S-DSM for Heterogeneous Machine Architectures *

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

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

April 2000

Abstract

Many—indeed most—distributed applications employ some notion of distributed shared state: information required at more than one location. For applications that span the Internet, this state is almost always maintained by means of hand-written, application-specific message-passing protocols. These protocols constitute a significant burden on the programmer. Rochester's InterWeave project seeks to eliminate this burden by automating the management of shared state for processes on heterogeneous, distributed machines.

Heterogeneity implies the need for strong typing and for automatic conversion to and from a common wire format when transmitting data and updates between machines. In addition to issues of byte order, alignment, and numeric formats, transparent sharing implies the need for pointers that refer to arbitrary shared locations, and that operate with the speed of ordinary machine addresses when the target is locally cached. To satisfy these constraints, InterWeave employs a typesafe interface description language and compiler, performs page-fault-driven pointer swizzling, and maintains an elaborate collection of metadata to support fast data access and updates.

1 Introduction

Software distributed shared memory (S-DSM) provides the illusion of coherent memory sharing for machines connected only by a message-passing network, or by a non-cache-coherent memory system. Traditional S-DSM systems provide a conceptually appealing programming model for processes that have been spread across a locally-distributed cluster for the purpose of parallel speedup. Rochester's InterWeave project, by contrast, attempts to provide this model for processes that have been distributed across a potentially wide-area network in order to be co-located with distributed users, devices, or data repositories. For processes that happen to share a high-bandwidth, low-latency network, InterWeave employs the pre-existing, high-performance Cashmere [12, 11] protocol. The emphasis of the project, however, is on more loosely-coupled distribution and targeted to a different programming model. Traditional DSM systems often require the programmer to write SPMD (Single Process Multiple Data) code. InterWeave, on the other hand, can be used by sequential (distributed) applications as well as parallel applications sharing data or communicating in any way the application programmer desires.

Because of its different underlying motivation, InterWeave differs from Traditional S-DSM in several important ways [2]. To support sharing among loosely-coupled, independently-developed and deployed processes, InterWeave supports an arbitrary number of independent shared persistent segments, each named by a URL and managed by a server at the site identified in that URL. Individual blocks of data are dynamically allocated within segments, and may also be given symbolic names.

Each segment moves over time through a series of internally consistent versions, each of which appears to have been created atomically from the point of view of processes other than the creator. InterWeave clients cache whole segments, and may continue to use a given version of a segment as long as it is "recent enough" according to an application-specific coherence predicate. Consistency across segments is optional, and is enforced when desired by means of a low-cost version history hashing scheme [2]. Because they are spread around the Internet, segments must incorporate mechanisms for

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

access control and fault tolerance. They must also accommodate heterogeneous programming languages and machine architectures.

We focus in this short paper on the subject of heterogeneity. Everything depends on strong typing. Each user data type in InterWeave is declared using Sun's External Data Representation (XDR) notation [13], which we translate, automatically, into language-specific type declarations and machine-specific metadata.[1] InterWeave functions can use the metadata to translate segments and segment updates to and from a universal wire format when communicating across machines and languages.

Our work is driven in large part by collaborations with colleagues in three application domains: remote visualization and steering of scientific simulations; interactive, incremental datamining; and "intelligent environments" for human-computer collaboration. As a concrete example, consider a desktop front end for a stellar simulation, running on a remote Cashmere cluster (we are working on such an application with Prof. Adam Frank of Rochester's Department of Physics and Astronomy). To support the front end, the Cashmere cluster would create a segment in which to place relevant data:

```
IW_handle_t ss = IW_create_segment (
    "http://iw.cs.rochester.edu/\
    stellar_simulation");
IW_wl_acquire (ss);
    /* grab write lock */
stellar_data_t* sd =
    (stellar_data_t *)
    IW_named_malloc (ss,
        stellar_data_desc,
        "stellar_root");
IW_wl_release (ss);
    /* release lock */
/* Data at *sd can now be used
   in simulation */
```

The `stellar_data_t` type and the descriptor `stellar_data_desc` would be created automatically from an XDR type declaration by the InterWeave XDR compiler. The descriptor provides `IW_malloc` with the information it needs to determine the wire format and local format of the allocated data, together with the mappings between them.

The name of a segment and the name of a data structure allocated in that segment can be combined to produce a *machine independent pointer* (*mip*): `http://iw.cs.rochester.edu/stellar_simulation#stellar_root`. A remote process that obtains the mip (through static convention, message passing, a directory service, or even console I/O) can then obtain access to the data (assuming appropriate rights). In our stellar simulation example, the front end program on the researcher's desktop machine might perform the following calls:

```
stellar_data_t* sd = IW_mip_to_ptr (
    "http://iw.cs.rochester.edu/\
    stellar_simulation#stellar_root");
IW_handle_t ss = IW_get_segment (sd);
```

Handle `ss` can now be used to lock and unlock the segment. While a lock is held, pointer `sd` can be dereferenced to inspect or (with a write lock) modify the segment.

The rest of this paper is organized as follows. Section 2 enumerates the InterWeave functions that must be type aware. Section 3 then describes the data structures, tools, and techniques that make type awareness possible. We briefly summarize related work in section 4 and conclude with our status and plans in section 5.

2 Requirements

InterWeave takes the form of a collection of servers and a library linked into client processes. The client library executes in response to synchronous calls from the user program and signals from the operating system. The user calls support segment creation and destruction, dynamic memory allocation and deallocation, and synchronization based on reader-writer locks. The signals reflect page faults for the purpose of tracking modified pages. In the remainder of this section, we enumerate the operations that must deal explicitly with heterogeneous machine types.

Creating local copies of segments. When first locked by a client, a segment is copied, in full, into the client's memory. The server sends the data in a universal *wire format*, which the client must translate into local machine-specific format. Pointers that refer to data that are already locally cached must be converted to machine addresses. Pointers that refer to data that are not locally cached must be initialized in such a way that they will point to valid data when eventually dereferenced.

Creating and deleting data blocks. InterWeave provides `IW_malloc()` and `IW_free()` calls to allocate and deallocate memory within segments. Each block has a serial number, an optional symbolic name, and an address in the local memory of the creating process. The allocation and deallocation routines employ a conven-

[1] The choice of XDR is somewhat arbitrary; we could use any reasonable interface description language. We do not in fact use any of Sun's XDR tools; the InterWeave compiler is home-grown.

tional space management algorithm, but with elaborate bookkeeping data structures (described in Section 3) to support the other operations listed here. In order to accommodate heterogeneity, data structures allocated by `IW_malloc()` must be strongly typed, so that conversion from and to wire format can take place when needed.

Tracking writes. Like many S-DSM systems, InterWeave tracks writes to shared data by write-protecting pages and catching page fault signals. The signal handler creates a pristine copy (*twin*) of the page for later reference, and unprotects the page.

Creating and applying diffs. At the direction of the coherence protocol (not described here), InterWeave must send a description of recent segment changes to the segment's server. At other times (again at the direction of the coherence protocol), InterWeave must apply diffs, obtained from the server, to the local copy of a segment.

To support outgoing diffs, the `IW_malloc()` and `IW_free()` calls must keep track of newly allocated and deallocated blocks. When applying an incoming diff, the library must also be prepared to allocate or deallocate blocks created or destroyed by other clients.

When creating an outgoing diff, InterWeave can identify modified words by comparing modified pages to their twins. Unlike most other S-DSM systems, however, InterWeave must express both incoming and outgoing diffs not in terms of bytes and pages, but in terms of the segments, blocks, and (machine-independent) offsets—i.e., in wire format. Data values, including pointers, must also be converted to machine-independent form. These conversions require that the library maintain metadata containing detailed type information. To obtain the metadata, we have our XDR compiler produce type descriptors, which the user program then passes to `IW_malloc()`.

Localizing pointers. For the sake of performance and compatibility with existing compilers, we would like to ensure that pointers, both intra- and inter-segment, are represented as ordinary machine addresses whenever the data to which they refer is locally cached. Unfortunately, when a pointer itself is first cached its target may not be local. We must initialize the pointer (with back-up metadata) in such a way that it is sure to point to valid (local) data when actually dereferenced. Techniques that attempt to patch the value of the pointer itself, lazily, are undesirable, since they do not support pointer arithmetic prior to patching. Our chosen technique is described in the following section.

3 Design and Implementation

In this section, we outline the data structures and algorithms used by InterWeave to implement the functions described in the previous section.

3.1 Pointer Swizzling

Pointers in InterWeave are of two types: the *machine independent pointer* (mip) and the *machine dependent pointer*. A machine dependent pointer is whatever the client machine understands, usually a 32 or 64-bit address. Machine independent pointers are URLs of the form `domain/path#block#offset`, where `domain` is a (numeric or symbolic) internet address, `path` serves to identify the segment in the server's namespace, `block` is the serial number or symbolic name of a block within the segment, and `offset` is an optional displacement inside the block, expressed as a number of primitive datatype fields. The following are all syntactically valid mips:

```
iw.rochester.edu/test/foo#17#3276
iw.rochester.edu/test/foo#17
iw.rochester.edu/test/foo#myblock
```

When a segment is first brought into memory, or when an update is received from a segment's server, pointers contained in the segment must be converted from the machine independent URL format to a format that the user can safely dereference. This conversion is known as *pointer swizzling*.

InterWeave maintains a hash table (the *segment table*) describing segments that are locally cached. If a mip obtained from the server refers to data within a locally cached segment, InterWeave uses data structures described in the following section to generate a real pointer in local format. If the mip refers to data in a segment of which there is no local copy, InterWeave must take special action. Our chosen mechanism resembles the "expanding frontier" mechanism of dynamic (lazy) linking, or of Wilson's pointer swizzling at page fault time [14]. Given a pointer into segment A, for which we have as yet no local copy, we reserve space for A, identify the address within that space at which the referenced data will lie, set the pointer to refer to that space, but leave the space unmapped. Before a process can access data within a segment, InterWeave semantics require it to obtain a reader or writer lock. The lock acquire operations provide a natural point at which to create a local copy of the segment, and allow InterWeave to assume that any dereferenced pointer will refer to local data.

3.2 Data Structures

InterWeave presents the programmer with two granularities of shared data: *segments* and *blocks*. Blocks are the pieces of memory the user allocates (with the `IW_malloc()` call). They contain any kind of data the user wishes to put in them, and can be of arbitrary size. Every block has a serial number within its segment, assigned by `IW_malloc()`. It may also have a symbolic name, specified as an addition parameter. A segment is a named collection of blocks. There is no a priori limit on the number of blocks in a segment, and blocks within the same segment can be of different types and sizes.

Internally, in addition to segments and blocks, InterWeave keeps track of both *subsegments* and *subblocks*. Subblocks are fixed-size arrays of bytes within a block, on the order of a cache line in size. They are used by the coherence protocol, and will not be addressed in this paper. Subsegments are contiguous regions of memory that comprise the local copy of a segment. The subsegments of a given segment are not necessarily contiguous in memory (and in general are not). Subsegments support blocks of arbitrary size, allow segments to grow over time, but ensure that a given memory page contains data from only one segment.

InterWeave manages its own heap area, rather than relying on the standard C library `malloc()`. The InterWeave heap routines manage subsegments, and maintain a variety of bookkeeping information. Among other things, this information includes a set of balanced search trees that allow InterWeave to quickly locate blocks by name or serial number, to support the translation of mips into local pointers.

Figure 1 illustrates the organization of memory into subsegments, blocks, and free space. 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, one for the first free space in the segment, and two for a pair of AVL trees containing the segment's blocks. One tree is sorted by block serial number, the other by block symbolic name. The segment table entry may also include a cached TCP connection over which to reach the server.

Each subsegment contains a *next* pointer and a wordmap of modified pages. The words in the wordmap are set to point to twins, created in response to write-protect page faults. Together, the wordmap and the linked list of subsegments in a given segment allow InterWeave to quickly determine which pages need to be diffed when the coherence protocol needs to send an update to the server.

Each subsegment contains a root pointer for an AVL tree of the subsegment's blocks, sorted by memory address. In addition, each subsegment contains a pair of pointers and balance bits for inclusion in a global AVL tree of subsegments, shared by all locally cached segments, and sorted by memory order. This last tree allows the page fault handler to find the appropriate wordmap entry on a write protect fault. In addition to setting the wordmap entry, the fault handler creates a pristine twin of the page, for later use in diffing, and re-enables write access. (To reiterate: there are four separate kinds of AVL trees: a global tree of subsegments, ordered by memory address; two trees of blocks within a segment, ordered by serial number and by symbolic name; and a tree of blocks within a subsegment, ordered by memory address.)

All blocks in a subsegment begin with a collection of control information (headers). A detailed view of this information appears in figure 2. The shaded part of the figure represents part of the data in the block. The control information includes a pointer to a type descriptor, together with pointers and balance bits for the AVL trees of blocks mentioned above. The `name left` and `name right` pointers and the `block name` field are present only if the block has a symbolic name and is thus a node of the AVL tree sorted by name.

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 1) that indicates whether that block is free or not, and if it is, where it starts.

The per-segment AVL trees of blocks sorted by serial number and by name support translation from mips to local pointers. The per-subsegment tree sorted by memory address is used to facilitate diffs. As noted above, the diffing routine scans the list of subsegments of a given segment and the wordmap within each subsegment. When it identifies a group of contiguous modified pages, it performs a byte-by-byte diff of the pages and their twins (twins are found by following the pointer in the wordmap). In order to convert this byte-by-byte diff into a machine-independent (wire format) diff, the diffing routine must have access to type descriptors. It uses the AVL tree to identify the block in which the first modified byte lies, and then scans blocks linearly, converting the diff as it goes, until it runs off the end of the last contiguous modified page. When done, it returns to the wordmap and subsegment list to find the next group of contiguous modified pages.

The per-segment AVL tree of blocks sorted by serial number also supports the application of updates from

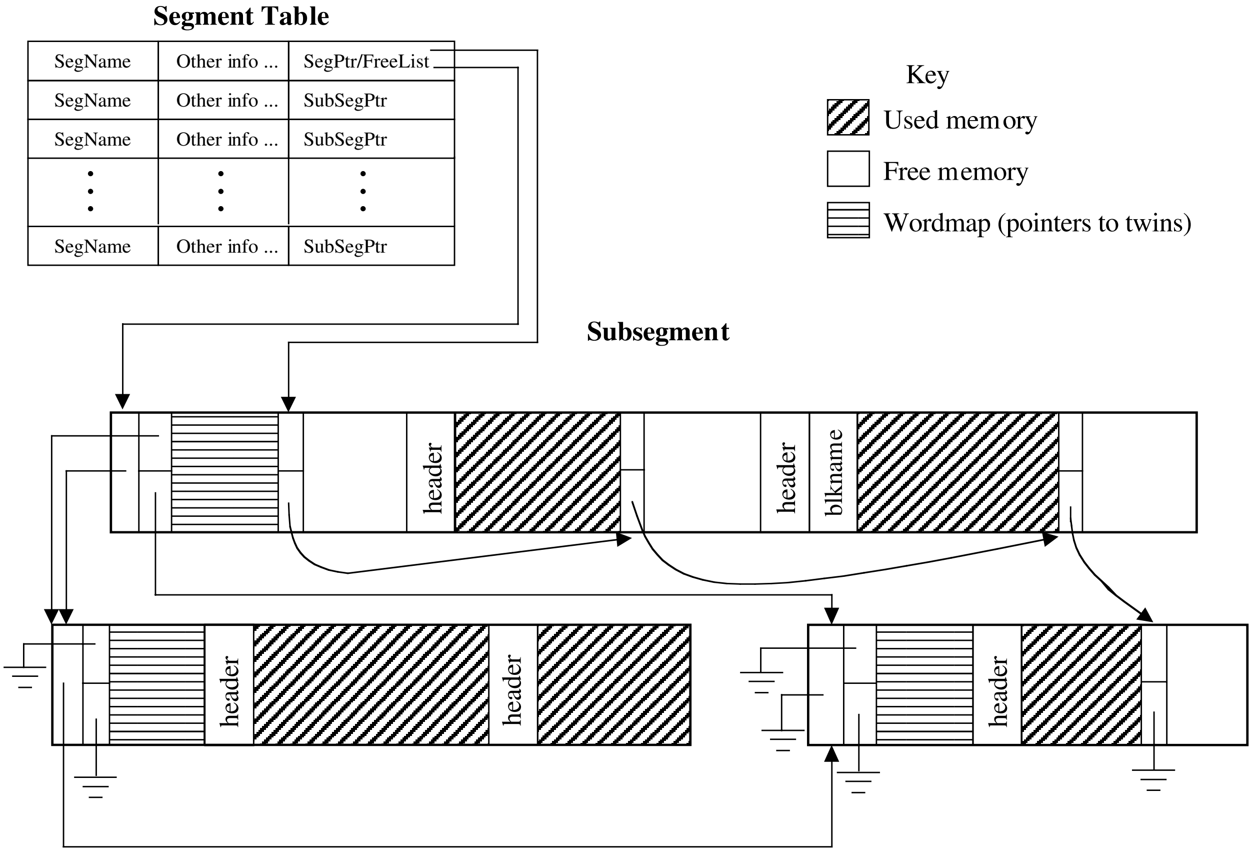

Figure 1: Simplified view of InterWeave data structures: the segment table, subsegments and blocks within segments. Blocks are dashed. White blocks are free space. Footers of blocks and free space are not shown.

the server. The wire format of the update identifies blocks by serial number. InterWeave searches for this number in the tree in order to find the actual bytes to be updated.

3.3 Wire Format

In order to be truly a heterogeneous system, InterWeave must communicate data in a generic format that can easily be converted to the appropriate local format for a given processor architecture, programming language, and compiler version. This generic format is called the *wire format* as briefly introduced above. The wire format includes both data and metadata. The metadata describes the type of each block of the data. To generate the metadata and to ensure mutually compatible type declarations in different languages, InterWeave employs a datatype compiler that accepts Sun XDR declarations [13] as input. XDR employs a strongly typed C-like syntax with well-known primitive data type sizes. When asked to produce output for C, the compiler generates a `.h` file containing type declarations and a `.c` file containing initialized variables that constitute the metadata. The InterWeave library interprets the metadata in order to translate to and from wire format.

4 Related Work

At least two early S-DSM systems, Mermaid and Agora, were designed with processor or language independence in mind. We address these in the first two subsections below. We then turn in the final subsection to a variety of other related work.

4.1 Mermaid

Mermaid [16] is a heterogeneous S-DSM system designed with the SPMD programming model in mind. Only processes belonging to the same application can share data, and there is no way for an independently initiated process to join an already running application. The data used by an application is lost when the application terminates.

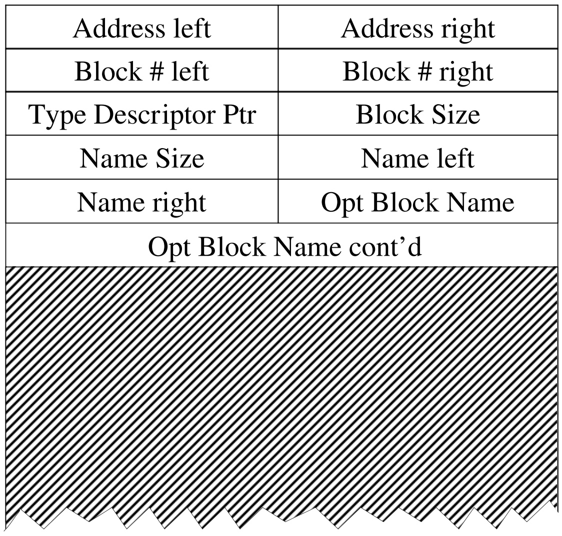

Figure 2: Description of the control structure for blocks.

Shared memory in Mermaid is strongly typed, and data is converted from the target format to the destination format when communication happens. There is no wire format, however: all data is converted directly from the sender's local format at the receiver's side. The conversion uses functions created automatically by a source compiler for each language supported by Mermaid.

The granularity of memory coherence in Mermaid is the page. (It is not clear what happens when different machines have different page sizes.) Pages are also limited to holding a single data type, making sharing significantly less general than in InterWeave. Finally, Mermaid supports only a single memory model: namely sequential consistency. InterWeave currently supports five different models [2], and we expect to allow users to define additional models.

4.2 Agora

Agora [1] is a distributed that incorporates support for shared data structures across heterogeneous machines, but with a type system and implementation significantly more restrictive than InterWeave's.

Data types in Agora are declared using a LISP-like declaration language and compiled by Agora's compiler. The resulting type descriptions are then linked into the user's application and become available for inspection whenever needed by the type conversion system. The description language, however, supports neither recursive types nor pointers.

Every shared data structure (SDS) in Agora is an instance of an abstract data type, and is accessed only through the methods of that type. Moreover calls to these methods access the SDS indirectly through a linear or hash table *map*, rather than directly with a machine address. New SDSes can be created dynamically, with a constructor method that returns an opaque handle. The handle can then be registered with a name server, using a character string name, allowing other processes to perform a lookup operation that returns its own local handle.

Once initialized, the memory used to hold the value of an SDS is never changed. Rather, update operations allocate new memory and change the local map to point to the new memory instead of the old. Updates are broadcast to all processes with a map entry for the modified SDS. The writer proceeds immediately, however, yielding a memory model reminiscent of processor consistency. As in Mermaid, there is no wire format; type conversion is done by the receiver if needed. Garbage memory blocks are reclaimed automatically when no longer in any process's map. Synchronization is based on a monitor-like mutual exclusion mechanism for update methods and a mechanism for asynchronous interprocessor events.

4.3 Other Related Work

While we believe InterWeave's support for distributed shared state to be uniquely general and transparent, much of the underlying technology is borrowed from previous work of others.

RPC and message-passing systems have used interface description languages to accommodate heterogeneous machine types for at least 20 years [15]. Popular modern systems include Sun's XDR [13], Microsoft's DCOM [10], and various flavors of CORBA [7].

Herlihy's thesis work [4] pioneered deep copy semantics for linked data structures. Similar facilities can be found in DCOM, in Emerald [5], and in the more recent "pickling" (serialization) of Java [9].

Wilson's page-fault-based pointer swizzling [14] dates from 1991. Similar mechanisms can be found in dynamic (lazy) linkers [3], and in Lisp systems (e.g. LOOM [6]) for limited-address space machines.

Several other groups have recently begun to explore the notion of shared state for processes spread across the Internet. References to many of these can be found in our to-appear paper at LCR 2000 [2].

5 Status and Future Work

As of April 2000, we have implemented most of the coherence mechanisms of InterWeave, most of which were inherited from our earlier InterAct [8] project. We have also completed the InterWeave XDR compiler, and are currently implementing the remaining mechanisms described in section 3. The know-how to implement fault handling, twinning, and diffing is available from our previous work on Cashmere [12]. Concurrent with the implementation of InterWeave, we are rewriting a message-based distributed action game to use the InterWeave API. We hope to verify that the code becomes significantly simpler while retaining acceptable performance.

Once our prototype is up and running, we expect to integrate InterWeave with Cashmere, to support remote front ends for visualization and steering of high-end simulations (joint work with colleagues in Physics and Astronomy, Chemistry, and Laser Energetics). We also plan to address issues of particular importance for Internet-based applications: security (user authentication, message encryption) and fault tolerance in particular.

References

[1] R. Bisiani and A. Forin. Multilanguage Parallel Programming of Heterogeneous Machines. *IEEETC*, 37(8):930–945, AUG 1988.

[2] D. Chen, S. Dwarkadas, S. Parthasarathy, E. Pinheiro, and M. L. Scott. InterWeave: A Middleware System for Distributed Shared State (extended abstract). In *PROC of the 5TH LCR*, Rochester, NY, May 2000.

[3] R. A. Gingell, J. P. Moran, and W. A. Shannon. Virtual Memory Architecture in SunOS. In *PROC of the USENIX Summer '87 Technical CONF*, pages 81–94, JUN 1987.

[4] M. Herlihy and B. Liskov. A Value Transmission Method for Abstract Data Types. *TOPLAS*, 4(4):527–551, OCT 1982.

[5] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-Grained Mobility in the Emerald System. *TOCS*, 6(1):109–133, FEB 1988. Originally presented at the *11TH SOSP*, NOV 1987.

[6] T. Kaehler. Virtual Memory on a Narrow Machine for an Object-Oriented Language. In *OOPSLA '86 CONF PROC*, pages 87–106, Portland, OR, SEP – OCT 1986.

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

[8] S. Parthasarathy and S. Dwarkadas. InterAct: Virtual Sharing for Interactive Client-Server Applications. In *4TH LCR*, MAY 1998.

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

[10] D. Rogerson. *Inside COM*. Microsoft Press, Redmond, Washington, JAN 1997.

[11] R. Stets, S. Dwarkadas, L. I. Kontothanassis, U. Rencuzogullari, and M. L. Scott. The Effect of Network Total Order, Broadcast, and Remote-Write Capability on Network-Based Shared Memory Computing. In *PROC of the 6TH HPCA*, Toulouse, France, JAN 2000.

[12] 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 SOSP*, St. Malo, France, OCT 1997.

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

[14] P. R. Wilson. Pointer Swizzling at Page Fault Time: Efficiently Supporting Huge Address Spaces on Standard Hardware. *CAN*, 19(4):6–13, JUN 1991.

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

[16] S. Zhou, M. Stumm, K. Li, and D. Wortman. Heterogeneous Distributed Shared Memory. In *IEEETPDS*, pages 540–554, 1992.