between local and wire format (see Chapter 3.2).
6.2.2 Kaffe Modifications

We modified the byte code interpreter and the JIT compiler in the Kaffe virtual machine to implement fine-grained write detection via instrumentation. We have also modified the garbage collector to ensure that InterWeave blocks are not accidentally collected.

Write Detection

To support diff-based transmission of InterWeave segment updates, we must identify changes made to InterWeave objects over a given span of time. The current C version of InterWeave, like most S-DSM systems, uses virtual memory traps to identify modified pages, for which it creates pristine copies (twins) that can be compared with the working copy later in order to create a diff.

Java InterWeave could use this same technique, but only on machines that implement virtual memory. To enable our code to run on handheld and embedded devices, we pursue an alternative approach, in which we instrument the interpretation of store bytecodes in the JVM and JIT with InterWeave-specific write barrier operations.

A write barrier monitors application writes to InterWeave block objects. In each Kaffe header, there is a pointer to the object method dispatch table. On most architectures, pointers are aligned on a word boundary so that the least significant bit is always zero. Thus, we use this bit as the flag for InterWeave objects.

We also place two 32-bit words just before the Kaffe object header, as shown in Figure 6.3. The second word—modification status—records which parts of the object have been modified. A block's body is logically divided into 32 parts, each of which corresponds to one bit in the modification status word. The first extended word is pre-computed when initializing an object. It is the shift value
used by the instrumented store bytecode code to quickly determine which bit in the modification status word to set (in other words, the granularity of the write detection). These two words are only needed for InterWeave blocks, and cause no extra overhead for normal Kaffe objects.

Figure 6.3: Extended Kaffe object header for fine-grained write detection.

**Integrity of InterWeave References**

The write barrier also checks the writes of Java object references to the InterWeave blocks. These operations are not allowed because InterWeave can not translate non-InterWeave objects. The references are invalid in the InterWeave global address space (represented by machine independent pointers). If the write barrier detects such a write, the Kaffe VM throws an error exception IWRefException to the running application. If the application does not catch the exception, Kaffe will subsequently abort the process.

**Garbage Collection**

Like distributed file systems and databases (and unlike persistent distributed object systems such as PerDiS [FSB+98]), InterWeave requires manual deletion of data; there is no garbage collection. Moreover, the semantics of InterWeave segments ensure that an object reference (pointer) in an InterWeave object (block) can never point to a non-InterWeave object. As a result, InterWeave objects
should never prevent the collection of unreachable Java objects. To prevent Kaffe from accidentally collecting InterWeave memory, we modify the garbage collector to traverse only the Kaffe heap.

6.2.3 InterWeave C library

The InterWeave C library needs little in the way of changes to be used by Java InterWeave. When an existing segment is mapped into local memory and its blocks are translated from wire format to local format, the library calls functions in the IWSegment native library to initialize the Kaffe object header for each block. When generating a description of modified data in the write lock release operation, the library inspects the modification bits in Kaffe headers, rather than creating diffs from the pristine and the working copies of the segment’s pages.

6.2.4 Discussion

Portability

As Java is supposed to be “Write Once, Run Anywhere”, our design choice of implementing InterWeave support at the virtual machine level can pose the concern of the portability of Java InterWeave applications. Our current implementation requires direct JVM support for the following requirements:

1. Mapping from InterWeave type descriptors to Java object classes.

2. Managing local segments and the translation between InterWeave wire format and local Java objects.

3. Supporting efficient write detection for objects in InterWeave segments.

We can use class reflection mechanisms along with pure Java libraries for InterWeave memory management and wire-format translation to meet the first two re-
requirements and implement Java InterWeave totally in pure Java. Write detection
could be solved using bytecode rewriting techniques as reported in BIT [LZ97],
but the resulting system would most likely incur significantly higher overheads
than our current implementation. We didn’t do this mainly because we wanted
to leverage the existing C version of the code and pursue better performance.

The Java memory model (JMM) [MD97] uses a weak consistent memory model
that follows Release Consistency [GLL+90] with the exception of volatile Java
objects [Lea99]. Under JMM, objects that are shared by multiple threads on a
single JVM must be correctly synchronized to allow correct concurrent program
behavior. In Java InterWeave, accesses to mapped InterWeave blocks (objects)
must also be correctly synchronized via Java object monitors and appropriate
InterWeave locks. The use of InterWeave Java objects is thus compatible with
JMM.

Support for Other Languages

As another effort to make InterWeave available to multiple languages, we have
implemented an InterWeave Fortran 77 support [TCDS02]. Unlike C/C++ and
Java, dynamic memory allocation is absent from the official Fortran 77 standards
(the Fortran 90 standards adds dynamic memory allocation support). Program-
mers declare data structures as common blocks that are statically allocated when
the process starts. InterWeave addresses the problem of mapping segments into
local Fortran 77 date structures by allowing processes to share static variables
(this mechanism also works in C).

More specifically, given an IDL file, the InterWeave IDL/Fortran compiler
generates a .f file containing common block declarations (structures in IDL are
mapped to common blocks in Fortran), a .c file containing initialized type de-
scriptors (similar to its C counterpart), and a .s file telling the linker to reserve
space for subsegments surrounding these variables. At run time, extensions to
the InterWeave API allow a process to attach variables to segments, optionally supplying the variables with block names. Once shared variables are attached, a Fortran program can access them in exactly the same way as local variables, given proper lock protections.

Fortran 77 differs from Java in many aspects. However, the fundamental challenge in implementing InterWeave support is similar: they all require a correct mapping between InterWeave type descriptors and the local language type declarations. Java has the additional challenge of being an interpreted language. Fortran has the additional challenge of lacking strong type support and dynamic memory allocation. As C/C++, Java, and Fortran 77 are all procedural languages, the InterWeave coherence and InterWeave's synchronization styles and coherence models prove to be intuitive and straightforward in our programming experience. However, we leave it as future work to move InterWeave into a broader range of languages that can include functional-based and logic-based programming languages.

6.3 Evaluation

In this section, we present performance results for the InterWeave Java implementation. All experiments employ a Java client running on a 1.7GHz Pentium-4 Linux machine with 768MB of RAM. In experiments involving data sharing, the InterWeave segment server is running on a 400MHz Sun Ultra-5 workstation.

6.3.1 Cost of Write Detection

We have used SPEC JVM98 [Sta] to quantify the performance overhead of write detection via bytecode instrumentation. Specifically, we compare the performance of benchmarks from JVM98 (medium configuration) running on top of the unmod-
ified Kaffe system to the performance obtained when all objects are treated as if they resided in an InterWeave segment. The results appear in Figures 6.4 and 6.5.

![Graph showing performance comparison between Kaffe (intrp.) and J-InterWeave (intrp.)](image)

**JVM98 Benchmarks**

Figure 6.4: Overhead of write-detect instrumentation in Kaffe’s interpreter mode.

![Graph showing performance comparison between Kaffe (JIT3) and J-InterWeave (JIT3)](image)

**JVM98 Benchmarks**

Figure 6.5: Overhead of write-detect instrumentation in Kaffe’s JIT3 mode.

Overall, the performance loss is small. In Kaffe’s interpreter mode there is less than 2% performance degradation; in JIT3 mode, the performance loss is
about 9.1%. The difference can be explained by the fact that in interpreter mode, the per-bytecode execution time is already quite high, so extra checking time has much less impact than it does in JIT3 mode.

The Kaffe JIT3 compiler does not incorporate more recent and sophisticated technologies to optimize the generated code, such as those employed in IBM Jalepeno [SOT+00] and Jackal [VHBB01] to eliminate redundant object reference and array boundary checks. We can also apply these techniques in the Java InterWeave implementation to eliminate redundant instrumentation for checking InterWeave object references. By merging unnecessary write detections to the same objects, we believe that the write detection overhead could be further reduced.

### 6.3.2 Translation Cost

As described in Sections 3.2.3, a Java application must acquire a lock on a segment before reading or writing it. The **acquire** operation will, if necessary, obtain a new version of the segment from the InterWeave server, and translate it from wire format into local Kaffe object format. Similarly, after modifying an InterWeave segment, a Java application must invoke a write lock **release** operation, which translates modified portions of objects into wire format and sends the changes back to the server.

From a high level point of view this translation resembles **object serialization**, widely used to create persistent copies of objects, and to exchange objects between Java applications on heterogeneous machines. In this subsection, we compare the performance of InterWeave's translation mechanism to that of object serialization in Sun's JDK v.1.3.1. We compare against the Sun implementation because it is significantly faster than Kaffe v.1.0.6, and because Kaffe was unable to successfully serialize large arrays in our experiments.
Figure 6.6: Comparison of double array translation between Sun JDK 1.3.1 and InterWeave.

We first compare the cost of translating a large array of primitive double variables in both systems. Under Sun JDK we create a Java program to serialize double arrays into byte arrays and to de-serialize the byte arrays back again. We measure the time for the serialization and deserialization. Under InterWeave Java support we create a program that allocates double arrays of the same size, releases (unmaps) the segment, and exits. We measure the release time and subtract the time spent on communication with the server. We then run a program that acquires (maps) the segment, and measure the time to translate the byte arrays back into doubles in Kaffe. Results are shown in Figure 6.6, for arrays ranging in size from 25000 to 250000 elements.

On average, InterWeave is about twenty-three times faster than JDK 1.3.1 in serialization, and 8 times faster in deserialization. InterWeave is more efficient here largely because the translation functions are implemented in the client library and is highly optimized, while the JDK uses a Java library. Furthermore, when translating pointers, the InterWeave library simply converts it between the local
pointers and the machine independent pointers. The JDK, on the other hand, has to keep a record of translated pointers to avoid loops in a deep-copy process.

6.3.3 Bandwidth Reduction

To evaluate the impact of InterWeave's diff-based wire format, which transmits an encoding of only those bytes that have changed since the previous communication, we modify the previous experiment to modify between 10 and 100% of a contiguous of a 200,000 element double array. Results appear in Figures 6.7 and 6.8. The former indicates translation time, the latter bytes transmitted.

It is clear from the graph that as we reduce the percentage of the array that is modified, both the translation time and the required communication bandwidth go down by linear amounts. By comparison, object serialization is oblivious to the fraction of the data that has changed.

![Graph showing time needed to translate a partly modified double array.](Image)

Figure 6.7: Time needed to translate a partly modified double array.
6.3.4 Java Visualization Client for the Astroflow Application

Several applications developed locally with colleagues from our own and other departments have benefited from using the InterWeave system and its Java implementation. One of them is the Astroflow application from the Physics and Astronomy Department. The original Astroflow visualization front-end is an offline Java visualization application. With limited changes, InterWeave allows it to connect directly to the simulator, creating an interactive system for visualization and steering. The architecture of the system is illustrated in Figure 1.1 (page 6).

The Astroflow front-end and the simulator share a segment with one header block specifying general configuration parameters and six \(256 \times 256\) arrays of doubles. The original simulator serializes the parameters and results into a data file. To use InterWeave, we wrote an XDR specification to describe the data structures we are sharing and replaced the original file operations with shared segment operations. No special care is required to support multiple visualization
<table>
<thead>
<tr>
<th></th>
<th>Kaffe with File I/O</th>
<th>InterWeave</th>
</tr>
</thead>
<tbody>
<tr>
<td>Per frame operation</td>
<td>74 msec.</td>
<td>25 msec.</td>
</tr>
</tbody>
</table>

Table 6.1: Astroflow Java visualization client performance using InterWeave.

clients or to control the frequency of updates. While the simulation data is relatively simple (and thus straightforward to write to a file), a simulator with more complex data structures would need no special code when using InterWeave. In comparison to writing an application-specific message protocol for the simulator and the satellite, we find data sharing to be a much more appealing mechanism for many applications, leading to code that is shorter, faster to write, and easier to understand.

Since we have evaluated the simulator performance on the InterWeave 3-level structure in the previous Chapter 5.3, we now examine the Java visualization front end. Table 6.1 compares the time required to read a frame of data from a file to the time required to lock the segment and obtain an update from the server. For a typical frame, InterWeave is three times faster than the file-based code. (In this particular application the use of diffs in wire format doesn’t lead to much savings in bandwidth: most of the simulation data is modified in every frame.)

Performance differences aside, we find the qualitative difference between file I/O and InterWeave segments to be compelling in this application. Our experience changing Astroflow from an off-line to an on-line client highlighted the value of letting InterWeave hide the details of network communication, multiple clients, and the coherence of transmitted simulation data.

6.4 Summary

In this Chapter we have described the InterWeave Java support. It allows Java applications to share information directly with applications written in other lan-
guages and running on heterogeneous platforms, using ordinary reads and writes [CTDS02, TCDS03]. We have demonstrated the efficiency and ease of use of the system through an evaluation on both real applications and artificial benchmarks. Quantitative data shows that the overhead incurred by our implementation is small and that the implementation is much faster in translating data than the object serialization implementation in standard JDK.
7 Conclusions

The growth of the Internet has enabled many application that require some sort of distributed shared state to coordinate processes at multiple sites. Message-based protocols can make the sharing extremely efficient, but considerable programmer expertise and effort is required. Distributed shared memory systems present a conceptually simpler programming model and can automate caching of remote state. However, they are usually considered less efficient because of the more conservative requirement to keep the distributed state coherent. Our work explores the design space of efficient support of distributed shared state in a heterogeneous, multi-level environment. Our evaluations on InterWeave, a middleware system that supports shared memory programming, demonstrate that InterWeave applications can be as efficient as applications using sophisticated messaging protocols, with a much simpler design and significantly less programming complexity.

7.1 Contributions

In this dissertation, we have discussed the design and implementation of the InterWeave system. InterWeave supports sharing among heterogeneous machines. An InterWeave programmer can use multiple languages to access shared state with ordinary reads and writes. We conducted experiments on a variety of applications
over different machine types, network configurations, and programming languages. Through these experiments, we evaluated our design choices and documented their benefit for improving application performance.

**Coherence** InterWeave uses a variety of techniques to optimize coherence update [CDP+00, CTC+01, CTC+02, TCDS03, CTS+03]. We use wire-format differencing to make the communication traffic for propagating modifications proportional to the amount of the data that has changed. The InterWeave relaxed coherence models allow applications to use temporal specifications to delay updates. An adaptive scheme allows applications to optimize lock acquisition time by automatically switching between client request and server notification modes. In the spatial dimension, dynamic views allow applications to limit the spatial range of coherence updates. We believe that InterWeave is the first to combine these techniques in a heterogeneous wide area networking environment.

Our evaluations on microbenchmarks and real applications show that any computation overhead from these techniques is amply compensated by the reduction in communication. It is much easier to optimize communication overhead using high-level coherence information than manually implementing caching in RPC or other message passing based systems. The benefit becomes even more substantial when we move applications to the Internet domain. Our experimentation in Chapter 3 and Chapter 4 demonstrate that over low-bandwidth Internet links, reductions in communication traffic result in proportional gains in application performance, making the overhead of coherence optimization negligible. Our evaluated applications have substantial performance advantages over those RPC applications that are naively implemented, while providing comparable efficiency with those RPC applications that have been manually optimized.

Our design of InterWeave complements message passing methods; InterWeave is not intended to replace them. Our experience with applications demonstrates
that InterWeave can be employed as a method to automate the management of caching in RPC systems or Java RMI. It can be used for caching reused parameters, results, or even internal service states. InterWeave's machine independent pointers can be used to support true "call by reference", thus saving the computation overhead and communication bandwidth of deep copying or callbacks.

**Multi-level Shared Memory** Realizing that distributed shared memory can be supported at different levels of hardware and software, InterWeave extends the two-level structure of the Cashmere system to a three-level structure [CDP+00, CTC+01, CTC+02]. When sharing heterogeneous data via an InterWeave segment, i.e., at level 3, an entire Cashmere cluster is treated as a single InterWeave client. The local mapping of the segment uses cluster wide Cashmere shared memory to leverage its support for maintaining consistency inside the cluster and for generating the level 3 wire-format diffs efficiently.

In the current 3-level design and implementation, we allow the lower level synchronization primitives to be exposed to applications running at a higher level. This is necessary for obtaining maximal performance benefit from the local support. Our experiments have demonstrated that the extra overhead for the three-level support on a Cashmere cluster is minimal. Applications can easily be adapted to allow sharing between a high performance cluster and remote satellites connected with much lower bandwidth network links.

**Java support** InterWeave not only supports heterogeneous machine types, it supports applications written in different languages [PCDS00, CTDS02, TCDS03]. Especially, its support for Java facilitates the sharing among legacy Fortran and C/C++ applications and interactive Java satellites. However, multi-language support is a special challenge for the Java runtime environment. We use a customized Java virtual machine to optimize write detection of InterWeave Java objects, as
required for efficient generation of wire-format diffs, although a pure Java implementation would be possible with reduced efficiency.

Our experiments show that the InterWeave Java implementation performs much better than Java object serializations in updating Java objects. The improvement is mostly due to the pre-compiled type information from the InterWeave IDL, dynamic type matching, and the elimination of deep copy of references.

7.2 Future Work

Improving InterWeave scalability is an important direction for future work. We need to optimize access to InterWeave segment servers as the number of sharing clients increases. One solution we are looking at is to replicate each InterWeave server and organize the replicas into a spanning-tree hierarchical structure. The original server sits at the root of the tree. A client can connect to any replica. Higher-level replicas issue writing leases to lower-level replicas to allow scalable local update, thus improving the server's throughput.

We have designed and implemented preliminary transaction support for InterWeave [SCTD03, TCDS04]. It supports better failure recovery from various application-specific or coherence and consistency errors. Based on this transactional support, we are investigating the possibility of speculative relaxed coherence models and dynamic views. For example, a programmer can mark a speculative view in an application from the run-time profile information. When the application runs with the view speculatively, the runtime transactional system dynamically monitors access to the shared segment. If the view is violated, InterWeave can simply abort the transaction and retry with the view disabled.

One important problem in Grid computing is the secure and efficient management of large volumes of data over geographically dispersed computing cen-
ters. As an infrastructure for sharing state efficiently, InterWeave could be very helpful for automating metadata management in the Grid. For example, in a data grid environment, cooperating labs share large volumes of experimental results. The result files might be replicated at multiple sites using different granularities. The data grid manages the files using a collection of replica directories [FVWZ02, RF02, SSA+01]. We are beginning to look at using InterWeave to optimize distributed access to these directories.

Privacy and security are important issues in distributed shared state. Currently, InterWeave provides limited ability to authenticate clients and servers by previously agreed public keys. The exchange of shared state between a server and its clients is not encrypted and could be subjected to attacks from eavesdroppers. In the case that the server is not trustworthy, the clients need to protect their information by encrypting their information using encryption keys that are mutually agreed upon. Otherwise, an impostor server can easily obtain private information that the clients have saved into the segment. However the encrypted segment would make it very difficult or even impossible for the server to maintain coherency for the data segment efficiently. For example, the server would not be able to compute diffs for lock acquires. It will be interesting to investigate available solutions to balance client privacy and server efficiency (e.g., by using the convergent encryption [DAB+02] technique).

### 7.3 Recent Publications

Most of the content found in this dissertation has been published before in URCS technical reports [CTC+01, TCDS02] and in ACM/IEEE conferences [CDP+00, PCDS00, CTC+02, CTDS02, TCDS03, SCTD03, CTS+03, TCDS04]. Besides InterWeave, I have published some other papers on Java virtual machine systems [CMB+01] and pervasive computing [CMMD03, MGB+02].
Bibliography


[CGS96] T. Cornileanu and Eric Gressier-Soudan. A combined-consistency ap-


[IRC] IRCache Project. Originally sponsored by the National Science Foundation (grants NCR-9616602 and NCR-9521745), and the National Laboratory for Applied Network Research. http://www ircache net.


[LS88] Richard J. Lipton and Jonathan S. Sandberg. PRAM: A Scalable


[RF02] Kavitha Ranganathan and Ian Foster. Decoupling Computation and Data Scheduling in Distributed Data-Intensive Applications. In *Pro-


[SNS02a] Brandon C.S. Sanders, Randal C. Nelson, and Rahul Sukthankar. The


plementation (OSDI'02), pages 225–238, Boston, Massachusetts, December 2002.


[TTP+95] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update con-


