Support for Machine and Language Heterogeneity in a Distributed Shared State System *

Chunqiang Tang, DeQing Chen,
Sandhya Dwarkadas, and Michael L. Scott

Technical Report #783

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

June 2002

Abstract

InterWeave is a distributed middleware system that supports the sharing of strongly typed, pointer-rich data structures across heterogeneous platforms. Unlike RPC-style systems (including DCOM, CORBA, Java RMI), InterWeave does not require processes to employ a procedural interface: it allows them to access shared data using ordinary reads and writes. To save bandwidth in wide area networks, InterWeave caches data locally, and employs *two-way diffing* to maintain coherence and consistency, transmitting only the portions of the data that have changed.

In this paper, we focus on the aspects of InterWeave specifically designed to accommodate heterogeneous machine architectures and languages. Central to our approach is a strongly typed, platform-independent wire format for diffs, and a set of algorithms and metadata structures that support translation between local and wire formats. Using a combination of microbenchmarks and real applications, we evaluate the performance of our heterogeneity mechanisms, and compare them to comparable mechanisms in RPC-style systems. When transmitting entire segments, InterWeave achieves performance comparable to that of RPC, while providing a more flexible programming model. When only a portion of a segment has changed, InterWeave's use of diffs allows it to scale its overhead down, significantly outperforming straightforward use of RPC.

1 Introduction

With the rapid growth of the Internet, more and more applications are being developed for (or ported to) wide area networks (WANs), in order to take advantage of resources available at distributed sites. One inherent property of the Internet is its heterogeneity in terms of architectures, operating

*This work was supported in part by NSF grants CCR-9988361, CCR-0204344, and EIA-0080124; by equipment grants from Compaq, IBM, Intel, and Sun; and by the U.S. Department of Energy Office of Inertial Confinement Fusion under Cooperative Agreement No. DE-FC03-92SF19460.

systems, programming languages, computational power, and network bandwidth. Conventionally, programming in the face of such heterogeneity has depended either on application-specific messaging protocols or on higher-level remote procedure call systems such as RPC [4], DCOM [6], CORBA [28], or Java RMI [33].

RPC-style systems have proven successful in a wide variety of programs and problem domains. They have some important limitations, however, and their wide acceptance does not necessarily imply that there is no alternative approach to addressing heterogeneity. In this paper we consider the possibility of constructing distributed applications—even wide-area distributed applications—using a shared-memory programming model rather than messages or remote invocation. We focus in particular on how such a programming model can accommodate heterogeneity.

For many distributed applications, especially those constructed from large amounts of (sequential) legacy code, or from parallel code originally developed for small to mid-size multiprocessors, we believe that shared memory constitutes the most natural communication model. Similar beliefs have spurred the development, over the past fifteen years, of a large number of systems for software distributed shared memory (S-DSM) on local-area clusters [2, 22]. While these systems have almost invariably assumed a homogeneous collection of machines, and while they have been motivated by a desire for parallel speedup rather than distribution, we nonetheless draw inspiration from them, and adopt several of their key techniques.

A common concern with S-DSM is its performance relative to explicit messaging [24]: S-DSM systems tend to induce more underlying traffic than does hand-written message-passing code. At first glance one might expect the higher message latencies of a WAN to increase the performance gap, making shared memory comparatively less attractive. We argue, however, that the opposite may actually occur. Communication in a distributed system is typically more coarse-grained than it is in a tightly coupled local-area cluster, enabling us to exploit optimizations that have a more limited effect in the tightly coupled world, and that are difficult or cumbersome to apply by hand. These optimizations include the use of variable-grain size coherence blocks [30]; bandwidth reduction through communication of *diffs* [8] instead of whole objects; and the exploitation of relaxed coherence models to reduce the frequency of updates [11, 38].

In comparison to S-DSM, RPC systems do not always provide the semantics or even the performance that programmers want, particularly for applications involving references. RPC and RMI transmit pointer-rich data structures using *deep copy* [17] semantics: they recursively expand all data referenced by pointers, and bundle the expansion into messages. CORBA allows real *object references* to be passed as arguments, but then every access through such a reference becomes an expensive cross-domain (potentially cross-WAN) call. The following example illustrates the dilemma faced by the user of a typical RPC system.

Suppose a linked list is shared among a collection of clients. If searches are frequent, encapsulating the list in an object at a central server is likely to result in poor performance [12, 20]. The alternative is to cache the list at every client and to maintain coherence among the copies, either manually in the application or with the help of object caching middleware [20]. But then when one client updates the list, e.g. by inserting an item at the head, the entire list will typically be propagated to every client due to deep copy message semantics. In addition to inducing significant performance overhead, the copy operation poses semantic problems: clients receiving an update now have two copies of the list. If they are to discard the old copy, what should become of local references that point to discarded but semantically unmodified items?

Pointers are not the only problem with existing RPC systems. Object granularity is another. Suppose the above example is carefully (if inconveniently) rewritten to avoid the use of pointers. When one field of an item in the list is modified by a client, the entire item will typically need to be propagated to all other clients, even if other items do not. If items are large, the update may still waste significant network bandwidth and message-processing time.

An attractive alternative, we believe, is to provide a common infrastructure for *distributed shared state* across heterogeneous distributed machines. This infrastructure can be made entirely compatible with RPC/RMI systems, supporting genuine reference parameters as an efficient alternative to deep-copy value parameters. Programmers who prefer a shared memory programming model can access shared variables directly. Programmers who prefer an object-oriented programming model can use shared memory inside a class implementation to obtain the performance benefits of diffing and relaxed coherence without the need to reimplement these mechanisms in every application. Programmers who prefer the implicit synchronization of traditional message passing can migrate, when appropriate, to a hybrid programming model in which messages can contain references to automatically managed shared variables.

In order to simplify the construction of heterogeneous distributed systems, while providing equal or improved performance, we have developed a system known as InterWeave [11]. InterWeave allows programs written in multiple languages to map shared *segments* into their address space, regardless of location or machine type, and to access the data in those segments transparently once mapped. In C, operations on shared data, including pointers, take precisely the same form as operations on non-shared data.

To support heterogeneous architectures and languages, InterWeave employs a type system based on a machine- and language-independent interface description language (IDL).[1] When communicating between machines, InterWeave converts data to and from a common wire format, assisted by type descriptors generated by our IDL compiler. The wire format is rich enough to capture diffs for complex data structures, including pointers and recursive types, in a platform-independent manner. The translation mechanism also provides a hook for spatial locality: when creating the initial copy of a segment on a client machine, InterWeave uses diffing history to choose a layout that maximizes locality of reference.

When translating and transmitting equal amounts of data, InterWeave achieves throughput comparable to that of standard RPC packages, and 20 times faster than Java RMI. When only a fraction of the data have been changed, the use of diffs allows InterWeave's translation cost and bandwidth requirements to scale down proportionally, while those of RPC-style systems remain unchanged.

We describe the design of InterWeave in more detail in Section 2. We provide implementation details in Section 3, and performance results in Section 4. We compare our design to related work in Section 5, and conclude with a discussion of status and plans in Section 6.

2 InterWeave Design

The InterWeave programming model assumes a distributed collection of servers and clients. Servers maintain persistent copies of shared data, and coordinate sharing among clients. Clients in turn must

[1] InterWeave's IDL is currently based on Sun XDR, but this is not an essential design choice. InterWeave could be easily modified to work with other IDLs.

```
node_t *head;
IW_handle_t h;
void list_init(void) {
    h = IW_open_segment("host/list");
    head = IW_mip_to_ptr("host/list#head");
}

node_t *list_search(int key) {
    node_t *p;
    IW_rl_acquire(h);      // read lock
    for (p = head->next; p; p = p->next) {
        if(p->key == key) {
            IW_rl_relese(h);    // read unlock
            return p;
        }
    }
    IW_rl_release(h);           // read unlock
    return NULL;
}

void list_insert(int key) {
    node_t *p;
    IW_wl_acquire(h);           // write lock
    p = (node_t*) IW_malloc(h, IW_node_t);
    p->key = key;
    p->next = head->next;
    head->next = p;
    IW_wl_release(h);           // write unlock
}
```

Figure 1: Shared linked list in InterWeave. Variable `head` points to an unused header node; the first real item is in `head->next`.

be linked with a special InterWeave library, which arranges to map a cached copy of needed data into local memory. InterWeave servers are oblivious to the programming languages used by clients, but the client libraries may be different for different programming languages. The client library for Fortran, for example, cooperates with the linker to bind InterWeave data to variables in common blocks (Fortran 90 programs may also make use of pointers).

Figure 1 presents a simple realization of the linked list example of Section 1. The InterWeave API used in the example is explained in more detail in the following sections. For consistency with the example we present the C version of the API. Similar versions exist for C++, Java, and Fortran.

InterWeave is designed to interoperate with our Cashmere S-DSM system [32]. Together, these systems integrate hardware coherence and consistency within multiprocessors (*level-1* sharing), S-DSM within tightly coupled clusters (*level-2* sharing), and version-based coherence and consistency across the Internet (*level-3* sharing). At level 3, InterWeave uses application-specific knowledge of minimal coherence requirements to reduce communication, and maintains consistency information in a manner that scales to large amounts of shared data. Further detail on InterWeave's coherence and consistency mechanisms can be found in other papers [11].

2.1 Data Allocation

The unit of sharing in InterWeave is a self-descriptive *segment* (a heap) within which programs allocate strongly typed *blocks* of memory.[2] Every segment is specified by an Internet URL. The blocks within a segment are numbered and optionally named. By concatenating the segment URL with a block name or number and optional offset (delimited by pound signs), we obtain a *machine-independent pointer (MIP)*: "`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.

Every 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. Assuming appropriate access rights, `IW_open_segment()` communicates with the appropriate server to open an existing segment or to create a new one if the segment does not yet exist. The call returns an opaque *handle* that can be passed as the initial argument in calls to `IW_malloc()`. InterWeave currently employs public key based authentication and access control. If requested by the client, communication with a server can optionally be encrypted with DES.

As in multi-language RPC systems, the types of shared data in InterWeave must be declared in an IDL. The InterWeave IDL compiler translates these declarations into the appropriate programming language(s) (C, C++, Java, Fortran). It also creates initialized *type descriptors* that specify the layout of the types on the specified machine. The descriptors must be registered with the InterWeave library prior to being used, and are passed as the second argument in calls to `IW_malloc()`. These conventions allow the library to translate to and from wire format, ensuring that each type will have the appropriate machine-specific byte order, alignment, etc. in locally cached copies of segments.

Synchronization (to be discussed further in Section 2.2) takes the form of reader-writer locks. A process must hold a writer lock on a segment in order to allocate, free, or modify blocks. The lock routines take a segment handle as parameter.

Given a pointer to a block in an InterWeave segment, or to data within such a block, a process can create a corresponding MIP:

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

This MIP can then be passed to another process through a message, a file, or an argument of a remote procedure in RPC-style systems. 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 unless and until the segment is locked.

[2] Like distributed file systems and databases, and unlike systems such as PerDiS [14], InterWeave requires manual deletion of data; there is no automatic garbage collection.

It should be emphasized that `IW_mip_to_ptr()` is primarily a bootstrapping mechanism. Once a process has one pointer into a data structure (e.g. the `head` pointer in our linked list example), any data reachable from that pointer can be directly accessed in the same way as local data, even if embedded pointers refer to data in other segments. InterWeave's pointer-swizzling and data-conversion mechanisms ensure that such pointers will be valid local machine addresses. It remains the programmer's responsibility 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, and to determine whether that segment is already locked:

```
IW_handle_t h = IW_get_handle(p);
IW_lock_status s = IW_get_lock_status(h);
```

Much of the time we expect that programmers will know, because of application semantics, that pointers about to be dereferenced refer to data in segments that are already locked.

2.2 Coherence

When modified by clients, InterWeave segments move over time through a series of internally consistent states. When writing a segment, a process must have exclusive write access to the most recent version. When reading a segment, however, the most recent version may not be required because processes in distributed applications can often accept a significantly more relaxed—and hence less communication-intensive—notion of coherence. InterWeave supports several relaxed coherence models. It also supports the maintenance of causality among segments using a scalable hash-based scheme for conflict detection.

When a process first locks a shared segment (for either read or write access), the InterWeave library obtains a copy from the segment's server. At each subsequent read-lock acquisition, the library checks to see whether the local copy of the segment is "recent enough" to use. If not, it obtains an update from the server. An adaptive polling/notification protocol [11] often allows the client library to avoid communication with the server when updates are not required. Twin and diff operations [8], extended to accommodate heterogeneous data formats, allow the implementation to perform an update in time proportional to the fraction of the data that has changed.

The server for a segment need only maintain a copy of the segment's most recent version. 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 remembers, for each block, the version number of the segment in which that block was last modified. This information allows the server to avoid transmitting copies of blocks that have not changed. As partial protection against server failure, InterWeave periodically checkpoints segments and their metadata to persistent storage. The implementation of real fault tolerance is a subject of future work.

As noted at the beginning of Section 2, an S-DSM-style system such as Cashmere can play the role of a single InterWeave node. Within an S-DSM cluster, or within a hardware-coherent node, data can be shared using data-race-free [1] shared memory semantics, so long as the cluster or node holds the appropriate InterWeave lock.

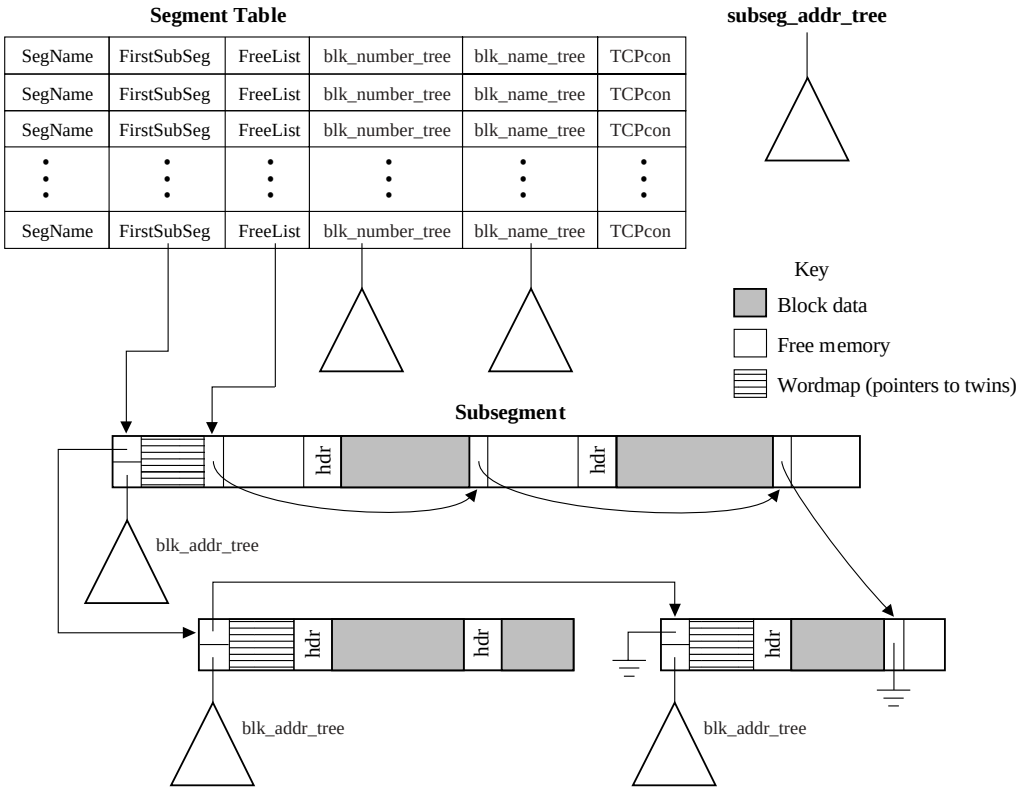

Figure 2: Simplified view of InterWeave client-side data structures: the segment table, subsegments, and blocks within segments. Type descriptors, pointers from balanced trees to blocks and subsegments, and footers of blocks and free space are not shown.

3 Implementation

3.1 Memory Management and Segment Metadata

Client Side. As described in Section 2, InterWeave presents the programmer with two granularities of shared data: *segments* and *blocks*. Each block must have a well-defined type, but this type can be a recursively defined structure of arbitrary complexity, so blocks can be of arbitrary size. Every block has a serial number within its segment, assigned by `IW_malloc()`. It may also have an optional symbolic name, specified as an additional 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.

The copy 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 therefore implement a segment as a collection of *subsegments*, invisible to the user. Each subsegment is contiguous, and can be any integral number of pages in length. These conventions support blocks of arbitrary size, and ensure that any given page contains data from only one segment. New subsegments can be allocated by the library dynamically, allowing a segment to expand over time.

An InterWeave client 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 collection of balanced search trees that allow InterWeave to quickly locate blocks by name, serial number, or address.

Figure 2 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 balanced trees containing the segment's blocks. One tree is sorted by block serial number (*blk_number_tree*), the other by block symbolic name (*blk_name_tree*); together they support translation from MIPs to local pointers. An additional global tree contains the subsegments of all segments, sorted by address (*subseg_addr_tree*); this tree supports modification detection and translation from local pointers to MIPs. Each subsegment has a balanced tree of blocks sorted by address (*blk_addr_tree*). Segment table entries may also include a cached TCP connection over which to reach the server. 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 flags for optimizations to be described in Section 3.5. 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 2) that indicates whether that block is free and, if it is, where it starts.

Server Side. To avoid an extra level of translation, an InterWeave server stores both data and type descriptors in wire format. The server keeps track of segments, blocks, and *subblocks*. The latter are invisible to clients.

Each segment maintained by a server has an entry in a segment hash table keyed by segment name. Each block within the segment consists of a version number (the version of the segment in which the block was most recently modified), a serial number, a pointer to a type descriptor, pointers to link the block into data structures described in the following paragraph, a pointer to a wire-format block header, and a pointer to the data of the block, again in wire format.

The blocks of a given segment are organized into a balanced tree sorted by serial number (*svr_blk_number_tree*) and a linked list sorted by version number (*blk_version_list*). The linked list is separated by markers into sublists, each of which contains blocks with the same version number. Markers are also organized into a balanced tree sorted by version number (*marker_version_tree*). Pointers to all these data structures are kept in the segment table, along with the segment name.

Each block is potentially divided into subblocks, comprising a contiguous group of primitive data elements (fixed at 16 for our experiments, but which can also vary in number based on access patterns) from the same block. Subblocks also have version numbers, maintained in an array. Subblocks and pointers to the various data structures in the segment table are used in collecting and applying diffs (to be described in Section 3.2). In order to avoid unnecessary data relocation, machine-independent pointers and strings are stored separately from their blocks, since they can be of variable size.

```
blk_diff    -> blk_serial_number
               blk_diff_len
               prim_RLEs
prim_RLEs   -> prim_RLE prim_RLEs
            ->
prim_RLE    -> prim_start
               num_prims
               prims
prims       -> prim prims
            ->
prim        -> primitive data
            -> MIP
MIP         -> string
            -> string
               desc_serial_number
```

Figure 3: Grammar for wire-format block diffs.

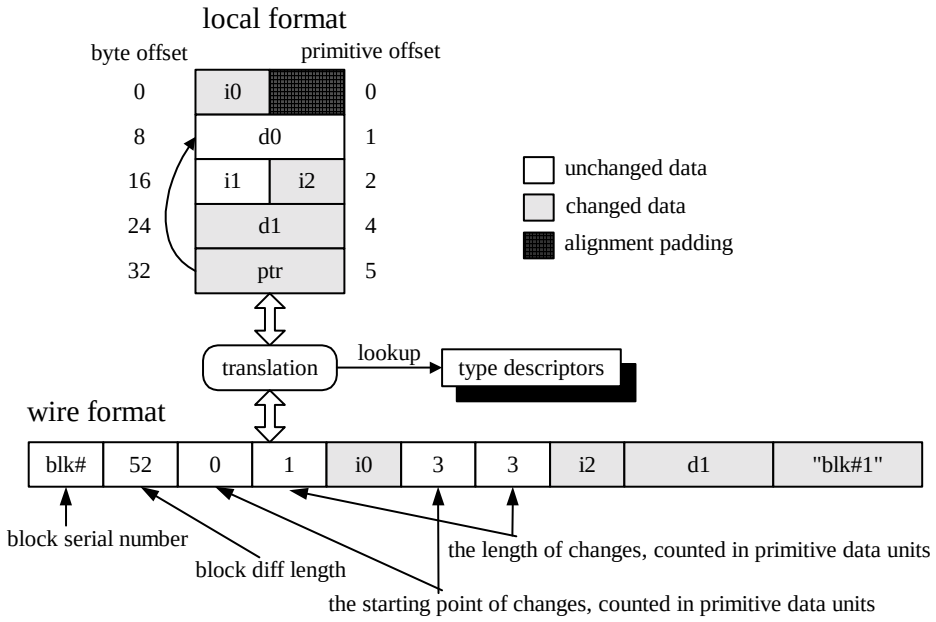

Figure 4: Wire format translation of a structure consisting of three integers (`i0`–`i2`), two doubles (`d0`–`d1`), and a pointer (`ptr`). All fields except `d0` and `i1` are changed.

3.2 Diff Creation and Translation

Client Side. When a process acquires a write lock on a given segment, the InterWeave library asks the operating system to write protect the pages that comprise the various subsegments of the local copy of the segment. When a page fault occurs, the `SIGSEGV` signal handler, installed by the library at program startup time, creates a pristine copy, or *twin* [8], of the page in which the write fault occurred. It saves a pointer to that twin in the faulting segment's *wordmap* for future reference, and then asks the operating system to re-enable write access to the page.

When a process releases a write lock, the library uses diffing to gather changes made locally and convert them into machine-independent wire format in a process called *collecting* a diff. Figure 4 shows an example of wire format translation. The changes are expressed in terms of segments, blocks, and primitive data unit offsets, rather than pages and bytes. A wire-format block diff (Figure 3) consists of a block serial number, the length of the diff, and a series of run length encoded data changes, each of which consists of the starting point of the change, the length of the change, and the updated content. Both the starting point and length are measured in primitive data units rather than in bytes. The MIP in the wire format is either a string (for an intra-segment pointer) or a string accompanied by the serial number of the pointed-to block's type descriptor (for a cross-segment pointer).

The diffing routine must have access to type descriptors in order to compensate for local byte order and alignment, and in order to swizzle pointers. The content of each descriptor specifies the substructure and layout of its type. For primitive types (integers, doubles, etc.) there is a single pre-defined code. For other types there is a code indicating either an array, a record, or a pointer, together with pointer(s) that recursively identify the descriptor(s) for the array element type, record field type(s), or pointed-at type. The type descriptors also contain information about the size of blocks and the number of primitive data units in blocks. For structures, the descriptor records both the *byte offset* of each field from the beginning of the structure in local format, and the machine-independent *primitive offset* of each field, measured in primitive data units. Like blocks, type descriptors have segment-specific serial numbers, which the server and client use in wire-format messages. A given type descriptor may have different serial numbers in different segments. Per-segment arrays and hash tables maintained by the client library map back and forth between serial numbers and pointers to local, machine-specific descriptors. An additional, global, hash table maps wire format descriptors to addresses of local descriptors, if any. When an update message from the server announces the creation of a block with a type that has not appeared in the segment before, the global hash table allows InterWeave to tell whether the type is one for which the client has a local descriptor. If there is no such descriptor, then the new block can never be referenced by the client's code, and can therefore be ignored.

When translating local modifications into wire format, the diffing routine takes a series of actions at three different levels: subsegments, blocks, and words. At the subsegment level, it scans the list of subsegments of the segment and the wordmap within each subsegment. When it identifies a modified page, it searches the *blk_addr_tree* within the subsegment to identify the block containing the beginning of the modified page. It then scans blocks linearly, converting each to wire format by using *block-level diffing*. When it runs off the end of the last contiguous modified page, the diffing routine returns to the wordmap and subsegment list to find the next modified page.

At the block level, the library uses *word-level diffing* (word-by-word comparison of modified pages and their twins) to identify a *run* of continuous modified words. Call the bounds of the run, in words, *change_begin* and *change_end*. Using the pointer to the type descriptor stored in the block header, the library searches for the primitive subcomponent spanning change_begin, and computes the primitive offset of this subcomponent from the beginning of the block. Consecutive type descriptors, from change_begin to change_end, are then retrieved sequentially to convert the run into wire format. At the end, word-level diffing is called again to find the next run to translate.

To ensure that translation cost is proportional to the size of the modified data, rather than the size of the entire block, we must avoid walking sequentially through subcomponent type descriptors

while searching for the primitive subcomponent spanning change_begin. First, the library subtracts the starting address of the block from change_begin to get the byte offset of the target subcomponent. Then it searches recursively for the target subcomponent. For structures, a binary search is applied to byte offsets of the fields. For an array, the library simply divides the given byte offset by the size of an individual element (stored in the array element's type descriptor).

To accommodate reference types, InterWeave relies on pointer swizzling [36]. There are two kinds of pointers, intra-segment pointers and cross-segment pointers. For a cross-segment pointer, it is possible that the pointer points to a not (yet) cached segment or that the pointed-to segment is cached but the version is outdated, and does not yet contain the pointed-to block. In both cases, the pointer is set to refer to reserved space in 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: a core of currently accessible pages surrounded by a "fringe" of reserved but not-yet accessible pages. In wire format, a cross-segment pointer is accompanied by the serial number of the pointed-to block's type descriptor, from which the size of the reserved space can potentially be derived without contacting the server.

To swizzle a local pointer to a MIP, the library first searches the *subseg_addr_tree* for the subsegment spanning the pointed-to address. It then searches the *blk_addr_tree* within the subsegment for the pointed-to block. It subtracts the starting address of the block from the pointed-to address to obtain the byte offset, which is then mapped to a corresponding primitive offset with the help of the type descriptor stored in the block header. Finally, the library converts the block serial number and primitive offset into strings, which it concatenates with the segment name to form a MIP.

When a client acquires a read lock and determines that its local cached copy of the segment is not recent enough, it asks the server to build a 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 *applying* a diff. Diff application is similar to diff collection, except that searches now are based on primitive offsets rather than byte offsets. To convert the serial numbers employed in wire format to local machine addresses, the client traverses the balanced tree of blocks, sorted by serial number, that is maintained for every segment.

To swizzle a MIP into a local pointer, the library parses the MIP and extracts the segment name, block serial number or name, and the primitive offset, if any. The segment name is used as an index into the segment hash table to obtain a pointer to the segment's *blk_number_tree*, which is in turn searched to identify the pointed-to block. The byte offset of the pointed-to address is computed from the primitive offset with the help of the type descriptor stored in the block header. Finally, the byte offset is added to the starting address of the block to get the pointed-to address.

Server Side. Each server maintains an up-to-date copy of each segment it serves, and controls access to the segments.

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 (some portion of) the content of the block or subblock was most recently modified. This convention strikes a balance between the size of server-to-client diffs and the size of server-maintained metadata. The server also remembers, for each block, the version number in which the block was created. Finally, the server maintains a list of the serial numbers of deleted blocks along with the versions in which they were deleted. The creation version number allows the server to tell, when sending a segment update to a client, whether the

metadata for a given block need to be sent in addition to the data. When *malloc*ing a new block, a client can use any available serial number. Any client with an old copy of the segment in which the number was used for a different block will receive new metadata from the server the next time it obtains an update.

Upon receiving a diff, the server first appends a new marker to the end of the *blk_version_list* and inserts it in the *marker_version_tree*. Newly created blocks are then appended to the end of the list. Modified blocks are first identified by search in the *svr_blk_number_tree*, and then moved to the end of the list. If several blocks are modified together in both the previous diff and the current diff, their positions in the linked list will be adjacent, and moving them together to the end of the list only involves a constant-cost operation to reset pointers into and out of the first and last blocks, respectively.

At the time of a lock acquire, a client must decide whether its local copy of the segment needs to be updated. (This decision may or may not require communication with the server; see below.) If an update is required, the server traverses the *marker_version_tree* to locate the first marker whose version is newer than the client's version. All blocks in the *blk_version_list* after that marker have some subblocks that need to be sent to the client. The server constructs a wire-format diff and returns it to the client. Because version numbers are remembered at the granularity of blocks and (when blocks are too large) subblocks, all of the data in each modified block or subblock will appear in the diff. To avoid extra copying, the modified subblocks are sent to the client using `writev()`, without being compacted together.

The client's type descriptors, generated by our IDL compiler, describe machine-specific data layout. At the server, however, we store data in wire format. When it receives a new type descriptor from a client, the server converts it to a machine-independent form that describes the layout of fields in wire format. Diff collection and application are then similar to the analogous processes on the client, except that searches are guided by the server's machine-independent type descriptors.

3.3 Support for Heterogeneous Languages

Independent of its current implementation, InterWeave establishes a universal data sharing framework by defining a protocol for communication, coherence, and consistency between clients and servers, along with a machine- and language-independent wire format. Any language implemented on any platform can join the sharing as long as it can provide a client library that conforms to the protocol and the wire format. Library implementations may differ from one another in order to take advantage of features within a specific language or platform. Our current implementation employs a common back-end library with front ends tailored to each language. Our current set of front ends supports C, C++, Fortran, and Java; support for additional languages is planned.

For Fortran 77 the lack of dynamic memory allocation poses a special problem. InterWeave addresses this problem by allowing processes to share static variables (this mechanism also works in C). 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 descriptors (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.

Java introduces additional challenges for InterWeave, including dynamic allocation, garbage collection, constructors, and reflection. Our J-InterWeave system is described in a companion paper [10].

While providing a shared memory programming model, InterWeave focuses on pure data sharing. Methods in object-oriented languages are currently not supported. When sharing with a language with significant semantic limitations, the programmer may need to avoid certain features. For example, pointers are allowed in segments shared by a Fortran 77 client and a C client, but the Fortran 77 client has no way to use them.

3.4 Portability

InterWeave currently consists of approximately 31,000 lines of heavily commented C++ code: 14,000 lines for the client library, 9,000 lines for the server, an additional 5,000 lines shared by the client library and the server, and 2,700 lines for the IDL compiler. This total does not include the J-InterWeave Java support, or the (relatively minor) modifications required to the Cashmere S-DSM system to make it work with InterWeave. Both the client library and server have been ported to a variety of architectures (Alpha, Sparc, x86, and MIPS) and operating systems (Windows NT, Linux, Solaris, Tru64 Unix, and IRIX). Applications can be developed and run on arbitrary combinations of these. The Windows NT port is perhaps the best indication of InterWeave's portability: only about 100 lines of code needed to be changed.

3.5 Optimizations

Several optimizations improve the performance of InterWeave in important common cases. We describe here those related to memory management and heterogeneity. Others are described in other papers [11, 9].

Data layout for cache locality. InterWeave's knowledge of data types and formats allows it to organize blocks in memory for the sake of spatial locality. When a segment is cached at a client for the first time, blocks that have the same version number, meaning they were modified by another client in a single write critical section, are placed in contiguous locations, in the hope that they may be accessed or modified together by this client as well. We do not currently relocate blocks in an already cached copy of a segment. InterWeave has sufficient information to find and fix all pointers within segments, but references held in local variables would have to be discarded or updated manually.

Diff caching. The server maintains a cache of diffs that it has received recently from clients, or collected recently itself, in response to client requests. These cached diffs can often be used to respond to future requests, avoiding redundant collection overhead. In most cases, a client sends the server a diff, and the server caches and forwards it in response to subsequent requests. The server updates its own master copy of the data asynchronously when it is not otherwise busy (or when it needs a diff it does not have). This optimization takes server diff collection off the critical path of the application in common cases.

No-diff mode. As in the TreadMarks S-DSM system [3], a client that repeatedly modifies most of the data in a segment will switch to a mode in which it simply transmits the whole segment to the server at every write lock release. This *no-diff* mode eliminates the overhead of `mprotect`s, page faults, and the creation of twins and diffs. Moreover, translating an entire block is more efficient than translating diffs. The library can simply walk linearly through the type descriptors of primitive subcomponents and put each of them in wire format, avoiding searches for `change_begin` and checks for `change_end`. The switch occurs when the client finds that the size of a newly created diff is at least 75% of the size of the segment itself; this value strikes a balance between communication cost and the other overheads of twinning and diffing. Periodically afterwards (at linearly increasing random intervals), the client will switch back to diffing mode to verify the size of current modifications. When `malloc`s and `free`s force us to diff a mostly modified segment, we still try to put individual blocks into no-diff mode.

Isomorphic type descriptors. For a given data structure declaration in IDL, our compiler outputs a type descriptor most efficient for run-time translation rather than strictly following the original type declaration. For example, if a struct contains 10 consecutive integer fields, the compiler generates a descriptor containing a 10-element integer array instead. This altered type descriptor is used only by the InterWeave library, and is invisible to the programmer; the language-specific type declaration always follows the structure of the IDL declaration.

Cache-aware diffing. A blocking technique is used during word-level diffing to improve the temporal locality of data accesses. Specifically, instead of diffing a large block in its entirety and then translating the diff into wire format, we diff at most 1/4 of the size of the hardware cache, and then translate this portion of the diff while it is still in the cache. The ratio 1/4 accommodates the actual data, the twin, and the wire-format diff (all of which are about the same size), and additional space for type descriptors and other meta-data.

Diff run splicing. In word-level diffing, if one or two adjacent words are unchanged while both of their neighboring words are changed, we treat the entire sequence as changed in order to avoid starting a new run length encoding section in the diff. It already costs two words to specify a head and a length in the diff, and the spliced run is faster to apply.

Last-block searches. On both the client and the server, block predictions are used to avoid searching the balanced tree of blocks sorted by serial number when mapping serial numbers in wire format to blocks. After handling a changed block, we predict that the next changed block in the diff will be the block following the current block in the previous diff. Due to repetitive program patterns, blocks modified together in the past tend to be modified together in the future.

4 Performance Results

The results presented here were collected on a 500MHz Pentium III machine, with 256MB of memory, a 16KB L1 cache, and a 512KB L2 cache, running Linux 2.4.18. Earlier results for other machines can be found in a technical report [9].

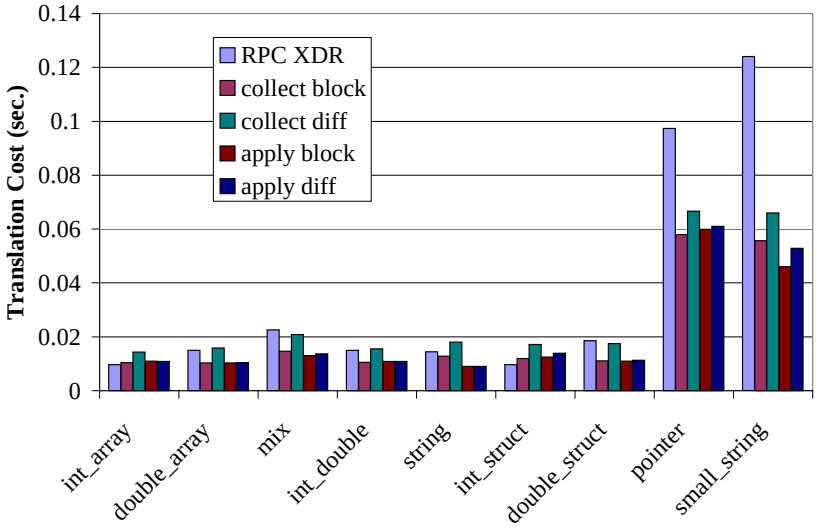

Figure 5: Client's cost to translate 1MB of data.

4.1 Microbenchmarks

Basic Translation Costs

Figure 5 shows the overhead required to translate various data structures from local to wire format and vice versa. In each case we arrange for the total amount of data to equal 1MB; what differs is data formats and types. The `int_array` and `double_array` cases comprise a large array of integers or doubles, respectively. The `int_struct` and `double_struct` cases comprise an array of structures, each with 32 integer or double fields, respectively. The `string` and `small_string` cases comprise an array of strings, each of length 256 or 4 bytes, respectively. The `pointer` case comprises an array of pointers to integers. The `int_double` case comprises an array of structures containing integer and double fields, intended to mimic typical data in scientific programs. The `mix` case comprises an array of structures containing integer, double, string, small string, and pointer fields, intended to mimic typical data in non-scientific programs such as calendars, CSCW, and games.

All optimizations described in Section 3.5 are enabled in our experiments. All provide measurable improvements in performance, bandwidth, or both. Data layout for cache locality, isomorphic type descriptors, cache-aware diffing, diff run splicing, and last block searches are evaluated separately in Section 4.2.

Depending on the history of a given segment, the *no-diff* optimization may choose to translate a segment in its entirety or to compute diffs on a block-by-block basis. The *collect diff* and *apply diff* bars in Figure 5 show the overhead of translation to and from wire format, respectively, in the block-by-block diff case; the *collect block* and *apply block* bars show the corresponding overheads when diffing has been disabled.

For comparison purposes we also show the overhead of translating the same data via RPC parameter marshaling, in stubs generated with the standard Linux `rpcgen` tool. Unmarshaling costs (not shown) are nearly identical.

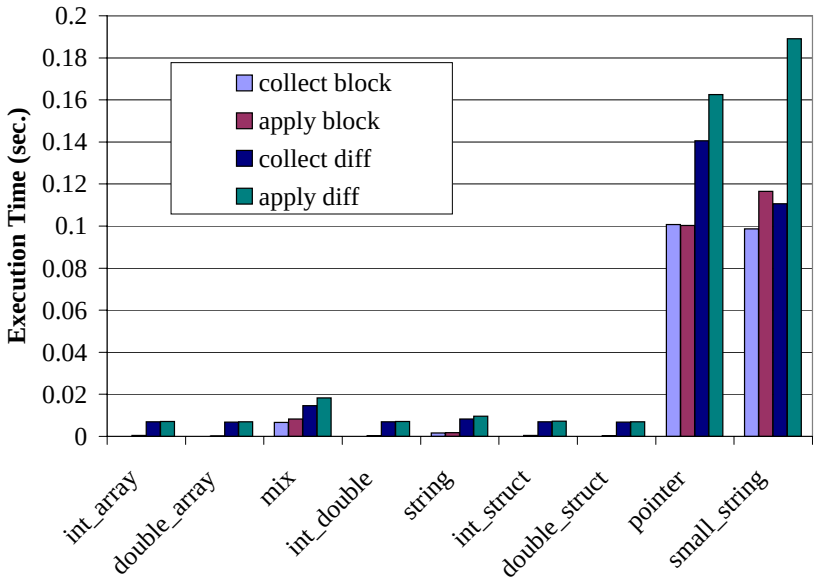

Figure 6: Server's cost to translate 1MB of data.

Generally speaking, InterWeave overhead is comparable to that of RPC. `Collect block` and `apply block` are 25% faster than RPC on average; `collect diff` and `apply diff` are 8% faster. It is clear that RPC is not good at marshaling pointers and small strings. Excluding these two cases, InterWeave in *no diff* (collect/apply block) mode is still 18% faster than RPC; when diffing has to be performed, InterWeave is 0.5% slower than RPC. `Collect block` is 39% faster than `collect diff` on average, and `apply block` is 4% faster than `apply diff`, justifying the use of the *no diff* mode.

When RPC marshals a `pointer`, deep copy semantics require that the pointed-to data, an integer in this experiment, be marshaled along with the pointer. The size of the resulting RPC wire format is the same as that of InterWeave, because MIPs in InterWeave are strings, longer than 4 bytes. The RPC overhead for structures containing doubles is high in part because `rpcgen` does not inline the marshaling routine for doubles.

Figure 6 shows translation overhead for the InterWeave server in the same experiment as above. Because the server maintains data in wire format, costs are negligible in all cases other than `pointer` and `small_string`. In these cases the high cost stems from the fact that strings and MIPs are of variable length, and are stored separately from their wire format blocks. However, for data structures such as `mix`, with a more reasonable number of pointers and small strings, the server cost is still low. As noted in Section 3.5, the server's diff management cost is not on the critical path in most cases.

Comparisons between InterWeave and Java RMI appear in a companion paper [10]. The short story: translation overhead under the Sun JDK1.3.2 JVM is 20X that of J-InterWeave.

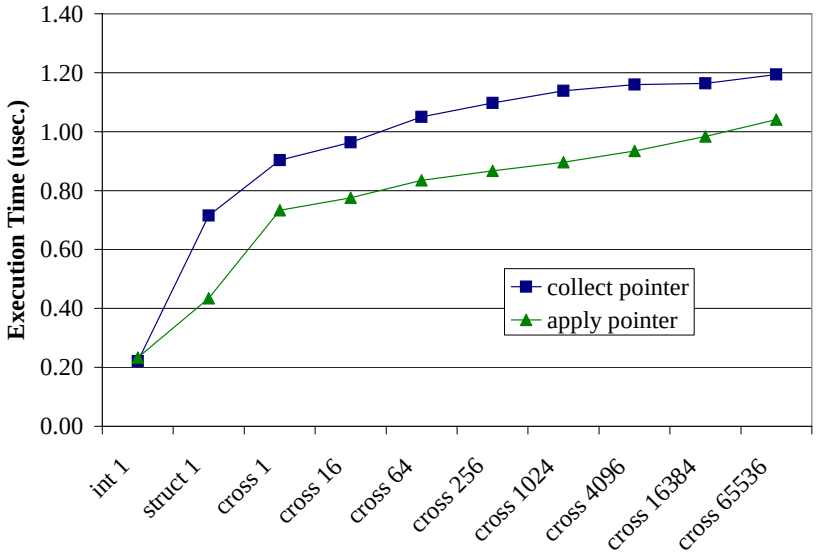

Figure 7: Pointer swizzling cost as a function of pointed-to object type.

Pointer Swizzling

Figure 7 shows the cost of swizzling ("collect pointer") and unswizzling ("apply pointer") a pointer. This cost varies with the nature of the pointed-to data. The `int 1` case represents an intra-segment pointer to the start of an integer block. `struct 1` is an intra-segment pointer to the middle of a structure with 32 fields. The `cross` $\#n$ cases are cross-segment pointers to blocks in a segment with n total blocks. The modest rise in overhead with n reflects the cost of search in various metadata trees. Performance is best in the `int 1` case, which we expect to be representative of the most common sorts of pointers. However, even for moderately complex cross-segment pointers, InterWeave can swizzle about one million of them per second.

Figure 8 shows the breakdown for swizzling a `cross 1024` local pointer into a MIP, where `searching block` is the cost of searching the pointed-to block by first traversing the $subseg\_addr\_tree$ and then traversing the blk_addr_tree; `computing offset` is the cost of converting the byte offset between the pointed-to address and the start of the pointed-to block into a corresponding primitive offset; `string conversion` is the cost of converting the block serial number and primitive offset into strings and concatenating them with the segment name; and `searching desc_srlnum` is the cost of retrieving the serial number of the pointed-to block's type descriptor from the type descriptor hash table.

Similarly, Figure 9 shows the breakdown for swizzling a `cross 1024` MIP into a local pointer, where `read MIP` is cost of reading the MIP from wire format; `parse MIP` is the cost of extracting the segment name, block serial number, and primitive offset from the MIP; `search segment` is the cost of locating the pointed-to segment by using the hash table keyed by segment name; `search block` is the cost of finding the block in the blk_number_tree of the pointed-to segment; and `offset to pointer` is the cost of mapping the primitive offset into a byte offset between the pointed-to address and the start of the pointed-to block. As seen in Figures 8 and 9, pointer swizzling is a complex process where no single factor is the main source of overhead.

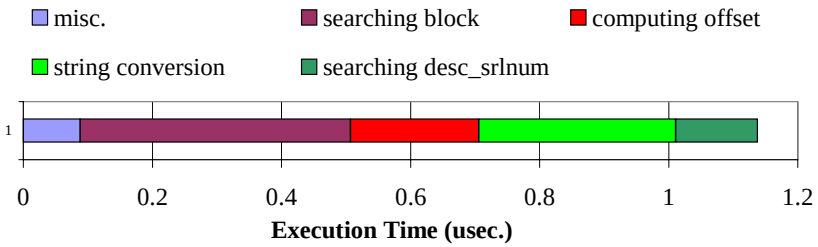

Figure 8: Breakdown of swizzling local pointers into MIPs (using `cross 1024`).

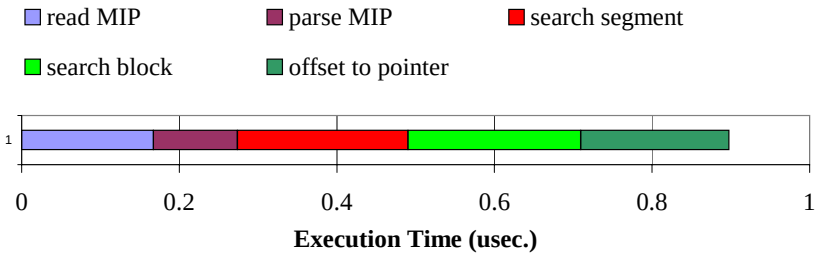

Figure 9: Breakdown of swizzling MIPs into local pointers (using `cross 1024`).

Modifications at Different Granularities

Figure 10 shows the client and the server diffing cost as a function of the fraction of a segment that has changed. In all cases the segment in question consists of a 1MB array of integers. The x axis indicates the distance in words between consecutive modified words. Ratio 1 indicates that the entire block has been changed. Ratio 4 indicates that every 4th word has been changed.

The `client collect diff` cost has been broken down into `client word diffing`—

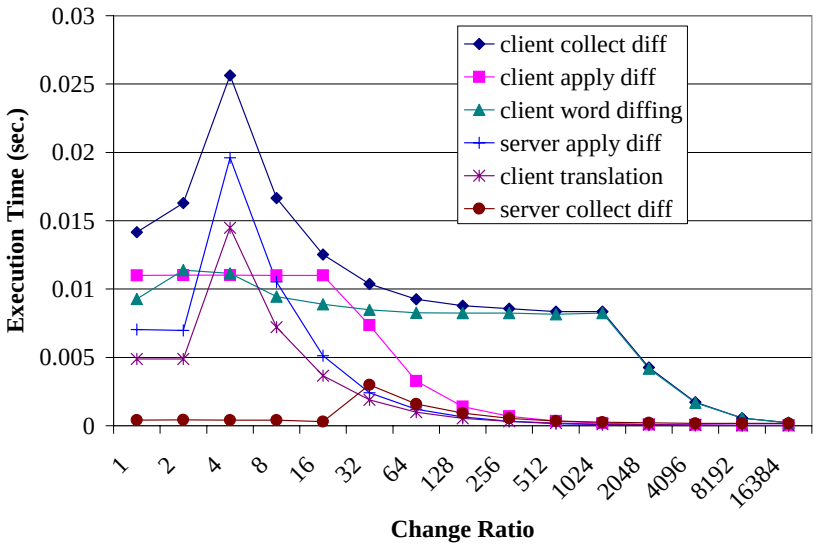

Figure 10: Diff management cost as a function of modification granularity (1MB total data).

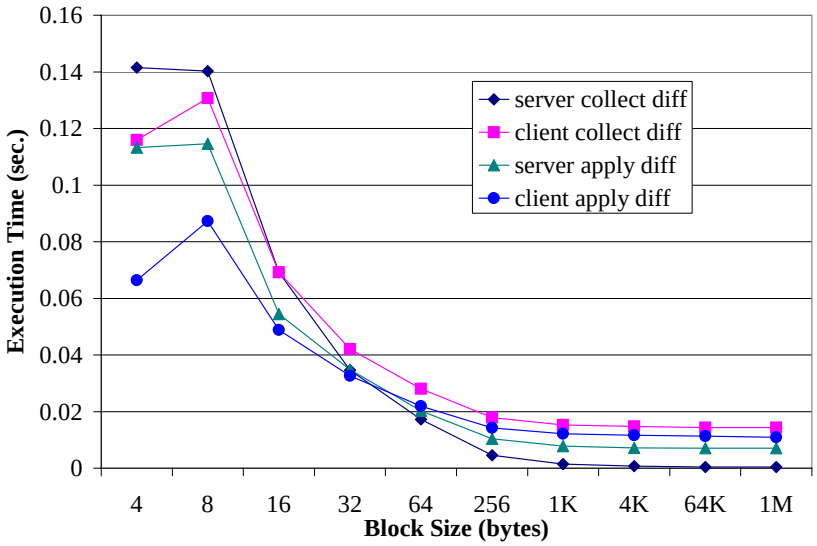

Figure 11: The effect of varying block size.

word-by-word comparison of a page and its twin—and `client translation`—converting the diff to wire format. (Values on these two curves add together to give the values on the `client collect diff` curve.) There is a sharp knee for word diffing at ratio 1024. Before that point, every page in the segment has been modified; after that point, the number of modified pages decreases linearly. Due to the artifact of subblocks (16 primitive data units in our current implementation), the `server collect diff` and `client apply diff` costs are constant for ratios between 1 and 16, because in those cases the server loses track of fine grain modifications and treats the entire block as changed. The jump in `client collect diff`, `server apply diff`, and `client translation` between ratios of 2 and 4 is due to the loss of the *diff run splicing* optimization described in Section 3.5. At a ratio of 2 the entire block is treated as changed, while at a ratio of 4 the block is partitioned into many small isolated changes. The cost for `word diffing` increases between ratios of 1 and 2 because the diffing is more efficient when there is only one continuous changed section.

Figures 5 and 6 show that InterWeave is efficient at translating entirely changed blocks. Figure 10 shows that InterWeave is also efficient at translating scattered modifications. When only a fraction of a block has changed, InterWeave is able to reduce both translation cost and required bandwidth by transmitting only the diffs. With straightforward use of an RPC-style system, both translation cost and bandwidth remain constant regardless of the fraction of the data that has changed.

Varying Block Size and Number of Blocks

Figure 11 shows the overhead introduced on the client and the server while varying the size of blocks and the number of blocks in a segment. Each point on the x axis represents a different segment configuration, denoted by the size of a single block in the segment. For all configurations, the total size of all blocks in the segment is 1MBytes. In the `4K` configuration, for example, there are 256 blocks in the segment, each of size 4K. On both the client and the server, the curves flatten

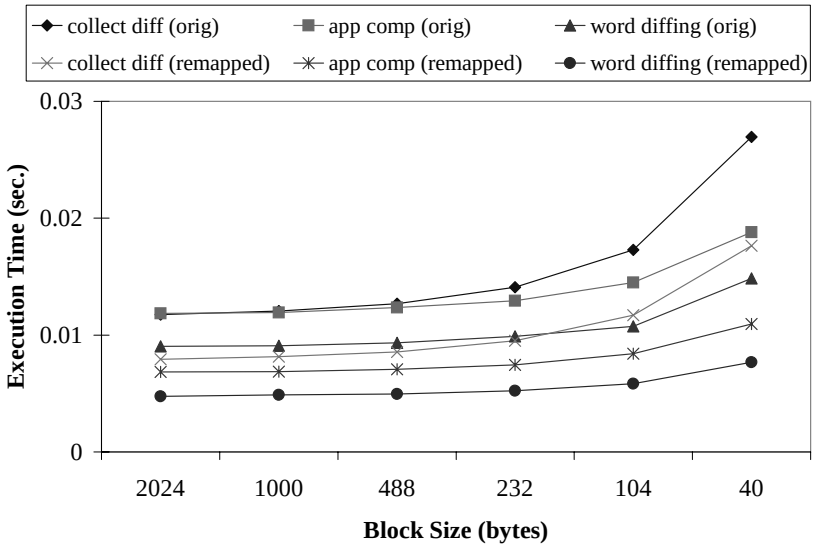

Figure 12: Performance impact of version based block layout, with 1MB total data.

out between `64` and `256`. Blocks of size 4 bytes are primitive blocks, a special case InterWeave can recognize and optimize accordingly.

4.2 Optimizations

Data Layout for Cache Locality

Figure 12 demonstrates the effectiveness of version-based block layout. As in Figure 11, several configurations are evaluated here, each with 1MB total data, but with different sizes of blocks. There are two versions for each configuration: `orig` and `remapped`. For the `orig` version, every other block in the segment is changed; for the `remapped` version, those changed blocks, exactly half of the segment, are grouped together. As before, the cost for *word diffing* is part of the cost in *collect diff*. Because fewer pages and blocks need to be traversed, the saving in the `remapped` version is significant. `App comp.` is simply the time to write all words in the modified blocks, and is presented to evaluate the locality effects of the layout on a potential computation. The savings in `app comp.` is due to fewer page faults, fewer TLB misses, and better spatial locality of blocks in the cache.

Cache-Aware Diffing and Diff Run Splicing

Figure 13 shows the effectiveness of cache-aware diffing and diff run splicing, where `none` means neither of the two techniques is applied; `cache` means only cache-aware diffing is applied; `merge` means only diff run splicing is applied; and `cache-merge` means both are applied. On average, diff run splicing itself degrades performance by 1%, but it is effective for `double_array` and `double_struct` as noted in Section 3.5. Cache-aware diffing itself improves performance by only 12%. The two techniques are most effective when combined, improving performance by 20% on average.

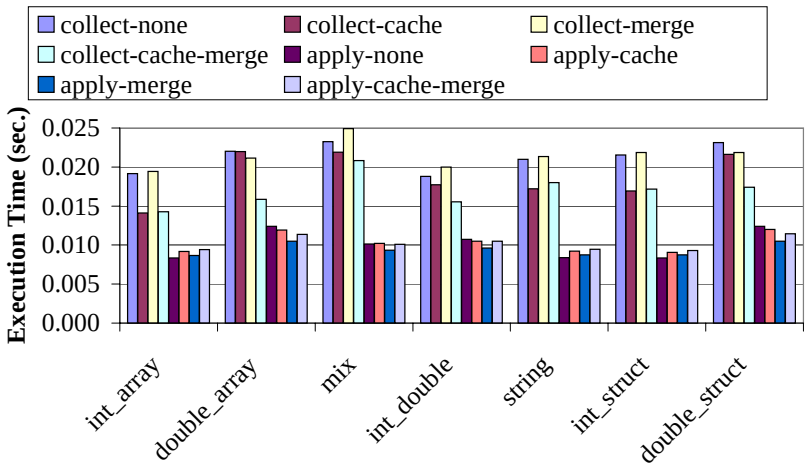

Figure 13: Cache-aware diffing and diff run splicing.

Isomorphic Type Descriptors

Figure 14 shows the potential performance improvement obtained by our IDL compiler when several adjacent fields of the same size in a structure can be merged into a single field as an array of elements. `Int_struct` from Figure 5 is used in this experiment. The x axis shows the number of adjacent fields merged by our IDL compiler. For instance, the data point for `2` on the x axis means the 32 fields of the structure are merged into 16 fields, each of which is a two element array. For `apply diff`, the small increase from no merging to merging two adjacent fields is due to the overhead in handling small, 2-element, arrays. This overhead also exists for `collect diff`, but the saving due to the simplification in finding `change_begin` is more significant, so the total cost still goes down. The savings in `collect diff` and `collect block` are more substantial than those in `apply diff` and `apply block`. On average, merging 32 fields into a single field improves performance by 51%.

Last Block Searches

Figure 15 demonstrates the effectiveness of block prediction. The setting for this experiment is the same as that in Figure 11, but we show the number of blocks on the x axis here rather than the size of blocks. `Predict 100` indicates the `apply diff` costs at the client or the server when block prediction is 100% correct; `predict 50` indicates the costs when prediction is 50% correct; `no predict` indicates the costs when no prediction is implemented. As shown in the figure, the reduction in block searching costs is significant even at just 50% accuracy, when there is a large number of blocks. When the number of blocks is medium, say less than or equal to 4K, other costs dominate, and the saving due to block prediction is not significant.

4.3 Translation Costs for a Datamining Application

In an attempt to validate the microbenchmark results presented in the Section 4.1, we have measured translation costs in a locally developed datamining application. The application performs incremen-

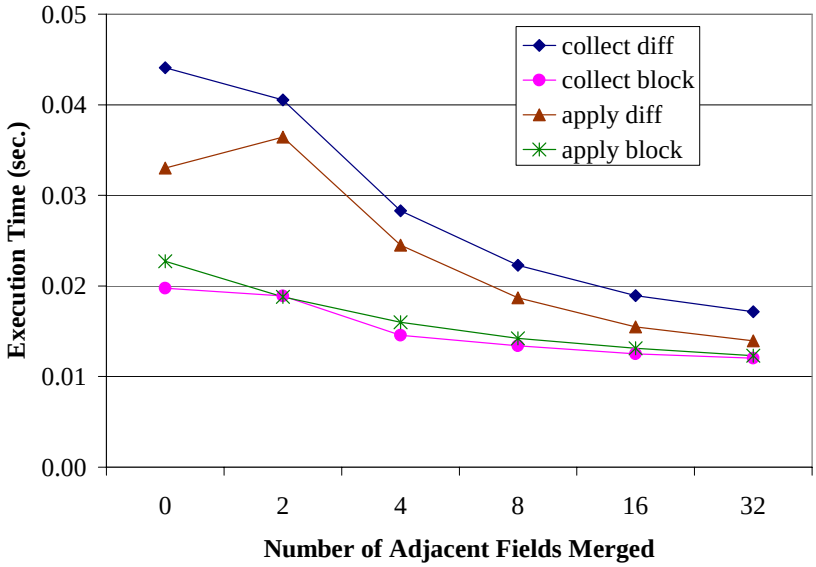

Figure 14: Isomorphic type descriptors

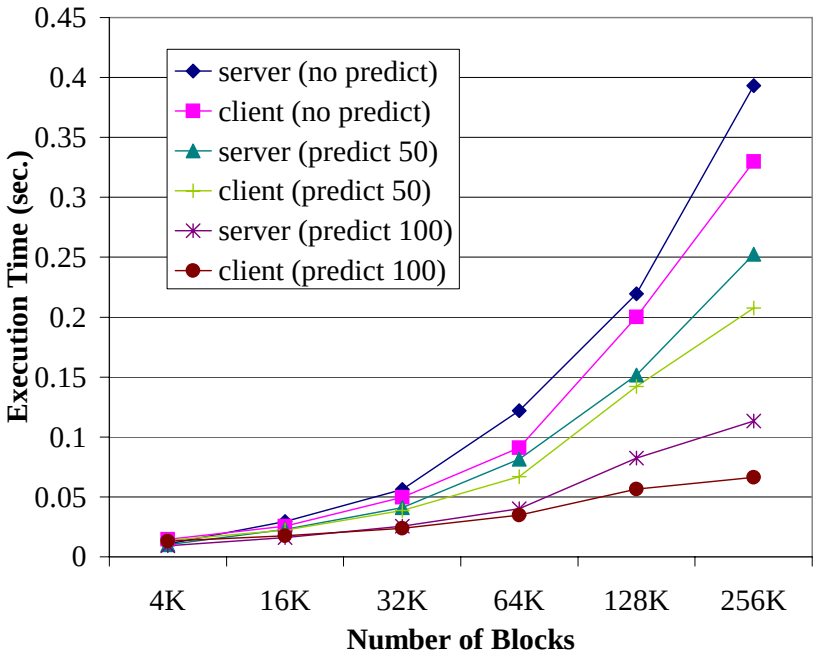

Figure 15: Block prediction.

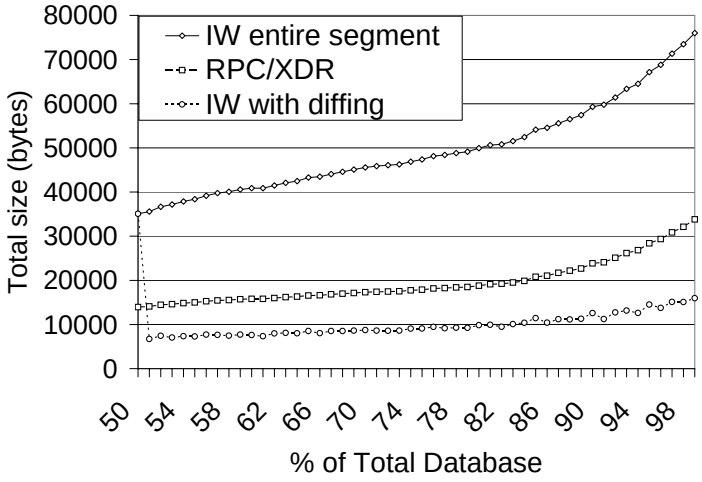

Figure 16: Wire format length of datamining segment under InterWeave and RPC.

tal sequence mining on a remotely located database of *transactions* (e.g., retail purchases). Details of the application are described elsewhere [11].

Our sample database is generated by tools from IBM research [31]. It includes 100,000 customers and 1000 different items, with an average of 1.25 transactions per customer and a total of 5000 item sequence patterns of average length 4. The total database size is 20MB.

In our experiments, we have a database server and a datamining client. Both are InterWeave clients. The database server reads from an active, growing database and builds a summary data structure (a *lattice* of item sequences) to be used by mining queries. Each node in the lattice represents a potentially meaningful sequence of transactions, and contains pointers to other sequences of which it is a prefix. This summary structure is shared between the database server and mining client in a single InterWeave segment. Approximately 1/3 of the space in the local-format version of the segment is consumed by pointers.

The summary structure is initially generated using half the database. The server then repeatedly updates the structure using an additional 1% of the database each time. As a result the summary structure changes slowly over time.

To compare InterWeave translation costs to those of RPC, we also implemented an RPC version of the application. The IDL definitions for the two versions (InterWeave and RPC) are identical.

Figure 16 compares the translated wire format length between InterWeave and RPC. Points on the x axis indicate the percentage of the entire database that has been constructed at the time the summary data structure is transmitted between machines. The middle curve represents the RPC version of the application. The other curves represent InterWeave when sending the entire data structure (upper) or a diff from the previous version (lower). The roughly 2X increase in space required to represent the entire segment in InterWeave stems from the use of character-string MIPS and the wire-format meta-data for blocks such as block serial numbers. The blocks in the segment are quite small, ranging from 8 bytes to 28 bytes. When transferring only diffs, however, InterWeave enjoys a roughly 2X space *advantage* in this application.

Figure 17 presents corresponding data for the time overhead of translation. When the whole segment needs to be translated, InterWeave takes roughly twice as long as RPC, due to the high cost

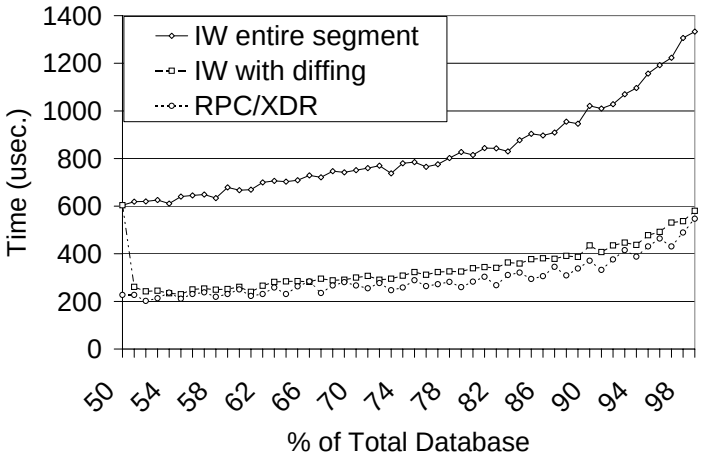

Figure 17: Translation time for datamining segment under InterWeave and RPC.

of pointers. When transferring only diffs, however, the costs of the two versions are comparable.

The high cost of pointer translation here does not contradict the data presented in the previous subsections. There the pointer translation cost for RPC included the cost of translating pointed-to data, while for InterWeave we measured only the cost of translating the pointers themselves. Here all pointers refer to data that are internal to the summary data structure, and are translated by both versions of the application. This data structure represents the potential worst case for InterWeave.

Though it is beyond the scope of this paper, we should also note that substantial additional savings are possible in InterWeave by exploiting relaxed coherence [11]. Because the summary data structure is statistical in nature, it does not have to keep completely consistent with the master database at every point in time. By sending updates only when the divergence exceeds a programmer-specified bound, we can decrease overhead dramatically. Comparable savings in the RPC version of the application would require new hand-written code.

4.4 Ease of Use

We have implemented several applications on top of InterWeave, in addition to the datamining application mentioned in Section 4.3. One particularly interesting example is a stellar dynamics code called Astroflow [16], developed by colleagues in the department of Physics and Astronomy, and modified by our group to take advantage of InterWeave's ability to share data across heterogeneous platforms.

Astroflow is a computational fluid dynamics system used to study the birth and death of stars. The simulation engine is written in Fortran, and runs on a cluster of four AlphaServer 4100 5/600 nodes under the Cashmere [32] S-DSM system. As originally implemented it dumps its results to a file, which is subsequently read by a visualization tool written in Java and running on a Pentium desktop. We used InterWeave to connect the simulator and visualization tool directly, to support on-line visualization and steering. The changes required to the two existing programs were small and isolated. We wrote an IDL specification to describe the shared data structures and replaced the original file operations with access to shared segments. No special care is required to support

multiple visualization clients. Moreover the visualization front end can control the frequency of updates from the simulator simply by specifying a temporal bound on relaxed coherence [11].

Performance experiments [11] indicate that InterWeave imposes negligible overhead on the existing simulator. More significantly, we find the qualitative difference between file I/O and InterWeave segments to be compelling in this application. We also believe the InterWeave version to be dramatically simpler, easier to understand, and faster to write than a hypothetical version based on application-specific messaging. Our experience changing Astroflow from an off-line to an on-line client highlighted the value of middleware that hides the details of network communication, multiple clients, and the coherence of transmitted data.

5 Related Work

InterWeave finds context in an enormous body of related work—far too much to document thoroughly in this paper. We attempt to focus here on the most relevant systems in the literature.

Toronto's Mermaid system [39] allowed objects to be shared across more than one type of machine, but required that all data in the same VM page be of the same type and that objects be of the same size on all machines, with the same byte offset for every subcomponent.

CMU's Agora system [5] supported sharing among more loosely-coupled processes, but in a significantly more restricted fashion than in InterWeave. Pointers and recursive types were not supported, and all shared data had to be accessed indirectly through a local mapping table.

Stardust [7] supports both message passing and a shared memory programming model on heterogeneous machines, but users must manually supply type descriptors. Recursive data structures and pointers are not supported.

Mneme [26] combines programming language and database features in a persistent programming language. As in InterWeave, persistent data are stored at the server and could be cached at clients. However, objects in Mneme are untyped byte streams, and references inside an object are identified by user supplied routines rather than the runtime system.

Rthread [13] is a system capable of executing `pthread` programs on a cluster of heterogeneous machines. It enforces a shared object model, in which remote data can only be accessed through read/write primitives. Pointers are not supported in shared global variables.

Weems et al. [35] provided a survey of languages supporting heterogeneous parallel processing. Among them, Delirium [25] is the only one adopting a shared memory programming model and it requires that users list explicitly all variables that each routine might destructively modify.

Many S-DSM systems, including Munin [8], TreadMarks [2], and Cashmere [32], have used (machine-specific) diffs to propagate updates, but only on homogeneous platforms. Several projects, including ShareHolder /citeshareholder, Globus [15], and WebOS [34], use URL-like names for distributed objects or files.

Interface description languages date from Xerox Courier [37] and related systems of the early 1980s. Precedents for the automatic management of pointers include Herlihy's thesis work [17], LOOM [18], and the more recent "pickling" (serialization) of Java [29].

Smart RPC [19] is an extension to conventional RPC that allows argument passing using call-by-reference rather than deep copy call-by-value. Smart RPC lacks a shared global name space,

however, with a well-defined cache coherence model. Where InterWeave supports extensive cache reuse, Smart RPC invalidates the cache after each RPC session. Krishnaswamy and Haumacher [21] describe a fast implementation of Java RMI capable of caching objects to avoid serialization and retransmission.

Object oriented databases (OODBs) such as Thor [23] allow objects to be cached at client front ends, but they usually neither address heterogeneity problems nor attempt to provide a shared memory programming model.

PerDiS [14] is a persistent distributed object system featuring object caching, transactions, security, and distributed garbage collection. It does not support heterogeneous languages, however, and has only a single coherence model. ScaFDOCS [20] is an object caching framework built on top of CORBA. As in Java RMI, shared objects are derived from a base class and their *writeToString* and *readFromString* methods are used to serialize and deserialize internal state. CASCADE [12] is a distributed caching service, structured as a CORBA object. The designers of both [20] and [12] report the problem with CORBA's reference model described in Section 1: every access through the CORBA reference is an expensive cross-domain call.

LBFS [27] is a low bandwidth network file system that saves bandwidth by taking advantage of commonality *between* files; InterWeave saves bandwidth by taking advantage of commonality among *versions* of a segment.

6 Conclusions and Future Work

We have described the design and implementation of a middleware system, InterWeave, that allows processes to access shared data transparently using ordinary reads and writes. InterWeave is, to the best of our knowledge, the first system to fully support shared memory across heterogeneous machine types and languages. Key to our work is a wire format, and accompanying algorithms and metadata, rich enough to capture machine- and language-independent diffs of complex data structures, including pointers or recursive data types. In a challenge to conventional wisdom, we argued that S-DSM techniques may actually improve the performance of distributed applications, while simultaneously simplifying the programming process. InterWeave is compatible with existing RPC and RMI systems, for which it provides a global name space in which data structures can be passed by reference.

We are actively collaborating with colleagues in our own and other departments to employ InterWeave in three principal application domains: remote visualization and steering of high-end simulations, incremental interactive data mining, and human-computer collaboration in richly instrumented physical environments. We are also using InterWeave as an infrastructure for research in efficient data dissemination across the Internet, and in the partitioning of applications across mobile and wired platforms.

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] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *Computer*, 29(2):18–28, Feb. 1996.

[3] C. Amza, A. Cox, S. Dwarkadas, and W. Zwaenepoel. Software DSM Protocols that Adapt between Single Writer and Multiple Writer. In *Proc. of the 3rd Intl. Symp. on High Performance Computer Architecture*, San Antonio, TX, Feb. 1997.

[4] A. D. Birrell and B. J. Nelson. Implementing Remote Procedure Calls. *ACM Trans. on Computer Systems*, 2(1):39–59, Feb. 1984.

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

[6] N. Brown and C. Kindel. *Distributed Component Object Model Protocol — DCOM/1.0*. Microsoft Corporation, Redmond, WA, Nov. 1996.

[7] G. Cabillic and I. Puaut. Stardust: An Environment for Parallel Programming on Networks of Heterogeneous Workstations. *Journal of Parallel and Distributed Computing*, 40(1):65–80 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] D. Chen, C. Tang, X. Chen, S. Dwarkadas, and M. L. Scott. Beyond S-DSM: Shared State for Distributed Systems. TR 744, Computer Science Dept., Univ. of Rochester, Mar. 2001.

[10] D. Chen, C. Tang, S. Dwarkadas, and M. L. Scott. JVM for a Heterogeneous Shared Memory System. In *Proc. of the Workshop on Caching, Coherence, and Consistency (WC3 '02)*, New York, NY, June 2002. Held in conjunction with the *16th ACM Intl. Conf. on Supercomputing*.

[11] D. Chen, C. Tang, X. Chen, S. Dwarkadas, and M. L. Scott. Multi-level Shared State for Distributed Systems. In *Proc. of the 2002 Intl. Conf. on Parallel Processing*, Vancouver, BC, Canada, Aug. 2002.

[12] G. Chockler, D. Dolev, R. Friedman, and R. Vitenberg. Implementing a Caching Service for Distributed CORBA Objects. In *Proc., Middleware 2000*, pages 1–23, New York, NY, Apr. 2000.

[13] B. Dreier, M. Zahn, and T. Ungerer. Parallel and Distributed Programming with Pthreads and Rthreads. In *Proc. of the 3rd Intl. Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS '98)*, pages 34–40, 1998.

[14] 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.

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

[16] A. Frank, G. Delamarter, R. Bent, and B. Hill. AstroFlow Simulator. Available at `http://astro.pas.rochester.edu/~delamart/Research/Astroflow/Astroflow.html`.

[17] 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.

[18] 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.

[19] K. Kono, K. Kato, and T. Masuda. Smart Remote Procedure Calls: Transparent Treatment of Remote Pointers. In *Proc. of the 14th Intl. Conf. on Distributed Computing Systems*, pages 142–151, Poznan, Poland, June 1994.

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

[21] V. Krishnaswamy, D. Walther, S. Bhola, E. Bommaiah, G. Riley, B. Topol, and M. Ahamad. Efficient Implementations of Java Remote Method Invocation (RMI). In *Proc. of the 4th Conf. on Object-Oriented Techniques and Systems*, pages 19–36, 1998.

[22] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. on Computer Systems*, 7(4):321–359, Nov. 1989.

[23] B. Liskov, M. Castro, L. Shrira, and A. Adya. Providing Persistent Objects in Distributed Systems. In *Proc. of the 13th European Conf. on Object-Oriented Programming*, pages 230–257, Lisbon, Portugal, June 1999.

[24] H. Lu, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Message Passing Versus Distributed Shared Memory on Networks of Workstations. In *Proc. SuperComputing '95*, Dec. 1995.

[25] S. Lucco and O. Sharp. Delirium: An Embedding Coordination Language. In *Proc., Supercomputing '90*, pages 515–524, New York, New York, Nov. 1990. IEEE.

[26] J. E. B. Moss. Design of the Mneme Persistent Object Store. *ACM Trans. on Information Systems*, 8(2):103–139, 1990.

[27] A. Muthitacharoen, B. Chen, and D. Mazières. A Low-Bandwidth Network File System. In *Proc. of the 18th ACM Symp. on Operating Systems Principles*, Banff, Canada, Oct. 2001.

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

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

[30] D. J. Scales and K. Gharachorloo. Towards Transparent and Efficient Software Distributed Shared Memory. In *Proc. of the 16th ACM Symp. on Operating Systems Principles*, St. Malo, France, Oct. 1997.

[31] R. Srikant and R. Agrawal. Mining Sequential Patterns. IBM Research Report RJ9910, IBM Almaden Research Center, Oct. 1994. Expanded version of paper presented at the Intl. Conf. on Data Engineering, Taipei, Taiwan, Mar. 1995.

[32] 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.

[33] Sun Microsystems Inc. Java Remote Method Invocation Specification. Mountain View, CA, 2001. Avaialble at `http://java.sun.com/j2se/1.4/docs/guide/rmi/spec/rmiTOC.html`.

[34] 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.

[35] C. Weems, G. Weaver, and S. Dropsho. Linguistic Support for Heterogeneous Parallel Processing: A Survey and an Approach. In *3rd Heterogeneous Computing Workshop*, pages 81–88, Cancun, Mexico, Apr. 1994.

[36] 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.

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

[38] H. Yu and A. Vahdat. The Costs and Limits of Availability for Replicated Services. In *Proc. of the 18th ACM Symp. on Operating Systems Principles*, Banff, Canada, Oct. 2001.

[39] 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.