InterWeave: A Middleware System for Distributed Shared State *

DeQing Chen, Sandhya Dwarkadas, Srinivasan Parthasarathy,
Eduardo Pinheiro, and Michael L. Scott

Computer Science Department, University of Rochester
{lukechen,sandhya,srini,edpin,scott}@cs.rochester.edu

Abstract. As an alternative to message passing, Rochester's InterWeave system allows the programmer to map shared segments into programs spread across heterogeneous, distributed machines. InterWeave represents a merger and extension of our previous Cashmere and InterAct projects, combining hardware coherence within small multiprocessors, Cashmere-style lazy release consistency within tightly coupled clusters, and InterAct-style version-based consistency for distributed shared segments.
In InterWeave, each shared segment evolves through a series of consistent versions. When beginning a read-only critical section on a given segment, InterWeave uses a programmer-specified predicate to determine whether the currently cached version, if any, is "recent enough" to use. Inter-segment consistency is maintained by means of hashed vector timestamps. Automatic data conversions allow each program to employ its own natural data format, byte order, and alignment, with full support for intra- and inter-segment pointers. Timestamping is used to determine and communicate only those pieces of a segment that are different from the cached copy.
A preliminary implementation of InterWeave is currently running on our AlphaServer cluster. Driving applications include data mining, intelligent distributed environments, and scientific visualization.

1 Introduction

Advances in processing speed and network bandwidth are creating new interest in such ambitious distributed applications as interactive data mining, remote scientific visualization, computer-supported collaborative work, and intelligent environments. Most of these applications rely, at least in the abstract, on some notion of distributed shared state. When one of their processes must access data that are currently located elsewhere, one has the option of moving the process to the data or moving the data to the process. Either option may make sense from a performance point of view, depending on the amounts of data and computation involved, the feasibility of migration, and the frequency of data updates.

The first option—move the process to the data—corresponds to remote procedure call or remote method invocation, and is supported by widely available production-quality systems. The second option—move the data to the process—is not so well

* This work is supported in part by NSF grants EIA–9972881, CCR–9702466, CCR–9705594, and CCR-9988361; and an external research grant from Compaq.

understood. It still tends to be achieved through special-purpose, application-specific message-passing protocols. The creation of these protocols is a time-consuming, tedious, and error-prone activity. It is complicated by the need, for performance reasons, to cache copies of data at multiple locations, and to keep those copies consistent in the face of distributed updates.

At Rochester we have been discussing these issues with colleagues in data mining, scientific visualization, and distributed intelligent environments, all of whom have very large distributed data sets. To support their applications, we are developing a system, known as InterWeave, that allows the programmer to map shared segments into program components regardless of location or machine type. InterWeave represents a merger and extension of our previous Cashmere [39] and InterAct [30] projects. Once shared segments have been mapped, InterWeave can support hardware coherence and consistency within multiprocessors (*level-1* sharing), Cashmere-style software distributed shared memory within tightly coupled clusters (*level-2* sharing), and InterAct-style version-based consistency across the Internet (*level-3* sharing) for these segments (see Figure 1).

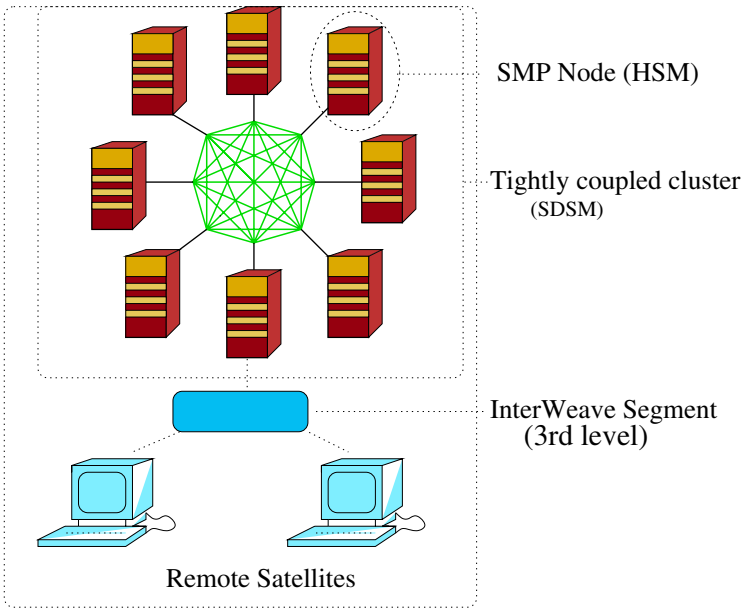

Fig. 1. InterWeave's target environment.

At the third level, each segment in InterWeave evolves through a series of consistent versions. When beginning a read-only critical section on a given segment, InterWeave uses a programmer-specified predicate to determine whether the currently cached version, if any, is "recent enough" to use. Several coherence models (notions of "recent enough") are built into the InterWeave system; others can be defined by application programmers. When the application desires causality among segments, to avoid causal-

ity loops, we invalidate mutually-inconsistent versions of other segments, using a novel hashing mechanism that captures the history of each segment in a bounded amount of space.

Like CORBA [29] and many older RPC systems, InterWeave employs a type system based on a machine- and language-independent interface description language, in our case Sun XDR [40]. We do not require that programmers adhere to an object-oriented programming style. We simply ensure that the version of a segment cached by a given program component is appropriate to the component's language and machine architecture. When transmitting data between machines, we convert to and from a standard wire format. We also swizzle pointers [45], so that references to data currently cached on the local machine are represented as machine addresses. We even allow programs to organize dynamically-allocated data within a segment in different ways on different machines, for the sake of spatial locality.

We describe the design of InterWeave in more detail in Section 2, covering synchronization, coherence, consistency, heterogeneity, and integration with existing shared memory. Our initial implementation and preliminary performance results are described in Section 3. We compare our design to related work in Section 4 and conclude with a discussion of status and plans in Section 5.

2 InterWeave Design

The unit of sharing in InterWeave is a self-descriptive data segment within which programs allocate strongly typed blocks of memory. Every segment has an Internet URL. The blocks within a segment are numbered and optionally named. By concatenating the segment URL with a block number/name and offset (delimited by pound signs), we obtain a machine-independent pointer: "`http://foo.org/path#block#offset`". To accommodate heterogeneous data formats, offsets are measured in primitive data units—characters, integers, floats, etc.—rather than in bytes. To create and initialize a segment in C, we execute the following calls:

```
IW_handle_t h = IW_create_segment (url);
IW_wl_acquire (h);
my_type* p = (my_type *) IW_malloc (h, my_type_desc);
*p = ...
IW_wl_release (h);
```

Every segment is managed by an InterWeave server at the IP address indicated in the segment's URL. Assuming appropriate access rights, the `IW_create_segment` call communicates with the server to create an uninitialized segment, and allocates space to hold (the initial portion of) a local cached copy of that segment in the caller's address space. The handle returned by `IW_create_segment` is an opaque, machine-dependent type that may be passed to `IW_malloc`, along with a type descriptor generated by our XDR compiler. Copies of a segment cached by a given process need not necessarily be contiguous in the application's virtual address space, so long as individually `malloc`ed blocks are contiguous; the InterWeave library can expand a segment as needed using unrelated address ranges.

Once a segment has been initialized, a process can create a machine-independent pointer to an arbitrary location within one of its allocated blocks:

```
IW_mip_t m = IW_ptr_to_mip (p);
```

This machine-independent pointer can then be passed to another process through a message, a file, or even console I/O. Given appropriate access rights, the other process can convert back to a machine-specific pointer:

```
my_type *p = (my_type *) IW_mip_to_ptr (m);
```

The `IW_mip_to_ptr` call reserves space for the specified segment if it is not already locally cached (communicating with the server if necessary to obtain layout information for the specified block), and returns a local machine address. Actual data for the segment will not be copied into the local machine until the segment is locked. The mechanism used to specify and verify access rights is still under development.

Any given segment A may contain pointers to data in some other segment B. The pointer-swizzling and data-conversion mechanisms described in Section 2.3 below ensure that such pointers will be valid local machine addresses, and may freely be dereferenced. It remains the programmer's responsibility, however, to ensure that segments are accessed only under the protection of reader-writer locks. To assist in this task, InterWeave allows the programmer to identify the segment in which the datum referenced by a pointer resides:

```
IW_handle_t h = IW_get_handle (p);
```

2.1 Coherence

Given the comparatively high and variable latencies of local-area networks, traditional hardware-inspired consistency models are unlikely to admit good performance in a distributed environment. Even the most relaxed of these models, release consistency [16], guarantees a coherent view of *all* shared data among *all* processes at synchronization points, resulting in significant amounts of communication.

Fortunately, processes in distributed applications can often accept a significantly more relaxed—and hence less costly—notion of consistency. Depending on the application, it may suffice to update a cached copy of a segment at regular (temporal) intervals, or whenever the contents have changed "enough to make a difference", rather than after every change.

The server for a given segment in InterWeave is responsible for the segment's coherence. This coherence is based on the notion that segments move over time through a series of internally consistent states, under the protection of reader-writer locks.

When writing a segment, a process must have exclusive 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. InterWeave inherits five different definitions of "recent enough" from its predecessor system, InterAct. It is also designed in such a way that additional definitions (coherence models) can be added easily. Among the current models, *Full coherence* always obtains the most recent version of the segment; *Null*

coherence always accepts the currently cached version, if any (the process must employ additional, explicit library calls to obtain an update); *Delta coherence* [38] guarantees that the segment is no more than x versions out-of-date; *Temporal coherence* guarantees that it is no more than x time units out of date; and *Diff-based coherence* guarantees that no more than x% of the segment is out of date. In all cases, x can be specified by the process.

When a process first locks a shared segment, the InterWeave library obtains a copy from the segment's server. At each subsequent read-lock acquisition, the InterWeave library checks to see whether the local copy of the segment is "recent enough". If not, it obtains a version update from the server. Twin and diff operations [8], extended to accommodate heterogeneous data formats (Section 2.3), allow InterWeave to perform an update in time proportional to the fraction of the data that has changed.

The relaxed semantics of read locks imply that a process may hold a write lock (with exclusive access to the current version of the segment) even when other processes are reading older versions. To support concurrent access by readers that need to exclude any writer, InterWeave also supports a strict read lock.

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 then override this default for its own lock operations. Different processes may therefore use different coherence models for the same segment. These are entirely compatible: the server for a segment always has the most recent version; the model used by a given process simply determines when it decides whether its own cached copy is recent enough.

The server for a segment need only maintain a copy of the segment's most recent version. Older versions are not required, because the API specifies that the current version of a segment is always acceptable, and since processes cache whole segments, they never need an "extra piece" of an old version. To minimize the cost of segment updates, the server maintains a timestamp on each block of each segment, so that it can avoid transmitting copies of blocks that haven't changed.

As noted in Section 1, a Cashmere-style "level-2" sharing system plays the role of a single node at level 3. A process in a level-2 system that obtains a level-3 lock does so on behalf of its entire level-2 system, and may share access to the segment with its level-2 peers. The runtime system guarantees that updates are propagated consistently, and that protocol overhead required to maintain coherence is not replicated at levels 2 and 3. Further details appear in Section 3.

2.2 Consistency

While InterWeave's predecessor, InterAct, has proven useful for many applications (in particular, we have used it successfully for interactive datamining [30]), it does not respect causality: in the face of multi-version relaxed consistency, the versions of segments currently visible to a process may not be consistent with what Lamport called the "happens-before" relationship [26]. Specifically, let A_i refer to version i of segment A. If B_j was created using information found in A_i, then previous versions of A are causally incompatible with B_j; a process that wants to use B_j (and that wants to respect

causality) should invalidate any cached segment versions that predate the versions on which B_j depends.

To support this invalidation process, we would ideally like to tag each segment version with the names of all segment versions on which it depends. Then any process that acquired a lock on a segment would check to see whether it depends on newer versions of any segments currently locally cached. If so, the process would invalidate those segments.

The problem with this scheme is that the number of segments in the system—and hence the size of tags—is unbounded. One simple solution is to hash the information. We let every segment version S_i carry 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 process maintains a local timestamp that indicates (in hashed form) the most recent segment versions it has read. 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. We have developed refinements to this scheme to accommodate roll-over of the values within timestamps, and to reduce the chance that hash collisions will cause repeated extraneous invalidations of a segment that seldom changes.

To support operations on groups of segments, we allow their locks to be acquired and released together. 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. For sharing levels 1 and 2, consistency is guaranteed for data-race-free [1] programs.

2.3 Heterogeneity

The Internet is highly heterogeneous. Even our local-area network includes Suns, Linux and Windows 2000 PCs, SGI machines, Macintoshes, Alphas, and a variety of special-purpose peripherals. To accommodate such a variety of architectures, remote procedure call systems usually incorporate a language- and machine-independent notation to describe the types of parameters, together with a stub compiler that automatically translates to and from a universal "wire format". Any system for distributed shared state must provide a similar level of support for heterogeneity.

Segments in InterWeave are currently similar to those in InterAct, and are derived from a C++ base class with special constructor, destructor, and synchronization methods. InterWeave uses the C++ reflection mechanism to obtain type information and to identify intra- and inter-segment pointers, so that data can be translated appropriately when sent from one machine to another. Other systems such as Orca [4] provide similar representations for address-independent segments.

We are in the process of eliminating our dependence on C++ by using Sun's XDR language to define the types of data within segments. Pointer swizzling [45] will be used to accommodate reference types. Briefly, swizzling uses type information to find all (machine-independent) pointers within a newly-acquired segment, and converts them to pointers that work on the local machine. Pointers to segments that are not (yet) locally cached point into reserved but unmapped pages where data will lie once properly locked. The set of segments currently cached on a given machine thus displays an "expanding frontier" reminiscent of lazy dynamic linking. As noted at the beginning of this section, each segment is structured as a heap in which blocks may be allocated dynamically. Further detail can be found in our paper at WSDSM 2000 [33]. In keeping with our work on InterAct, we will allow compilers and smart applications to control the relative placement of blocks within the heap, to maximize cache performance under different traversal orders. The code that transfers segments from one machine to another will automatically re-order items in the heap according to local preference, as part of the swizzling process.

3 Implementation and Performance

In this section, we describe the current version of our implementation prototype, and present preliminary performance data for remote visualization of an N-body simulation.

3.1 Implementation

Our current implementation (see Figure 2) employs a server process for each segment. The server keeps metadata for each active client of the segment, as well as a master copy of the segment's data. Communication with each client is managed by a separate thread.

Each segment client can be either a single process or a tightly coupled cluster. When a client obtains a write lock, it uses virtual memory mechanisms to identify pages of the segment's local copy that are about to be modified. For each such page it creates a pristine copy (called a *twin*) before allowing modification. At the time of the write lock release, the runtime library uses the twins and other local meta-data (specifically, type descriptors) to construct a *machine-independent diff* that describes changes in terms of field offsets within blocks. Blocks above a certain minimum size are logically subdivided into "chunks" so that a small change to a very large block need not create a large diff. The machine-independent diff is sent back to the segment server to update its master copy.

When a tightly coupled cluster, such as a Cashmere-2L system, uses an InterWeave segment, the cluster appears as a single client to the segment server. The InterWeave system uses cluster-wide shared memory for the segment local copy. Our goal is to minimize any additional overhead due to incorporating the third level into the system. In the current implementation, we designate a node inside the cluster as the cluster's manager node. All of the third level interactions with the segment server go through the manager node. During the period between a write lock acquire and release, the same twins are used by both the second and third level systems (see [39] for details on the

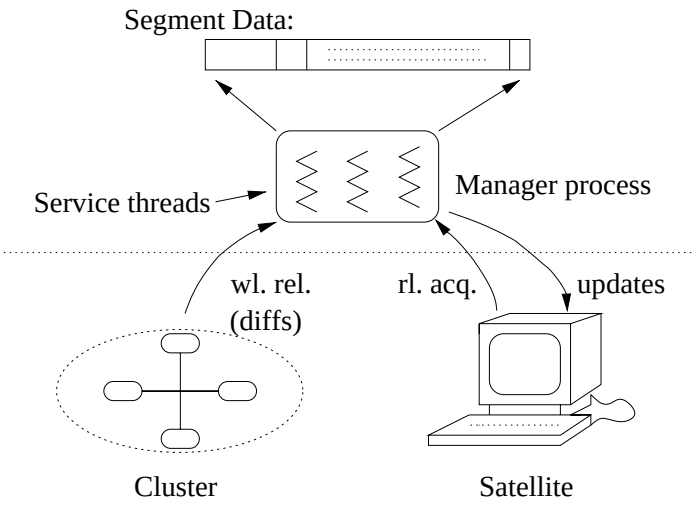

Fig. 2. Current InterWeave implementation.

Cashmere-2L implementation). At a second level release, in addition to sending diffs to the second level home node, the runtime system sends a third level diff to the manager node. The manager node merges all of these diffs and sends them to the segment server at the time of the third level write-lock release. One optimization would be to have the second level home nodes maintain the cumulative third level diffs. This would eliminate communication with a possibly separate manager node entirely. We are currently incorporating this optimization into our system.

3.2 Experiments

To evaluate our prototype implementation of InterWeave, we have collected performance measurements on a remote visualization of the Splash-2 [46] Barnes-Hut simulation. The simulation runs on a 4-node, 16-processor Cashmere system. Each node 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. The nodes are connected by a Memory Channel 2 [13] system area network. The simulation repeatedly computes new positions for 16,000 bodies. These positions may be shared with a remote visualization satellite via an InterWeave segment. The simulator uses a write lock to update the shared segment, while 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.

When we combine the high performance second level shared memory (Cashmere) with the third level shared memory (InterWeave), it would be ideal if there were no degradation in the performance of the second level 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 a *sole-sharer* optimization that avoids the transmission of diffs when there is only a single known copy of the data.

Execution times for the no-satellite experiment appear in Figure 3. Each bar gives aggregate wall-clock time for ten iteration steps. In each pair of bars, the one on the right is for the standard Cashmere system; the one on the left is for Cashmere linked with the InterWeave library and communicating with a server. The left-hand bars are subdivided to identify the overhead due to running the third-level protocol code. This overhead is negligible for small configurations, but increases to about 11% for 16 processors on 4 nodes. This non-scalability can be explained by our use of a single manager node within the Cashmere cluster. As the number of processes increases, the manager has to spend more time collecting diffs, which makes the system unbalanced. As described in Section 3.1, we are working to eliminate this bottleneck.

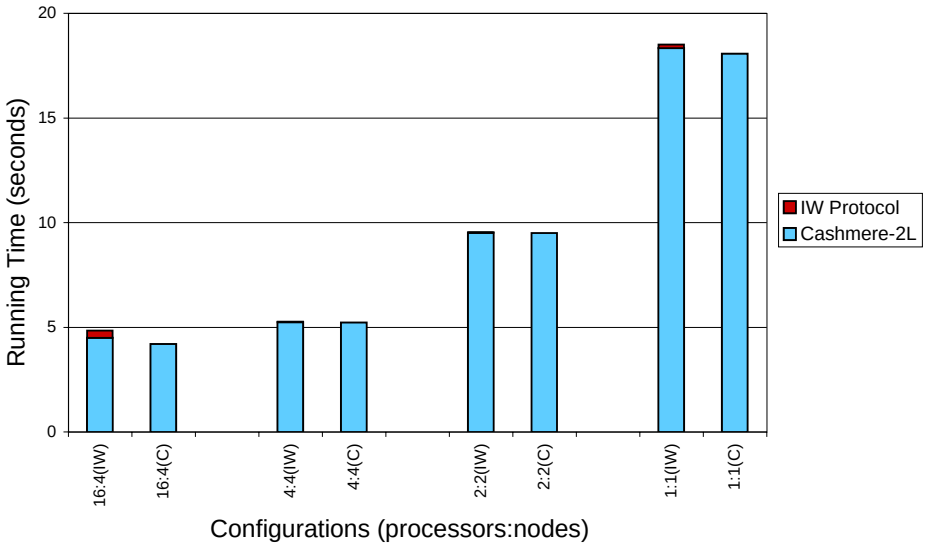

Fig. 3. Overhead of InterWeave library without a satellite.

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. Preliminary results appear in Figure 4. In all cases the satellite was running on another Alpha node, communicating with the cluster and server, if any, via TCP/IP over Fast Ethernet. We have again subdivided execution time, this time to separate out both communication and (for the left-hand bars) InterWeave protocol overhead. The overhead of the InterWeave protocol itself remains relatively small, but communi-

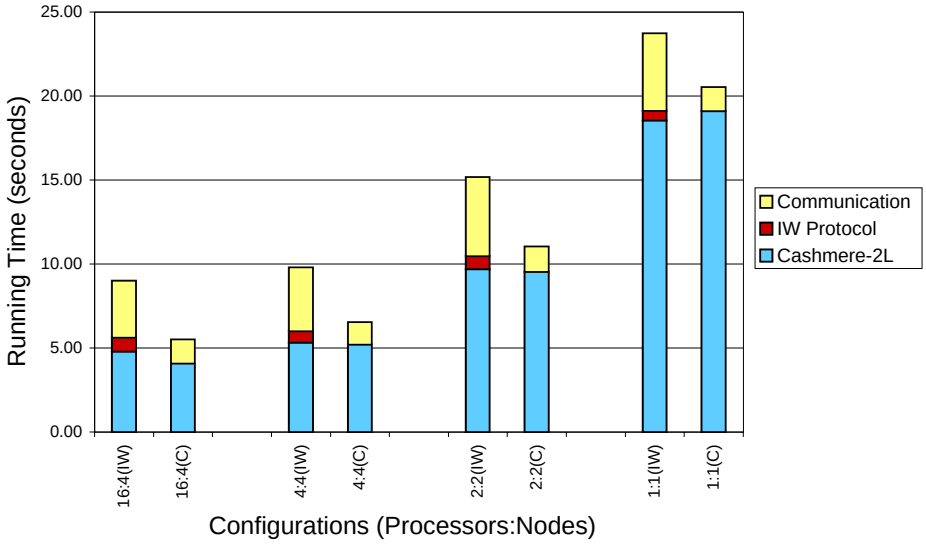

Fig. 4. Overhead of InterWeave library and communication during remote visualization.

cation overhead is significant, due to InterWeave's unoptimized and unaggregated communication with the server. We believe we can eliminate much of this overhead through implementation improvements; such improvements will be a major focus of ongoing work.

A key advantage of the InterWeave version of the visualization program is that the simulation need not be aware of the number of satellites or the frequency of sharing. In the version of the application that uses hand-written message passing, this knowledge is embedded in application source code.

4 Related Work

InterWeave finds context in an enormous body of related work—far too much to document thoroughly here.

Most systems for distributed shared state enforce a strongly object-oriented programming model. Some, such as Emerald [23], Argus [27], Ada [22], and ORCA [42], take the form of an explicitly distributed programming language. Some, notably Amber [10] and its successor, VDOM [11], are C++-specific. Many in recent years have been built on top of Java; examples include Aleph [20], Charlotte [5], Java/DSM [48], Javelin [7], JavaParty [31], JavaSpaces [41], and ShareHolder [18].

Language-independent distributed object systems include PerDiS [12], Legion [17], Globe [44], DCOM [35], and various CORBA-compliant systems [29]. Globe replicates objects for availability and fault tolerance. PerDiS and a few CORBA systems (e.g., Fresco [25]) cache objects for locality of reference. Thor [28] enforces type-safe object-oriented access to records in a heterogeneous distributed database.

At least two early software distributed shared memory (S-DSM) systems provided support for heterogeneous machine types. Toronto's Mermaid system [49] allowed data to be shared across more than one type of machine, but only among processes created as part of a single run-to-completion parallel program. All data in the same VM page was required to have the same type, and only one memory model—sequential consistency—was supported. CMU's Agora system [6] supported sharing among more loosely-coupled processes, but in a significantly more restricted fashion than in InterWeave. Pointers and recursive types were not supported, all shared data had to be accessed indirectly through a local mapping table, and only a single memory model (similar to processor consistency) was supported.

Perhaps the two projects most closely related to InterWeave are Khazana [9] and Active Harmony [21]. Both are outgrowths of previous work in software distributed shared memory. Both support distributed sharing without enforcing an object-oriented programming style. Khazana proposes a global, 128-bit address space for all the world's shared data. It does not impose any structure on that data, or attempt to translate it into locally-appropriate form. Active Harmony is more explicitly oriented toward high-performance parallel execution. Early work appears to have focussed primarily on load balancing and process management. Various Linda systems [32, 37] also provide a non-object-oriented distributed shared store.

Interface description languages date from Xerox Courier [47] and related systems of the early 1980s. Precedents for the automatic management of pointers include Herlihy's thesis work [19], LOOM [24], and the more recent "pickling" (serialization) of Java [34]. Friedman [15] and Agrawal et al. [2] have shown how to combine certain pairs of consistency models in a non-version-based system. Alonso et al. [3] present a general system for relaxed, user-controlled coherency. We explore a real implementation of a dynamically adjustable coherence mechanism in an environment that allows tightly-coupled sharing in addition to the relaxed coherence in a more distributed environment. Several projects, including ShareHolder, Globus [14], and WebOS [43], use URL-like names for distributed objects or files. Khazana proposes the use of multiple consistency models.

5 Status and Plans

A preliminary version of InterWeave is up and running on Rochester's AlphaServer cluster. It provides full support for user-specified coherence predicates, but does not yet implement inter-segment coherence. The type system is currently based on C++, rather than XDR, and though we support user-specified data layouts (rearrangement of blocks in the heap), we have not yet implemented data conversions for heterogeneous machine architectures. The current system supports working demonstrations of remote interactive parallel association mining and visualization of a parallel N-body simulation, demonstrating the utility of the system in combining distributed sharing with tightly-coupled coherence.

Once the basic features of InterWeave are in place, we expect to turn to several additional issues, including security, fault tolerance, and transactions. We hope to leverage the protection and security work of others, most likely using a group-based system

reminiscent of AFS [36]. Toleration of client failures is simplified by the version-based programming model: a segment simply reverts to its previous version if a client dies in the middle of an update. Server faults might be tolerated by pushing new versions through to stable storage. Ultimately, a true transactional programming model (as opposed to simple reader-writer locks) would allow us to recover from failed operations that update multiple segments, and to implement two-phase locking to recover from deadlock or causality violations when using nested locks.

References

[1] S. V. Adve and M. D. Hill. A Unified Formulation of Four Shared-Memory Models. *IEEE Trans. on Parallel and Distributed Systems*, 4(6):613–624, June 1993.

[2] D. Agrawal, M. Choy, H. V. Leong, and A. K. Singh. Mixed Consistency: A Model for Parallel Programming. In *Proc. of the 13th ACM Symp. on Principles of Distributed Computing*, Los Angeles, CA, Aug. 1994.

[3] R. Alonso, D. Barbara, and H. Garcia-Molina. Data Caching Issues in an Information Retrieval System. *ACM Trans. on Database Systems*, 15(3):359–384, Sept. 1990.

[4] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. In *IEEE Trans. on Software Engineering*, pages 190–205, June 1992.

[5] A. Baratloo, M. Karaul, V. Keden, and P. Wyckoff. Charlotte: Metacomputing on the Web. *Intl. Journal on Future Generation Computer Systems*, 15(5-6):559–570, 1999.

[6] R. Bisiani and A. Forin. Multilanguage Parallel Programming of Heterogeneous Machines. *IEEE Trans. on Computers*, 37(8):930–945, Aug. 1988.

[7] P. Cappello, B. O. Christiansen, M. F. Ionescu, M. O. Neary, K. E. Schauser, and D. Wu. Javelin: Internet-Based Parallel Computing Using Java. In *1997 ACM Workshop on Java for Science and Engineering Computation*, Las Vegas, NV, June 1997.

[8] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proc. of the 13th ACM Symp. on Operating Systems Principles*, pages 152–164, Pacific Grove, CA, Oct. 1991.

[9] J. Carter, A. Ranganathan, and S. Susarla. Khazana: An Infrastructure for Building Distributed Services. In *Intl. Conf. on Distributed Computing Systems*, pages 562–571, May 1998.

[10] J. S. Chase, F. G. Amador, E. D. Lazowska, H. M. Levy, and R. J. Littlefield. The Amber System: Parallel Programming on a Network of Multiprocessors. In *Proc. of the 12th ACM Symp. on Operating Systems Principles*, pages 147–158, Litchfield Park, AZ, Dec. 1989.

[11] M. J. Feeley and H. M. Levy. Distributed Shared Memory with Versioned Objects. In *OOPSLA '92 Conf. Proc.*, pages 247–262, Vancouver, BC, Canada, Oct. 1992.

[12] P. Ferreira, M. Shapiro, X. Blondel, O. Fambon, J. Garcia, S. Kloosterman, N. Richer, M. Roberts, F. Sandakly, G. Coulouris, J. Dollimore, P. Guedes, D. Hagimont, and S. Krakowiak. PerDiS: Design, Implementaiton, and Use of a PERsistent DIstributed Store. Research Report 3525, INRIA, Rocquencourt, France, Oct. 1998.

[13] M. Fillo and R. B. Gillett. Architecture and Implementation of Memory Channel 2. *Digital Technical Journal*, 9(1):27–41, 1997.

[14] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *Intl. Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[15] R. Friedman. Implementing Hybrid Consistency with High-Level Synchronization Operations. In *Proc. of the 12th ACM Symp. on Principles of Distributed Computing*, Ithaca, NY, Aug. 1993.

[16] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. L. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proc. of the 17th Intl. Symp. on Computer Architecture*, pages 15–26, Seattle, WA, May 1990.

[17] A. S. Grimshaw and W. A. Wulf. Legion — A View from 50,000 Feet. In *Proc. of the 5th Intl. Symp. on High Performance Distributed Computing*, Aug. 1996.

[18] J. Harris and V. Sarkar. Lightweight Object-Oriented Shared Variables for Distributed Applications on the Internet. In *OOPSLA '98 Conf. Proc.*, pages 296–309, Vancouver, Canada, Oct. 1998.

[19] M. Herlihy and B. Liskov. A Value Transmission Method for Abstract Data Types. *ACM Trans. on Programming Languages and Systems*, 4(4):527–551, Oct. 1982.

[20] M. Herlihy. The Aleph Toolkit: Support for Scalable Distributed Shared Objects. In *Workshop on Communication, Architecture, and Applications for*, Orlando, FL, Jan. 1999.

[21] J. K. Hollingsworth and P. J. Keleher. Prediction and Adaptation in Active Harmony. In *Proc. of the 7th Intl. Symp. on High Performance Distributed Computing*, Chicago, IL, Apr. 1998.

[22] International Organization for Standardization. Information Technology — Programming Languages — Ada. Geneva, Switzerland, 1995. ISO/IEC 8652:1995 (E). Available in hypertext at `http://www.adahome.com/rm95/`.

[23] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-Grained Mobility in the Emerald System. *ACM Trans. on Computer Systems*, 6(1):109–133, Feb. 1988. Originally presented at the *11th ACM Symp. on Operating Systems Principles*, Nov. 1987.

[24] T. Kaehler. Virtual Memory on a Narrow Machine for an Object-Oriented Language. In *OOPSLA '86 Conf. Proc.*, pages 87–106, Portland, OR, Sept. – Oct. 1986.

[25] R. Kordale, M. Ahamad, and M. Devarakonda. Object Caching in a CORBA Compliant System. *Computing Systems*, 9(4):377–404, Fall 1996.

[26] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Comm. of the ACM*, 21(7):558–565, July 1978.

[27] B. Liskov. Distributed Programming in Argus. *Comm. of the ACM*, 31(3):300–312, Mar. 1988.

[28] B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, and L. Shrira. Safe and Efficient Sharing of Persistent Objects in Thor. In *Proc. of the 1996 ACM SIGMOD Intl. Conf. on Management of Data*, Montreal, Canada, June 1996.

[29] Object Management Group, Inc. The Common Object Request Broker: Architecture and Specification, Revision 2.0. Framingham, MA, July 1996.

[30] S. Parthasarathy and S. Dwarkadas. InterAct: Virtual Sharing for Interactive Client-Server Applications. In *4th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, May 1998.

[31] M. Philippsen and M. Zenger. JavaParty — Transparent Remote Objects in Java. *Concurrency — Practice and Experience*, 9(11):1125–1242, Nov. 1997.

[32] G. P. Picco, A. L. Murphy, and G. Roman. Lime: Linda meets mobility. In *Proc. of the 21st Intl. Conf. on Software Engineering*, pages 368–377, Los Angeles, CA, May 1999.

[33] E. Pinheiro, D. Chen, S. Dwarkadas, S. Parthasarathy, and M. L. Scott. S-DSM for Heterogeneous Machine Architectures. In *Proc. of the 2nd Workshop on Software Distributed Shared Memory*, Santa Fe, NM, May 2000. In conjunction with the *14th Intl. Conf. on Supercomputing*.

[34] R. Riggs, J. Waldo, A. Wollrath, and K. Bharat. Pickling State in the Java System. *Computing Systems*, 9(4):291–312, Fall 1996.

[35] D. Rogerson. *Inside COM*. Microsoft Press, Redmond, Washington, Jan. 1997.

[36] M. Satyanarayanan. Integrating Security in a Large Distributed System. *ACM Trans. on Computer Systems*, 7(3):247–280, Aug. 1989.

[37] Scientific Computing Associates Inc. Virtual Shared Memory and the Paradise System for Distributed Computing. Technical Report, New Haven, CT, April 1999.

[38] A. Singla, U. Ramachandran, and J. Hodgins. Temporal Notions of Synchronization and Consistency in Beehive. In *Proc. of the 9th Annual ACM Symp. on Parallel Algorithms and Architectures*, Newport, RI, June 1997.

[39] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In *Proc. of the 16th ACM Symp. on Operating Systems Principles*, St. Malo, France, Oct. 1997.

[40] *Network Programming Guide — External Data Representation Standard: Protocol Specification.* Sun Microsystems, Inc., 1990.

[41] Sun Microsystems. JavaSpaces Specification. Palo Alto, CA, Jan. 1999.

[42] A. S. Tanenbaum, M. F. Kaashoek, and H. E. Bal. Parallel Programming Using Shared Objects and Broadcasting. *Computer*, 25(8):10–19, Aug. 1992.

[43] A. Vahdat, T. Anderson, M. Dahlin, D. Culler, E. Belani, P. Eastham, and C. Yoshikawa. WebOS: Operating System Services for Wide Area Applications. In *Proc. of the 7th Intl. Symp. on High Performance Distributed Computing*, Chicago, IL, July 1998.

[44] M. van Steen, P. Homburg, and A. S. Tanenbaum. Globe: A Wide-Area Distributed System. In *IEEE Concurrency*, pages 70–78, Jan.-Mar. 1999.

[45] P. R. Wilson. Pointer Swizzling at Page Fault Time: Efficiently and Compatibly Supporting Huge Address Spaces on Standard Hardware. In *International Workshop on Object Orientation in Operating Systems*, page 244ff, Paris, France, Sept. 1992.

[46] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. Methodological Considerations and Characterization of the SPLASH-2 Parallel Application Suite. In *Proc. of the 22nd Intl. Symp. on Computer Architecture*, Santa Margherita Ligure, Italy, June 1995.

[47] Xerox Corporation. Courier: The Remote Procedure Call Protocol. Technical Report XSIS 038112, Dec. 1981.

[48] W. Yu and A. Cox. Java/DSM: A Platform for Heterogeneous Computing. *Concurrency — Practice and Experience*, 9(11), Nov. 1997.

[49] S. Zhou, M. Stumm, K. Li, and D. Wortman. Heterogeneous Distributed Shared Memory. In *IEEE Trans. on Parallel and Distributed Systems*, pages 540–554, 1992.