Indigo: User-level Support for Building Distributed Shared Abstractions[*]

Prince Kohli
Mustaque Ahamad
Karsten Schwan

GIT–ICS–94/53

Revised October 27, 1995

Abstract

Distributed systems that consist of workstations connected by high performance interconnects offer computational power comparable to moderate size parallel machines. Middleware like Distributed Shared Memory (DSM) or Distributed Shared Objects (DSO) attempts to improve the programmability of such hardware by presenting to application programmers interfaces similar to those offered by shared memory machines. This paper presents the portable *Indigo* communications library which provides a small set of primitives with which arbitrary shared abstractions are easily and efficiently implemented across distributed hardware platforms. Sample shared abstractions implemented with Indigo include DSM and a variety of DSM protocols as well as fragmented objects, where object state is split across different machines and where fragment communications may be customized to application-specific consistency needs. The Indigo library's design and implementation are evaluated on two different target platforms, a workstation cluster and an IBM SP-2 machine. As part of this evaluation, a novel DSM system and consistency protocol are implemented and evaluated with several high performance applications. Application performance attained with the DSM system is compared to the performance experienced when utilizing the underlying basic message passing facilities or when employing Indigo to construct customized fragmented objects implementing the application's shared state. Such experimentation results in insights concerning the efficient implementation of DSM systems (e.g., how to deal with false sharing). It also leads to the conclusion that Indigo provides a sufficiently rich set of abstractions for efficient implementation of the next generation of parallel programming models for high performance machines.

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332–0280

[*]This work was supported in part by the National Science Foundation grants CCR-9106627 and CDA-9501637 and ARPA contract DABT-63-95-C-0125.

1 Introduction

Recent hardware developments include the application of HPCC interconnects to networked computers and the construction of loosely coupled 'parallel machines' from sets of workstations linked over the network. In addition, parallel machines of moderate sizes are becoming ubiquitous, ranging from SMP nodes in distributed memory architectures to workstations using multiple processors for increased throughput. The resulting availability of such heterogeneous architectures has led to user demands that it should be possible to program both shared and distributed memory machines in a similar fashion.

Demands for increased programmability of distributed memory machines have given rise to research efforts resulting in Distributed Shared Memory (DSM) libraries layered on top of the networking and virtual memory systems [6,7,14,19,23]. They have also resulted in the development of distributed object-oriented systems offering user programs network-wide access to shared services [12,18,32] as well as object-oriented concurrent programming layers on top of shared and distributed memory machines [10,21,33,36].

Our contribution to the emerging field of heterogeneous parallel programming is the development of the *Indigo* communication library. Using the small set of calls offered by Indigo, developers can implement efficient, network-wide distributed shared abstractions (DSA) of any size or type, ranging from memory objects (DSM), to fragmented shared objects (FSO) in which object state and functionality may be fragmented across multiple machines' memory units [10]. This breadth of support distinguishes Indigo from communication libraries like active messages [37] that do not offer the specific functionality required by DSM or FSO implementations. Similarly, in comparison with the Nexus library developed for concurrent C++ [14] and with the lower RPC-like layers of distributed object systems like Spring [32], Chorus [27], and distributed objects [33], Indigo attempts to combine both the functionality required by distributed objects and by distributed shared memory. Indigo's goals are similar to those of the Tempest library [19] but it differs from Tempest in its additional support for objects and in its exclusion of constructs required only for DSM implementations (e.g., read-write access detection). Such DSM-specific support is implemented above Indigo's level of abstraction. Our hope is that libraries like Indigo may provide one basis for the development of standard, lower level interfaces, perhaps even supported by operating systems or hardware, based on which the next generation of parallel programming models may be implemented.

Using implementations of Indigo on a set of workstations and on the IBM supercomputer SP2, this paper also addresses several specific research issues of general interest to parallel and distributed system developers and end users:

1. *Consistency mismatch:* one critical cause of performance problems in parallel and distributed systems is the potential mismatch in the state consistency required by application programs versus the one created by communications in underlying DSM or object systems. This mismatch can cause additional communications, expensive buffering and copying of message data [13], and increased communication latencies compared to equivalent implementations directly layered on mes-

sage passing systems using only the 'minimal' set of required communications. For example, on a write to a shared data item, some DSM systems invalidate or update copies of the data at other nodes. This may be unnecessary because the write is done inside a critical section and processes at other nodes will not be able to access the modified data item until control of the critical section is released [7,20,23].

2. *Granularity mismatch:* in many DSM systems, consistency for shared data is provided at the level of a page. This implies that an update to a small amount of shared data may result in the transfer of an entire page. It also introduces the problem of false sharing; multiple unrelated data items may be stored in the same page and consistency actions for one item may interfere with accesses to another item because both items reside on the same page. Solutions to the false sharing problem like the *diff* mechanism [6] can introduce additional copying and processing overheads.

3. *Structure of shared state:* DSM systems treat all shared information as an array of shared bytes, but these bytes are used to store information that has structure or type (e.g., fragmented objects like shared work queues). It should be possible to exploit such structural information both to communicate object state changes only to those object fragments affected by them and to permit object programmers to implement object-specific consistency- and granularity-correct consistency protocols [33].

The Indigo library described in this paper is used to address each of the three problems listed above. This is possible because rather than providing a single notion of consistency, Indigo instead provides a small set of programming constructs with which application-specific consistency protocols may be implemented for DSM or FSO. For example, using Indigo, we implement an extended causal memory model. This model makes use of synchronization interactions with consistency maintenance as is done in Treadmarks [22]. It differs from Treadmarks in two ways: (1) coherence is not provided at page level but at the level of user defined shared data objects and (2) several methods of handling false sharing are explored, including one method that imposes zero overhead when the program does not actually exhibit false sharing. In addition, we explore how type information may be used to reduce communication overhead in sample distributed shared abstractions considered as FSOs compared to their performance with their message-based and DSM-based implementations.

In the remainder of this paper, we first describe the interface and implementation of the *Indigo* library. For portability across the various target architectures addressed by our current and future research, Indigo's implementation is layered on the PVM heterogeneous programming system, such that both PVM and non-PVM underlying communication layers may be employed by Indigo calls. Indigo's performance with PVM is assessed with several application programs. These programs also serve in the evaluation of user-level causal memory and distributed shared abstractions implemented on Indigo. Specifically, we quantify the gains made possible by (1) consistency operations performed for user defined objects rather than pages, (2) various techniques for handling false sharing, and (3) the exploitation of type or structure of shared state.

2 The Indigo Library

One issue now being addressed by computing researchers and vendors is the support for both threaded and message passing models of parallel programming within a system consisting of shared memory and distributed memory multicomputers. The approach taken by Indigo is that such support should be layered on a small set of underlying hardware and operating system independent primitives, with which arbitrary shared abstractions may be built. Toward this end, Indigo views the underlying hardware as a cluster of logical processors (processes), which may or may not be physically distributed, each with its own local memory. Such memory may be private and therefore, not directly accessed by Indigo or shared, in which case it is treated as a cache of shared abstractions residing in it. The implications of this approach are that: (1) shared abstractions must be defined in terms of program-level rather than hardware-defined units like pages and (2) such abstractions must be able to range from memory units providing fixed operations and different degrees of consistency to typed objects with user-defined operations. In addition, (3) it should be straightforward to associate execution threads with any operation on those abstractions.

In the remainder of this section, we explain Indigo's basic functionality using several shared abstractions. The first abstraction is a single shared variable manipulated with 'read' and 'write' operations performed by threads residing on different machines. The second abstraction is a slight variation of the first in which a single shared variable is accessed in a consumer-producer fashion by reader and writer threads. The third abstraction is a shared queue that must be manipulated with type-specific operations. With these examples, it is straightforward to explain how Indigo's basic functions provide for implementation of higher level DSM and FSO functionalities.

2.1 Indigo Primitives

2.1.1 Naming

With Indigo, two processes in an application share a variable if they call it by the same name. Therefore, within each application, all shared abstractions must be named uniquely across all of their cached copies. Toward this end, Indigo maintains tables that list for each item its name, local address (i.e., address of the cached copy), and meta-information (e.g., size). The item's name is its unique identifier.

Naming as well as space allocation for shared abstractions are achieved with Indigo's *share* call. All processors sharing a certain variable use this call to "register" it with Indigo. Such registration returns an address pointer and also provides Indigo with information concerning the variable's size. In response, Indigo allocates appropriate amounts of cache space at each participating site, and it also keeps track of the abstraction's local addresses (which may differ across sites). The use of *share* calls is illustrated in the next subsection.

2.1.2 Data Movement Calls in Indigo

Indigo's main tasks are derived from its charge to maintain caches in which shared abstractions' state is stored. As a result, its basic functionality is comprised of the naming support identified in the previous section and of calls with which cache contents are manipulated via accesses to shared abstractions. Specifically, the *put* and *get* calls provided by Indigo permit access to abstractions stored in remote caches. (Local caches are accessed via read and write operations.) The *purge* call invalidates remote cache entries, and *events* with *event handlers* permit the association of user-defined operations with basic Indigo actions. Moreover, variants of *put* and *get* may be used to synchronize accesses to shared abstractions.

Using *put* to access a shared variable. Consider the manipulation of a single variable, say i, in a shared memory program running on some SMP machine. Concurrent access to this variable by two processes P_1 and P_2 requires the use of a critical section, typically implemented with lock and unlock calls, as shown by the pseudo-code on the left hand side of Figure 1. The corresponding Indigo code performing the same task is shown on the right hand side of the figure. In this code, a shared variable *sh_i* is declared by each process, followed by a *share* call to Indigo indicating these processes' joint use of *sh_i* (and providing to Indigo the name and size of the shared object). Henceforth, *sh_i* is used in place of i, where each process continues to use locks to ensure correct access to *sh_i*. However, whenever a new value is computed for *sh_i*, each process also performs a *sync_put* operation in which it passes to Indigo a pointer to the shared variable and the sender and recipient ids[1]. The effect of *sync_put* is to place the updated value into the appropriate remote cache. This call is performed inside the critical section, thereby guaranteeing that remote copies are updated before exiting the critical section. In this example, we use a refinement of *put*, called *sync_put* (for *synchronous put*), which guarantees that the update has been performed on the remote cache before the call returns.

Using *get* to access a shared variable. Consider another example where the two processes have a producer-consumer relationship. That is, instead of two processes updating a shared variable simultaneously, one of them updates it and the other reads it after being signaled when the update is complete. In a distributed system, either of two possible strategies may be implemented to achieve this. Either the producer pushes the data continuously to the consumer, or the consumer continuously pulls the data from the producer. The former can be accomplished using the *put* call discussed above. For the latter, an equivalent *get* call is used. It takes the same parameters as *put* and results in fetching the named data item from the remote cache to the local one. The Indigo code for the consumer is given in Figure 2. The producer makes no special Indigo calls except to declare its shared data.

Using *purge* to manipulate a shared variable. Indigo's *purge* call is used to invalidate the named data item at the remote location. Validation of an item implies setting a valid bit on that item, which means that the local cache has a good copy of the object data, and invalidation turns this bit off.

The full list of Indigo calls falls into the three broad categories mentioned above - *get*, *put* and *purge*. As stated previously, each call takes three parameters in this form:

[1]These ids are PVM process identifiers which are unique systemwide.

Shared Memory Code:

```
Process 1:
int i;
lock(x);
i = i + 1;
unlock(x);

Process 2:
int i;
lock(x);
i = i + 2;
unlock(x);
```

Code with Indigo calls:

```
Process 1:                        Process 2:
int *i;                           int *i;
struct item sh_i;                 struct item sh_i;
strcpy(sh_i.name, "i");           strcpy(sh_i.name, "i");
sh_i.size = sizeof(int);          sh_i.size = sizeof(int);
share(&sh_i);                     share(&sh_i);
i = (int *) sh_i.addr;            i = (int *) sh_i.addr;
lock(x);                          lock(x);
*i = *i + 1;                      *i = *i + 2;
sync_put(&sh_i,id_p2,id_p1);sync_put(&sh_i,id_p1,id_p2);
unlock(x);                        unlock(x);
```

Figure 1: Indigo Code Sample

```
Consumer:
Wait for data to be produced;
sync_get(&sh_i, producer, consumer);
f(i) = sin(* (float *) i);
```

Figure 2: Using *sync_get*

*call(char *shared_obj, int destination, int source).*

1. *put:* Copies a data item value to a remote location. It is refined into the following categories:

   - async_put: initiates a send of a data item to some destination and returns. This may be used in chaotic programs which do not require synchronization at every step and thus a process need not make sure that the data has reached the remote node before carrying on with its execution.
   - sync_put: sends a data item to a destination cache and returns only when it has been placed in that cache. This is most useful when some form of synchronization needs to be performed, like in RC (Release Consistent) memory [16], where one needs to place some data at another node before transferring a lock.

     (There also exists a sync_put_validate call that is similar to sync_put except that it also turns on the valid bit remotely. An async_put_validate has not been found necessary in the applications we have seen till now.)

2. *get:* Copies the data item value from a remote location to the caller's memory space. This also is of various types:

   - async_get: requests an item from a process and continues without blocking. It also sets the invalid bit for the data item at the remote location. This is similar to prefetch and in that capacity, it may be used to overlap communication with computation.
   - async_get_copy: similar to async_get, except that the remote item's invalid bit is not set.
   - sync_get: a blocking operation which also sets the invalid bit for the data item at the remote location. This, the most commonly used get command, may be utilized for data pulling operations.
   - sync_get_copy: similar to sync_get, except that the remote item's invalid bit is not set.
   - sync_get_if_valid: similar to sync_get, except that it will get a value only if it was valid at the remote location. This is useful for implementing servers, where one wishes to pull a value only if it is valid at the remote location.
   - async_get_if_valid: the non-blocking version of sync_get_if_valid.

3. *purge:* invalidates a data item remotely (sets invalid bit). This can be made to act like remote invalidation.

2.1.3 Events and Event Handlers

Our third example motivates the second set of calls provided by Indigo- those that allow the active handling of data movement messages generated by the functions discussed above. Consider a shared 'work queue' abstraction fragmented across the processors that share it. Each processor dequeues and performs jobs from its local queue fragment until there are no more jobs available locally. Since the queue is logically shared, to ensure that a processor is able to remove work from remote fragments, our queue implementation creates a ring-communication structure between the processors. A processor completes all of the jobs in its own queue fragment and, subsequently, makes a request to a neighbour for more jobs. If such jobs exist, the latter passes them on, else the request is forwarded to the next node in the ring.

When this abstraction is implemented, we realize that, besides providing simple data movement calls, some notion of *active messages* [37] is needed. In other words, we need the ability to asynchronously invoke handlers for *events* such as a *put* or a *get* when one process performs these operations on another process's shared memory cache. For this purpose, Indigo defines a list of *events* that correspond to each data movement call defined above. At the same time, it also provides the ability to associate an *event handler* with each such event, which is a subroutine that is invoked at the time the specified event occurs. Now, using its data movement and event handling facilities, the Indigo library can be used to implement an abstraction like the 'work queue' illustrated in Figure 3.

```
/* Application process using the work queue. Shared variables are "req" and "jobs" which are used to send
   and fulfill requests. "lower" and "upper" outline the two limits of the local unprocessed fragment of the
   queue, and are shared between the event handler and the process. Local synchronization has not been shown
*/

while(1)
{jobid = lower;
 if (jobid < 0) {echo done; break;}                  /* No more jobs. Done. */
 if (jobid < upper)                                  /* Process the next job */
     {lower++; break;}
 if (jobid == upper)                                 /* We are starting on the last job, so request */
     {lower++;                                       /* more jobs from the next node for next time */
      async_put(&req, next_in_ring(myid), myid);     /* Send control data "req" (req.name = "req_name")
      break;                                            to request jobs */
     }
 if (jobid > upper)                                  /* The reqested queue fragment has not yet arrived,
     sleep(2);                                          so wait for it */
}
```

```
/* Event handler for this application. Handles remote requests for queue fragments as well as the replies to such
   requests made locally */

void put_handler(arg)
{if (arg→name == "req_name")                          /* A remote request for a local queue fragment */
    if (arg→requester == arg→myid)                    /* My request has returned empty-handed
      lower = -1;                                        to me, so no more jobs to do. */

    else
     if (upper - lower > 2)                           /* If more than two unprocessed elements locally,
                                                         fulfill request, else forward to the next node */

      {Place half of local unprocessed jobs
       in shared object 'jobs';                       /* jobs.name = "job_name" */
       async_put(&jobs, arg→requester, arg→myid);     /* ... and send it back */
      }
     else async_put(&req, next_in_ring(arg→myid), arg→requester); /* Forward...*/

 else if (arg→name == "job_name")                     /* Request made locally has been fulfilled */
    {Unpack data and update local queue pointers ('lower' and 'upper'); }
}
```

Figure 3: Shared Work Queue Abstraction using Indigo calls

The first part of the figure shows the application code for a process that performs jobs dequeued from the shared queue. In each iteration of the `while` loop, it checks to see if the next job available is the last one (if `jobid < 0`) in the local fragment of the work queue. In that case, it sends a request to the next node (via `async_put()`) and completes the last job while its request is getting processed. Indigo calls are used to exchange such requests. Specifically, *put*s are performed on control data that defines commands rather than on the queue data itself.

The 'shared queue' event handler is shown in the second part of the figure. Invoked as a result of a *put*, it checks its argument each time to figure out which action to perform. If a queue part is requested (a dequeue remotely invoked on the local queue fragment), this request is fulfilled by sending half of the locally available jobs. If such jobs are not available, the request is forwarded. All these actions are performed via *async_put*s. If the incoming *put* contains jobs requested previously, the event handler places them in the appropriate location.

Besides providing message handling independent of the local process, event handlers also facilitate the overlap of computation and communication as there is always a concurrent handler thread running at each node which can handle incoming messages. Handlers can also be used to implement arbitrary shared abstractions. Alternatively, for complex abstraction-specific computations, handlers may simply trigger application-level threads which can perform abstraction-specific operations of arbitrary complexity. Such association of a handler with incoming messages provides a form of active messages, the use of which has been shown to improve program performance and modularity [33,37].

Event types. The following different types of events are recognized by Indigo (and can thus be "captured"). Each such event has an associated per-process handler that is executed when the event occurs in that process's domain. For example, if some remote process performs a *put* on a local process's address space (i.e., on one of its shared variables), then the local handler associated with *put_recvd* is invoked.

1. *Put-related events:*
   - put_recvd: the associated handler, if any, is called when (after) a put is performed on a data item in the process's shared address space.
   - put_done: the associated handler is executed when a put has been acknowledged by the remote destination.
2. *Get-related events:*
   - get_recvd: similar to put_recvd (the associated handler is called when some remote process tries to get a data item from the process's shared address space, before the data is copied into the outgoing buffer).
   - get_done: similar to put_done (if present, the handler is executed when a get call returns).
   - get_cp_recvd: associated handler executed when a get_copy is performed on the specified data item.

3. *Purge-related events:*
   - purge_recvd: associated handler executed when the data item is purged.
   - purge_done: associated handler executed when the purge has been performed remotely.

Mapping particular events to appropriate handlers allows us to associate execution threads with (local or remote) operations on distributed abstractions. Remote object invocations, for example, are asynchronously handled at the remote processor by simply providing the event handler code that must be run when the said invocation arrives. This functionality is useful in developing various FSOs as well as DSMs, as will be shown later.

We should note here that many DSMs need some way of detecting accesses to shared data (e.g., write to shared data). Several techniques exist for access detection [11,38]. Due to that reason and also due to the fact that many DSAs do not need such mechanisms, access detection techniques are not part of Indigo. Our DSM implementation programs its own access detection, as discussed in a later section.

3 Building DSM and FSOs on Indigo

Indigo's operations are sufficient for programming a wide range of Distributed Shared Abstractions (DSAs), including DSMs. In this section, we demonstrate this fact by implementing a DSM on top of the Indigo library and also by programming a number of non-DSM abstractions. The efficiency of the library is the topic of the next section.

One of the contributions of this paper is an innovative implementation of Extended Causal Memory (ECM) [20] using the Indigo library. This implementation takes advantage of Indigo's ability to provide arbitrary consistency for user-defined objects. We also explore how the calls provided by Indigo can be explicitly used to implement application-specific DSAs. This will help us demonstrate how structure and type information can be easily utilized to obtain performance that is close to or even better than message passing implementations of the applications. The specific DSAs implemented in our work are Fragmented Shared Objects (FSOs), similar to the 'topology' objects described in [33]. Topologies are event-trigger-able fragmented shared objects (on a network in our case) where the events are generally associated with the arrival of remote *invocations* on local fragments. This in turn implies that all such invocations (or messages sent by the user and/or on the behalf of the user) are *active* in the sense that they are capable of triggering additional messages and/or computation upon their arrival, in addition to updating the state of the local process.

3.1 Extended Causal Memory

As a way of avoiding the unnecessary communication costs that are incurred in strongly consistent DSM systems, a number of memory systems with weaker consistency have been explored and implemented [1,2,3,4,6,7,15,16,17,20,22,24,29,30]. We claim that any one or a combination of these can easily be constructed with Indigo. We illustrate this

$$\begin{array}{lll} P_1: & w(x)1 & r(y)0 \\ P_2: & w(y)1 & r(x)0 \end{array}$$

Figure 4: A causal execution history

by exploring the implementation of one memory system – Extended Causal Memory (ECM) [20]. ECM provides benefits similar to other systems such as Treadmarks. It can be defined for all types of programs including ones with data races, and it exploits both weak-consistency and weak-ordering approaches to improve performance.

ECM definition. In a strong memory model like Sequential Consistency (SC) [26], the memory system performs all operations of all processors so that they appear to be executed in some sequence respecting program order. This ordered set of operations constitutes the *view* of each processor (i.e., how it perceives the operation of the memory system). Different memory systems offer different sets of views to the processors. In memories weaker than SC, a processor view may not include all accesses of all processors. For example, in ECM, the view of each processor includes all of its own accesses (reads and writes) but only the writes of other processors all of which are constrained to be placed in a causal order [25]. ECM thus places weaker constraints on the consistency of shared variables. Consider the example in Figure 4. Processes P_1 and P_2 first each write to location x or y, respectively, and then each read the location written by the other processor. This execution is not possible with SC because there exists no legal sequential execution (thus, an SC view) that includes all four operations and maintains the program order dependencies between the operations. However, this execution is possible with ECM, because in P_1's view, $w(y)1$ is placed after its read of y and a similar view exists for P_2.

ECM also considers orderings between memory operations that are introduced by synchronization operations like lock, unlock and barrier and incorporates them into the causal order. For example, a memory operation that precedes an unlock or a barrier in program order is ordered before memory operations that follow the matching lock or barrier call.

3.2 Implementation of ECM on Indigo

ECM system components. As shown in Figure 5, the implementation of ECM on Indigo utilizes (1) a per-machine daemon process created for all Indigo applications, (2) synchronization servers, and (3) data servers. The daemon process captures all Indigo messages addressed to that node and posts them as required. The synchronization servers handle all *lock*, *unlock* as well as *barrier* requests. Data servers act as a storehouse for shared data. Namely, when a process faults on some item that is not available locally, it communicates with a data server to retrieve it.

All communication between the daemon and the local application processes is through Unix shared memory (allocated using *shmget* calls). In addition, all ECM memory is allocated in the memory shared between the daemon and the application process, so that user processes can simply read or write ECM's shared objects. Such read and write accesses may trigger *get* and *put* operations which in turn result in messages being sent to the data servers. Finally, the daemon at the data server node can respond to many such messages using ECM-specific handlers without involving the actual data server process. For example, ECM handlers perform version maintainence and also assist in operations required for false sharing (see Figure 5).

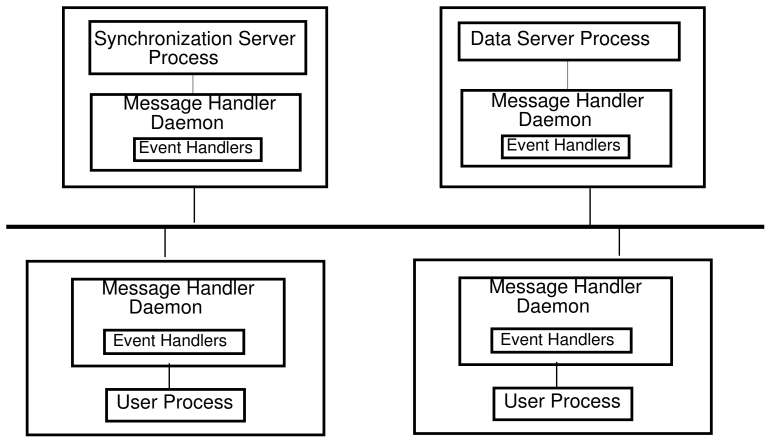


Figure 5: System Layout for ECM

A data server may handle any number of ECM objects. By allocating unique pages for each shared item (when *share* is called), ECM can track reads and writes to an object via page traps using the Unix *mprotect* call.[2] Writing to an object for the first time in a critical section sets a dirty bit for that object. Any access to an unavailable object (i.e., the page containing the object has protection set to NULL) results in communication with the data server and a fetch of the current value of the object via the *sync_get* call provided by Indigo. Notice that page traps are only used to detect writes and the unavailability of the object in the local cache. Data transfers are done only for the object data and not for pages.

ECM consistency algorithm. In our implementation of ECM, all consistency-related actions for cached objects take place at synchronization points. Such consistency of shared objects relies on vector timestamps (VTS) [31] stored with each object. These timestamps are read from vector clocks that have one entry for each process in the application. A process increments its entry in its vector clock (VT_p is p's clock) when a write dirties a cached object. The sequence of steps taken at such points that affect the value of the clock are enumerated in Figure 6 and are explained below.

The algorithm shown in the figure assumes that data-race free programs access shared objects stored in ECM. It has been shown that in such a case, consistency actions need to be performed when synchronization operations are executed [20,22]. In particular, vector timestamps received with synchronization variables are used to detect objects in the processor cache which may potentially be overwritten. Such

[2]We chose the page-based approach because it is simpler to implement as compared to the compiler-based approaches used elsewhere. Moreover, the latter tag each write to shared variables and thus perform extra processing on all such writes rather than on just the first write to an object in a critical section.

At Lock Time, process p, lock x:

```
lock_req(sync_server, $x$);
$\forall$ shared variables $d$,                /* On getting the lock */
    if ($VTS(d) < VTS(x)$)                     /* Compare timestamps: smaller */
    OR ($VTS(d) \bowtie VTS(x)$)               /* or concurrent */
        then mark_invalid($d$);
            if dirty($d$) then sync_put(&$d$, data_server, $p$);
$VT_p := \max(VT_p, VTS(x))$;                  /* Update local timestamp */
return;
```

At unlock time, process p, lock x:

```
$\forall$ shared variables $d$, if dirty($d$),
    sync_put(&$d$, data_server, $p$);
    mark_clean($d$);                           /* Unset dirty bit */
$VTS(x) := VT_p$;                              /* Update timestamp of lock to local timestamp */
release_lock(sync_server, $x$);                /* Release lock */
return;
```

Write fault handler for object d, process p:

```
                                               /* Entered only when a write is attempted on an
                                                  object which was marked invalid or clean */
if invalid($d$) sync_get(&$d$, data_server, $p$);
mark_dirty($d$);
Increment $p$th component of $VT_p$;
$VTS(d) := VT_p$;
return;
```

Read fault handler for object d, process p:

```
                                               /* Entered only when a read is attempted on an
                                                  object which was marked invalid */
sync_get(&$d$, data_server, $p$);
return;
```

Figure 6: Implementation of ECM

objects, which have lower timestamps, are locally invalidated. Our algorithm performs invalidations differently because of the way false sharing is handled by us. We allow multiple writers to write an object concurrently. As as result, an object may have to be invalidated even when its timestamp is concurrent with the timestamp received with a lock. We explain why this is necessary in the next section. If a dirty object is invalidated, it is sent to the data server using a *sync_put* call so the server can merge the modifications to the object with the copy that it maintains. This is done even for objects for which the process holds the lock as false sharing might be present (Section 3.3). Thus, when a lock variable is acquired, all objects in the cache that have an older or concurrent timestamp relative to the timestamp of the lock are locally invalidated after dirty copies are sent to the data servers.

At unlock time, as the figure shows, all dirtied data items are sent to the server, again via a *sync_put* call, before the unlock message is sent. This ensures that when another process acquires the lock and requests the data, it will receive the current copy of the object. The sent data is again marked clean. The unlock message simply sends a timestamp read from the local clock to the synchronization server as part of the lock. In our implementation, at a barrier, the same actions are performed as would be in the case of an unlock followed by lock. The only difference is that the synchronization server waits for all processes that were to synchronize before sending the release message with a timestamp that is the component-wise maximum of the incoming timestamps.

Algorithm optimizations. We have optimized the algorithm in several ways. First, as shown in Figure 6, the *sync_put*s of the dirty data take place *inside* the critical section (CS). This results in an increase in the length of the CS, which can potentially increase the execution time of applications. We have thus moved the sending of the dirty data *outside* the CS as shown in Figure 7. This results in the same number of *put*s as before but is likely to reduce lock contention. Furthermore, dirty data is pushed to the server at lock time only when there is nesting of lock calls, that is, when a lock command is issued before the previous lock has been released. If there is no nesting, then all dirty data would already have been pushed to the server at unlock time and marked clean and no extra work would be performed on a lock.

The second optimization of ECM's consistency algorithm concerns reducing the amount of communication. Let us assume that the algorithm stated previously (see Figure 6) is used to execute the code shown in Figure 8. There are two data items d_1 and d_2, with corresponding locks x and y. Let us assume no false sharing (the next section considers false sharing). After the unlocks of x and y by processes P_1 and P_2 respectively, the timestamp of x and d_1 at P_1 is $\{10\}$ and that of y is $\{01\}$. Thus, when P_1 acquires y, d_1 is marked as *invalid* (see the algorithm in Figure 6). So is d_2 (with timestamp $\{00\}$) of course, which results in the process faulting and thus going to the server and getting the new value of d_2 before incrementing it. Lock y is then released and x is acquired. Assuming no one else has acquired x in the meantime, it will still have the timestamp $\{10\}$. On comparing it with the timestamp of d_1, since the two are equal, a *sync_get* of d_1 is performed even though P_1 still has the correct value in its cache. We need to detect that d_1 is current by using timestamps. When the test to mark shared objects *valid* or *invalid* is performed, if the two timestamps being compared are equal (see Figure 7), then the object is remarked as valid. (The

```
*At Lock Time, process p, lock x:*

∀ shared variables $d$, if dirty($d$),
    sync_put(&$d$, server, $p$);
    mark_clean($d$);                      /* Unset dirty bit */
lock_req(server, $x$);
∀ shared variables $d$,                   /* On getting the lock */
    if ($VTS(d) \geq VTS(x)$)             /* Compare timestamps */
        then mark_valid($d$);
        else mark_invalid($d$);
$VT_p = \max(VT_p, VTS(x))$;              /* Update local timestamp */
return;
```

Figure 7: Optimized Code at Lock time

P_1: Lock(x); $d_1 = 1$; Unlock(x); Lock(y); $d_2 = d_2 + 1$; Unlock(y);
Lock(x); $d_1 = d_1 + 1$; Unlock(x);
P_2: Lock(y); $d_2 = d_2 + 1$; Unlock(y);

Figure 8: Timestamps and False Sharing

case where the data timestamp is greater than the lock timestamp is considered in the next section.) This *revalidation* process avoids unnecessary message transfers even in the absence of any information about the presence of false sharing in the program. For instance, in the example being discussed, when P_1 re-acquires lock x with timestamp $\{10\}$ and compares it with the timestamp of d_1, since the two are equal, d_1 is re-marked as *valid* which makes sure that the process reads the local (and correct) value rather than ask the server for the new value unnecessarily.

The invalidation done on acquiring a lock is similar to the consistency actions of Treadmarks and the *implicit invalidate* protocol of Distributed Filaments [15]. However, compared to the latter, in the case when the data is indeed good, we are able to revalidate it using the timestamps.

We have thus demonstrated how Indigo facilitates DSM implementations with the variety and power of its calls. Its data movement calls which fetch data from or place it in remote caches, as well as the event handler functionality via which consistency-related actions can be executed both locally and remotely, help greatly in implementing a DSM system. The above implementation can now be used to explore various issues that are of concern to DSM builders and programmers. One such issue is that of false sharing, which is the topic of our next section.

3.3 False Sharing and its Handling in ECM

We demonstrated above how we were able to implement ECM using the infrastructure provided by Indigo. Although false sharing (FS) is expected to be rare because consistency is provided at the level of user-defined objects, it may not always be possible to eliminate it completely [8]. In this section, we use the Indigo-based ECM implementation to investigate some of the issues involved in FS and also explore several ways of handling it. Moreover, an innovative approach for FS handling is implemented that incurs the overheads associated with FS only when actual FS exists.

A new approach to handle FS. Various software-based DSMs ([6,7]) have used client-based *diff* approaches, where the user process accessing shared objects computes the *diff*. However, since they make copies for all objects that could potentially be written, they actually introduce copying overhead even when there is no actual false sharing. In our approach, which we call *server-diff*, if there is actual false sharing, *diff* operations are performed by the server node. The server keeps track of all the objects that have been sent out and to whom using copy sets and also maintains their versions. At each *put*, if the sender was the only process with a copy of the object, the server's event handler overwrites its copy. If two or more processes have copies of the object, then the first *put* creates a new version of the object and places the incoming data there. Each subsequent *put* results in a *diff* being performed between the incoming object and the version that was sent to the process earlier. That is, if O was the object whose copy had been requested by more than one process, and the first process returns with the updated version O_1, then O_1 is stored along with O. If a second process subsequently returns with another updated version, O_2, a *diff* is performed between O and O_2 and the changes are applied to O_1. A *put* also removes the sender from the copy set of the object. If the *put*ting process is not in the copy set of any version of the object, the changes are applied to all versions that exist at that time. Requests for the object are satisfied with the most recent version of the object.

For a program free of data races and exhibiting no FS, in the server-based *diff* approach, the data server will not create additional copies of an object as more than one processor will never write the object concurrently. The copy creation and *diff* operations will be performed only when the server detects that two or more processors have changed the object at the same time (only dirty copies of the object are sent back to the server) thus implying false sharing. Programs with no false sharing therefore do not incur any overhead due to false sharing which is not the case with client *diff*s since client nodes do need to make copies and compute *diff*s in the latter case.

Let us now consider the previous example (Figure 8) when there is false sharing present, that is, when d_1 and d_2 are components of the same object. As a result of FS, though the program is data-race free, two processes might be writing concurrently to the same object, i.e., write-write false sharing exists. Specifically, when P_1 acquires y (which has already been released by P_2), it still invalidates d_1 which means that it also invalidates d_2, since both d_1 and d_2 are part of the same object. Thus, when P_1 tries to read the value of d_2 inside its critical section, it faults and fetches the new value from the data server which has merged the updates of both P_1 and P_2 to create a new copy. The timestamp of the object that contains d_2 is set to $\{21\}$ after P_1 writes it.

P_1: Lock(x); $d_1 = 1$; Lock(y); $d_2 = d_1 + d_2 + 1$; Unlock(y); Unlock(x);
P_2: Lock(y); $d_2 = d_2 + 1$; Unlock(y);

Figure 9: Nested Lock Calls

y is unlocked and x is acquired again. Since d_1 is in the object that has timestamp $\{21\}$, when we compare it with x's timestamp ($\{10\}$), the object in which d_1 resides is re-marked as *valid*, which is fine as the correct value of d_1 still resides in the cache.

Invalidating objects and pushing dirty data. We can now also understand why objects with timestamps concurrent with the lock timestamp still need to be invalidated and dirty data needs to be pushed back to the server at lock time. Consider the execution in Figure 9. Let us assume as before that d_1 and d_2 are part of the same object. P_1 obtains the lock x and increments d_1. Concurrently, P_2 obtains and releases y after incrementing d_2. P_1 then tries to acquire lock y. Although y's timestamp is concurrent to the timestamp of the object on which d_1 and d_2 reside, it is necessary to invalidate the object. This will ensure that when P_1 accesses d_2 later, it will fault and get it again from the server. If we did not invalidate objects with concurrent timestamps, then on receiving y, the new value of d_2 will not be received by P_1 which will lead to an incorrect execution. Moreover, if (the dirty) d_1 is not pushed back at the time of locking y, then when the new value of d_2 arrives, it will overwrite the modification in d_1 as they are both part of the same object. This shows why false sharing requires pushing the dirty data back as well as marking objects with concurrent timestamps invalid at lock time.

The client-based-*diff* approach would work better for programs which may have false sharing but the amount of the object that is actually "dirtied" in an object in some critical section is small compared to the object size. In this case, only the changed data is sent over the network, which may result in considerable savings. The data server performs only a small amount of work, involving the incorporation of the *diff*s into the previously present object copy, as compared to the server-based approach case where the data server also performs the *diff* computation. However, in applications where a large fraction of shared data is dirtied, client-*diff* could perform worse as the message sizes will actually increase in this implementation thus leading to larger communication times.

We have argued that in the no-false-sharing case, which should be the common one, processes do not incur any overhead for handling false-sharing. Moreover, false sharing is handled by the system itself and the programmer does not need to provide any hints to the system.

Indigo's facilities were used extensively in the implementation of server-diff. Specifically, it provided event handlers, which consisted of the above functions at the server node and captured all the *put*s and *get*s that were performed there by application processes. A comparative implementation of client-diff was quite straightforward too, essentially involving *put* and *get* event handlers at the client nodes as well as *diff* packing at synchronization time. This thus demonstrates that Indigo can be used to explore various issues that arise in implementing DSM systems.

3.4 Distributed Shared Abstractions

Several concepts in Indigo exist for the purpose of supporting arbitrary distributed shared objects. Specifically, event handlers are useful not only for computing *diffs* in DSM objects but also for a more general implementation of active object methods. This section demonstrates how fragmented shared objects (FSOs) can be implemented using Indigo's active message functionality. It was mentioned earlier that shared memory is used to store typed data items that are shared between processes. ECM does not exploit this type information and only cares about read/write accesses by user processes. In contrast, with Fragmented Shared Objects (FSOs), the state of shared objects is distributed across different processors, and the consistency of such fragments is maintained using code customized for each shared object. In general, the code implementing fragment communication is supplied by the application programmer and can implement a wide variety of functions, including the exchange of data or control information relevant to the executing processes and/or the communications required for task synchronization, message forwarding/filtering under program control, etc. Such communications may also utilize object-specific logical communication structures defined between participating processes. For example, a shared queue might be represented as an FSO where fragments on which user processes call object methods like enqueue and dequeue are organized in a ring structure.

Since DSM is one example of an FSO, both kinds of shared abstractions use the same underlying Indigo functionality. For instance, shared data is declared in the same fashion in both and asynchronous remote invocations on object fragments are accomplished using Indigo event handlers, as explained below (also see Figure 10).

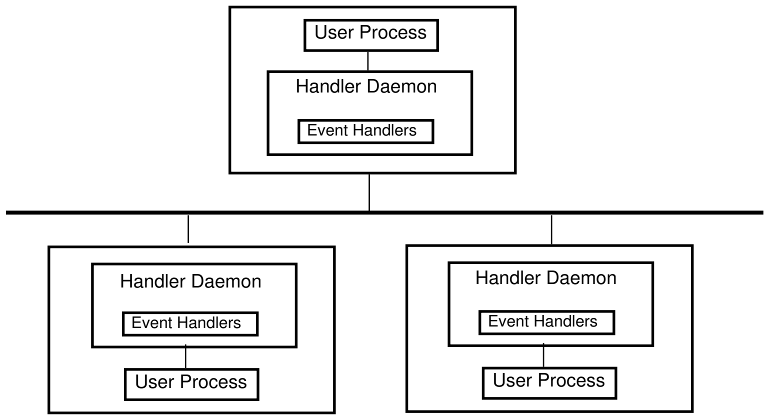


Figure 10: System Layout for FSO

In Indigo, the required communications and the associated type-specific operations are implemented using FSO-specific *event-handlers*. These are functions that are associated with events described in Section 2 and are called when the associated event takes place locally. For example, in the work queue shown in Figure 3, an application process can send some of its jobs to another process by executing a put call. At the other process, the *put_recvd* handler is executed when the *put* is done in its cache. These handlers are executed by the daemon process and are thus independent of the user process (the queue fragment is sent independently to the requesting process while the local process is performing its own computation). Since the programmer knows what kind of invocations are being sent, he can then incorporate that information in the event handlers, as well as use Indigo calls to send invocations to other processes (for example, to either reply to or forward the request in the queue implementation). Thus, event handlers are executed by a thread separate from the user application thread. These handlers can access all items in the process's shared item cache, make Indigo calls on its behalf and intercept all incoming accesses to the process's cache. This concurrency allows us to overlap computation with communication for many applications.

We presented a sample implementation of the work queue FSO in Section 2.1.3 (see Figure 3). Since an FSO implementation is customized for the particular application being programmed, we discuss it in the next section for each of our applications.

4 Performance of Indigo

We have demonstrated that Indigo can support different paradigms of distributed state sharing. The purpose of this section is to show that the flexibility provided by Indigo does not have an adverse impact on the performance of shared abstractions built with it. To demonstrate this, Indigo has been implemented on two different platforms. The first one is a cluster of Sparc2 workstations and the second one is an 8-processor IBM SP2. We measured the raw cost of Indigo primitives on each of these platforms as well as the performance of four different applications on two systems built on Indigo - ECM and FSOs, and their message passing (MP) implementations on PVM. This permits quantification of the overheads induced by Indigo as well as evaluation of the costs of DSM and FSO implementations, and the evaluation of their dependence on the nature of the underlying platforms.

We picked four applications that have different characteristics and exercise different parts of the Indigo framework, and are thus helpful in exploring the problems that this paper set out to address. Each of them, when implemented on DSM, as an FSO and MP, will provide reference numbers that quantify the cost of mismatch in consistency and granularity and the effect of type and structure information utilization. Moreover, the size of messages range from a few bytes to kilobytes, and synchronization methods vary from lock-based to barrier-based. We also introduce and observe the effects of false-sharing in one of these applications.

These applications are straightforward to program with ECM because data-race free applications can be programmed on ECM assuming the memory is sequentially-consistent. For FSOs, shared state fragments and their placement, as well as the logical communication structure between object fragments, have to be defined in an application-specific manner.

4.1 Applications

The four applications used to evaluate Indigo are listed below along with their data access patterns. Each of them interacts differently with the system thus providing us some useful insights. We also describe how the applications are programmed in the three systems: ECM, FSO and MP.

1. *The Traveling Salesperson Problem (TSP)*:
   Lock-based, small message size, data-dependent execution. The TSP algorithm is similar to the one used by Bal et al. [5] and uses a branch-and-bound method. It creates a statically defined job queue which is shared between the worker processes. The initial queue is built by unraveling all possible paths for the tour to a fixed depth. On ECM, locks are used to dequeue jobs from the queue as well for updates to the value of the best tour. Thus, there are two shared structures - the job queue and the best tour. Read accesses to the best tour are allowed without acquiring a lock and hence can have data races; however, updates to it are protected by a lock.

   Shared work queue FSO. We use the work queue abstraction defined earlier to implement this TSP algorithm as an FSO. As we explained before (see Figure 3), the handling and forwarding of requests is done by event handlers. The programmer informs Indigo about which functions to call when the various events occur. The event handlers also handle the propagation of better tour values to other nodes. This is done by updating the local copy of this variable and then passing it to the next node in the ring (by performing a *put* of that value, which then invokes the latter's *put_recvd* handler). Again, a user process never needs to execute a "receive" to update its local best tour value because the handlers take care of the local update and onward propagation. In MP, a master process stores the queue and best tour value and worker processes send messages to it to request the next job and the best tour value. The execution in each case is data dependent.

   In TSP, the message sizes are small, but quite a few messages are sent. On ECM, there is a (read-write) data race for the best tour value. The problem was solved for 17 cities.

2. *Linear Solver (LIN)*:
   Barrier-based, large messages. In this application, an iterative algorithm solves a system of equations $Ax = b$, where x is the set of unknowns. The processes split the x vector equally among themselves and each computes the part (x_p) assigned to it. Calculating x_i requires the values of $x_j, j = 1, 2, ..., n$, from the immediately preceding iteration. On ECM, all synchronization is barrier-based. There is one barrier per iteration. For each iteration, all values that were updated in the last stage are required. FSO and MP use a similar algorithm and thus the communication there is akin to broadcast. For FSO, the local event handlers take care of signaling the user process on the arrival of all required values from the previous iteration. There is no explicit synchronization in FSO since the arrival of data values allows the processor to start the next iteration. Message sizes are large, and all communication occurs at the end of an iteration. The algorithm

was run for an x-vector size of 2048 floats on the workstation cluster and 4096 floats on the SP2.

3. *Successive Over-Relaxation (SOR)*:
   Barrier-based, large messages, write-write false sharing. SOR is another iteration-based application that is frequently a kernel in many mathematical packages. The program is based on the parallel red/black SOR algorithm as described by Chase et al. [9]. At each iteration, a process writes some value into a set of locations in a matrix, the value being computed from the values of the four nearest neighbours in the grid. The communication is thus neighbour to neighbour. The grid is horizontally partitioned and only the rows at the boundary of each region are shared. Thus, each processor has two shared data items (in addition to the shared locks and barriers of course), one at each boundary of its grid partition. Each of them is shared with the corresponding neighbour. Although other implementations of SOR that have no false sharing or only read-write false sharing are possible on Indigo, we defined objects so the application has write-write false sharing. This allowed us to determine what effect a high degree of false sharing had on the performance of the application. Read-write conflicts are avoided by using a color-based algorithm that makes alternate processes compute odd and even positions in the matrix. All synchronization is barrier-based. The matrix in our problem was of the size 512×512 floats on the workstation cluster and 2048×2048 on the SP2.

   As an FSO and also in the MP implementation, the updated values are sent to the two neighbours (the communication structure used is a ring) and each process waits until it receives the new values from its neighbours before starting the next iteration. In the FSO, an event handler takes care of receiving the data and informing the local process when all the required data has arrived. It also does local buffer management, which is required as the neighbouring process might send in the next data values before the previous ones have been read by the local process.

4. *Embarrassingly Parallel (EP)*:
   Minimal synchronization, small messages. This is a Numerical Aerodynamic Simulation (NAS) kernel that is computation intensive. It evaluates integrals by means of pseudo-random trials and is used in many Monte-carlo simulations. Very little synchronization is required among the parallel processes. The only communication is towards the very end when all the processors participate in a reduction operation to generate a global sum, at which time, on ECM, locks are used to protect the value and a barrier to find out when everyone has completed. As an MP implementation, the worker processes inform a master process when they are done which then calculates the global sum as the last step. The EP FSO implementation is very similar to the MP one except that received values are immediately processed by handlers in FSO whereas an explicit receive must

be executed by processes in MP. A problem size of 33554432 was used on both the workstation cluster and the SP2.

We chose TSP because its message sizes are quite small and it has a very dynamic access pattern for locks. The performance of barrier-based synchronization was observed using the linear solver and the SOR applications, which also provided the other extreme of very large message sizes and predictable synchronization patterns. SOR also exhibited write-write false sharing which allowed us to evaluate our different FS-handling strategies. Finally, baseline performance is measured with an embarrassingly parallel application in which there is little sharing between nodes.

The ability to execute handlers when data movement operations are executed helps greatly in TSP, SOR and LIN implementations. For TSP, the handlers help achieve significant overlap between communication and computation. For the other two applications, they help as they receive new data from other nodes, buffer the received data and notify the process when all the data has been received. The local process can compute its values from the previously received data while this is being done. In contrast, in MP, receive calls are executed by the process only after the local computation is over.

We now discuss the performance of these four applications when they are programmed with ECM and FSO on top of Indigo on two different platforms - a cluster of Sun workstations and the IBM SP2 machine. Their performance is also measured when they are programmed directly on PVM. The latter allows us to quantify the overheads introduced by Indigo as well as the consistency-related activities of ECM and FSO. The role of active messages (used by FSO implementations) in improving performance is then discussed. Finally, there is an evaluation of the affect of the granularity of shared objects on the results and how handling of false sharing impacts application performance.

While performing the evaluation, it should be kept in mind that the extra cost in ECM is due to, firstly, the consistency-related actions that must be performed, and secondly, due to Indigo overheads. For FSO, some of the Indigo overheads are offset by the use of active messages and the related type-specific event-handling, plus communication-computation overlap in some cases.

4.2 Results on a Workstation Cluster

The purpose of the measurements presented in this section is to evaluate how well the two state-sharing techniques (ECM and FSO) perform as compared to message passing. We also need to explain the differences, if any, and to infer what effects the Indigo system had on the performance of the applications. These measurements are done on network of Sun Sparc2s on a 10Mbps Ethernet. The hardware page size was 4 Kbytes and the main memory size of the nodes varied from 24-64 Mbytes. PVM [35] was used as the underlying communication medium. Though it is not the fastest such medium available, our aim was to understand the issues involved in building a system such as ours and thus, the ease of programming with PVM helps. We ran the applications on 1-8 nodes.

Raw Indigo overheads. The raw times for the commonly used Indigo primitives are given in Table 1. These times account for the software overhead due to Indigo primitives. They do not include the time that is spent inside PVM packing and unpacking the actual data sent, the protocol processing time or the time spent on the wire. Most of the Indigo overhead is in table lookups and buffer management. The measurements are averaged over several hundred thousand calls to each primitive on a network of two machines. As we see from the table, for a shared data size of 4 bytes, the Indigo layer overhead of *purge* is 130 microseconds; *sync_put* and *sync_get* overheads are 210 and 250 microseconds, respectively. On the other hand, the total time needed to execute a *sync_get* call, which includes a PVM message send and receive, when 4 bytes are transferred is 8 milliseconds. Thus, the Indigo overhead is very low compared to the communication times required for data transfers across nodes.

Purge	Sync_put	Sync_get
130	210	250

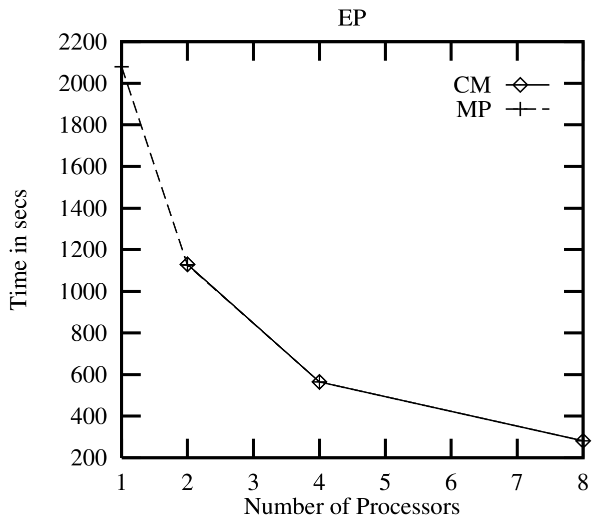
Table 1: Indigo overhead for various primitives, in microseconds

Completion Time Breakdown. The completion time of an application is measured at each node executing the application code and, for ECM, consists of three parts - *comp* for computation time, which is the actual CPU time spent executing user code at each node, *sync* for synchronization time, that is, the time spent by a process waiting to acquire a lock or for a barrier release, and *comm*, the communication time, which is the time spent in sending data or waiting to receive data if the cached data is out of date. The latter two are upper bounds on the actual *sync* and *comm* times as the machine load at the time of the experiment may cause some positive perturbation in their measurement (since they are measured as the difference between the wall clock times of message sends and receives, which would increase if other processes were running while the application process was in the ready queue).

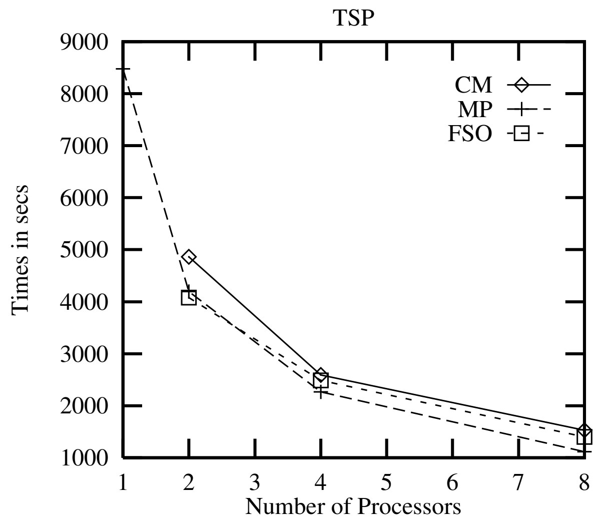
For the FSO and MP implementations, the partition of non-computation time into *sync* and *comm* is difficult to classify, as both these are interwoven into one-another. For example, in an MP implementation of LIN, the time spent waiting for the updated x-vector to arrive from all the other nodes counts as both synchronization and communication time. Hence, we measure only *comp* and the completion times for the MP and FSO implementations.

4.2.1 Completion Times

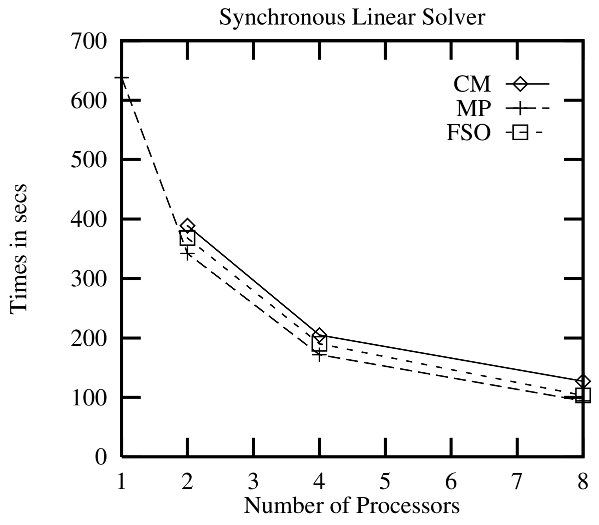
Figure 11 presents completion times for the four applications implemented on ECM, FSOs and MP. The times are per node and in seconds. The single node times are uniprocessor times directly on top of Unix when synchronization calls are removed, and the multi-node times for MP are implementations that directly use PVM as the message-passing layer.



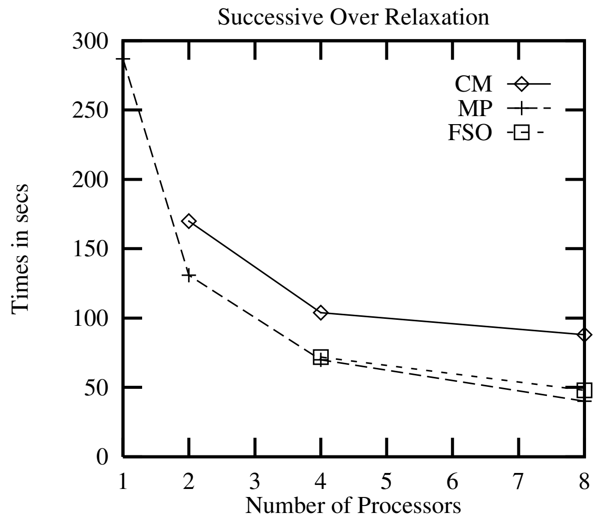
(a) Embarrassingly Parallel



(b) TSP - 17 cities



(c) LIN



(d) SOR

Figure 11: Completion Times

As the the completion time curves demonstrate, for EP and TSP, which have a high computation to communication ratio, good speedups are obtained on all implementations. In the case of the synchronous linear solver, the synchronization is barrier-based (hence more time is needed for synchronization as the number of nodes are increased) and message sizes are relatively large (for example, in the 4-node solution on ECM, the message transmitting the updated x-vector from one node to the other is 2 Kb in size). Thus, the speedup is limited by the time spent in synchronization and communication. The same holds for SOR, which also suffers because its messages do not get shorter as the number of nodes solving the problem increase, as is the case with LIN. Moreover, in general, since the computation time between synchronization points decreases for a larger number of processes solving the problem, the speedup also decreases as the number of processes is increased. Since the completion times of EP are practically indistinguishable on the three systems (which should be case because it exhibits little

# machines	2	4	8
TSP	7	3.5	2
LIN	6	9.5	17
SOR	23	23	27

comm

# machines	2	4	8
TSP	6	3.8	4
LIN	7	9	15
SOR	5	10	21

sync

Table 2: *Comm* and *sync* times for 2, 4 and 8 nodes, on ECM, in seconds

sharing), we discuss details of the other three applications.
TSP. For TSP, ECM is 15-35% slower than MP (for 2 and 8 nodes respectively), and FSO is, at best, 3% better (2 nodes), and at worse, 25% worse (8 nodes). As shown in (Table 2) for TSP, communication times reduce as the number of processors increase for ECM. That is to be expected as the number of times a processor approaches a server to acquire a lock to dequeue a tour decreases as more processes execute the application. The synchronization time also reduces for the same reason. ECM takes longer than the other two for TSP as the best tour value does not get propagated quickly (the old value is invalidated and a new one is fetched only when a lock is acquired).
LIN. In the case of LIN, the FSO curve matches the MP curve closely but ECM has higher completion times. This is because synchronization here is barrier-based and message sizes are relatively large. Thus, ECM performance is limited by the time spent in synchronization calls. For two nodes, ECM has 13% higher completion time than MP and FSO's completion time is 7% more than MP. For four nodes, the corresponding numbers are 19% and 10%, and for eight nodes, they are 35% and 9%. The ECM times are higher than the others not due to Indigo but, firstly, due to the synchronization overhead, and secondly, due to the particular implementation of ECM. More specifically, in this implementation, when a barrier is completed, the data at all nodes is invalidated. As a result, all nodes request data from the respective servers at the same time, leading to increased communication and synchronization times. The *sync* and *comm* times shown in Table 2 support these claims.
SOR. For SOR, MP and FSO's performances are quite identical. ECM, however, does not have completion times close to either of these systems. First, simple barriers are used in the ECM implementation whereas FSO and MP require a process to synchronize only with its neighbours. Second, due to false sharing among processes in the case of ECM, the servers suffer from some amount of extra computation overhead (see Section 4.2.3).

In the case of SOR and LIN, the communication times do not drop as the number of nodes increase, as was the case with TSP. This is because each node executes the same number of iterations independent of the number of machines. Thus, they still receive the same amount of data from the server. The main reason for the gap between the ECM and the other curves here is the extra communication (which also results in additional work for the servers). Also, in applications that use barriers, we observe an almost linear increase in the synchronization times as we increase the number of nodes (Table 2). This is to be expected as the in the case of barriers, each processor has to wait for more nodes to arrive at the barrier, and hence the average waiting time increases.

The effect of consistency mismatch. As seen in almost all the applications, ECM performed worse than both FSO and MP, though in some cases, it came quite close. This was mainly due to the extra communication and synchronization that is an inherent part of a DSM system. For TSP, where the messages are very small and the communication and synchronization cost are very slight, ECM follows the FSO and the MP curves closely. For LIN and SOR, where ECM suffers due to barrier synchronization and some extra messages, especially in the latter where FS is present, the differences in completion times are more pronounced.

Using structure and type information. How much impact did exploiting structure and type information (like in our DSA implementations) have on the completion times? FSOs provide performance very close to MP (or even better than MP) because remote invocations are executed independently of the user process at that node, thus providing great flexibility in using the invocation itself to perform other functions and computations, like making more invocations or manipulating the values in the memory space of the local process. The gains are quite obvious from the graphs. FSOs for our four applications perform consistently near their message-passing equivalents, as can be seen from the completion time plots where the FSO curves are very close to the MP curves for all the applications. For TSP, on two nodes, *the FSO actually performs better than MP.* This is because of the presence of active messages - the best tour value gets automatically updated on other nodes without the process having to do a receive as in the case of MP, and this results in quicker pruning of the search tree. Moreover, as we explained earlier, in our TSP implementation, we overlap communication with local computation, thus resulting in lower wait latency for the next jobs in the work queue.

The effect of Indigo overheads. The fact that FSOs, which are built on Indigo, provide performance almost identical with MP shows that Indigo overheads are insignificant. This is also to be seen for ECM for EP and TSP applications. In fact, the differences in completion times between ECM and MP for these three applications are similar to differences between these systems when ECM is implemented at the operating system level [20]. For the other applications, the loss of performance in ECM is due to synchronization and false sharing. Thus, we conclude that Indigo allows multiple systems to be implemented without having a negative impact on performance.

4.2.2 Effect of Granularity Mismatch

We now use the Indigo facilities to explore the effects of granularity mismatch. If a DSM system provides consistency at the level of a page, there may be unnecessary data transfers, as a page (typically 4 Kbytes) is transferred even when a small amount of data on that page is shared. Other systems provide consistency at the level of user-defined objects. In this section, we explore the performance gains made possible by using the latter strategy.

The increased communication times for our applications' page-based implementations shown in Table 3 demonstrate the importance of user-defined object sizes. The effect is quite pronounced with LIN and SOR, as the table shows. SOR suffers a little more as the number of messages it sends are much more than in the case of TSP or LIN and hence the effect is more obvious. Larger messages also cause bottlenecks at the server, thus slowing down the execution even more.

	TSP	LIN	SOR
Object	2	17	37
Page	4	37	92

Table 3: Communication times for 8 nodes on ECM, in seconds

4.2.3 Handling False Sharing

An approach that maintains consistency at the level of user-defined objects and not pages will suffer from little or no false sharing (FS) (such an assumption has been made in Midway [7] that provides consistency at the level of user-defined objects). However, it is not possible to completely eliminate FS [8]. Our implementation makes the assumption that FS will not be common, and hence we optimize for the case of no FS. That is, in our implementation which does deal with FS, applications with no FS will not pay any penalty for its handling. We ensure this by using *server-diffs*, instead of the more commonly used *client-diff* technique.

To evaluate server-diffs, processors were made to share objects in a test harness with different kinds of FS: write-write and read-write, along with the corresponding implementations of no falsely shared objects. In the case of write-write FS (WWFS), a single object was shared between all the processors which wrote into different parts of it and then synchronized at a barrier. No-WWFS means that FS was removed by having as many objects as the number of processors, which wrote only into their "own" objects. For read-write FS (RWFS), again, there were as many shared objects as the number of processors (let's say n). A part of each was was written by the corresponding processor, which then read the not-written-into parts of the other $(n-1)$ objects. The corresponding No-RWFS had $2n$ shared objects, with each processor writing into one and reading n others, none of which have been written into. In each program above, a hundred iterations were performed, all synchronization being barrier-based.

The test harness was executed for both the client-based and server-based diff schemes. The results are tabulated in Table 4. We also present the completion times of the three applications (LIN, SOR and TSP) in Table 5 when the two FS handling techniques are used.

For the case of no FS, the server-diff scheme performed better than the client-based scheme. When there is FS, however, which one performs better is application-dependent, as we discussed in Section 3.3. We expect the client-based-diff approach to perform better than the server-based approach for applications that send large messages and do not modify a large portion of the object. This is because the message size in

	WWFS	No WWFS	RWFS	No RWFS
Server	26	12	53	56
Client	20	15	53	61

Table 4: Server- vs. Client-based diffs, completion times, test harness, 8 nodes

the former would, on an average, be less than what it would be in the latter (since the message from the client has size roughly proportional to the extent of modification in client-based diff but is always equal to the object size in server-based diff). SOR is such an application, and the reverse is true for LIN. In LIN, each message is 106 bytes larger on an average in the client-based diff approach as compared to the server-based diff approach (it is of a fixed 4096 byte length in the latter, for two nodes). In SOR, client-diff results in messages which are approximately 1974 bytes *shorter* on an average, for eight nodes. As we see in Table 5, the completion times change as expected (the main component changing here is the communication time). Also, we observe no change for the two cases for TSP, as the shared data size is quite small in this case, and hence the copying and diff overheads are minimal, besides resulting in small messages (the message size is shorter in the client diff case by around 11 bytes on an average, for eight nodes).

In summary, these results are consistent with our goal that with no false sharing, our implementation of ECM on top of Indigo avoids the copying, buffering and communication overheads which could improve performance. With FS, the results depend on the degree of FS and the pattern of modification of the data within the application, though they are still close to the numbers with client-diffs.

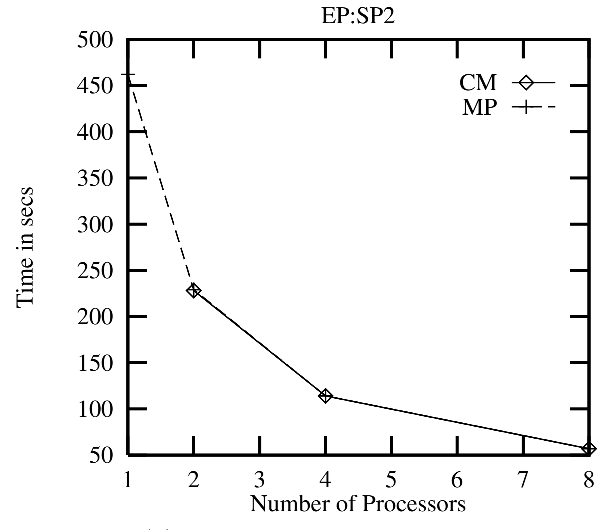
	LIN	SOR	TSP
Server	389	95	1441
Client	397	88	1441

Table 5: Server- vs. Client-based diffs. Completion times are for number of nodes showing maximum difference between the two cases on Sparc2s.

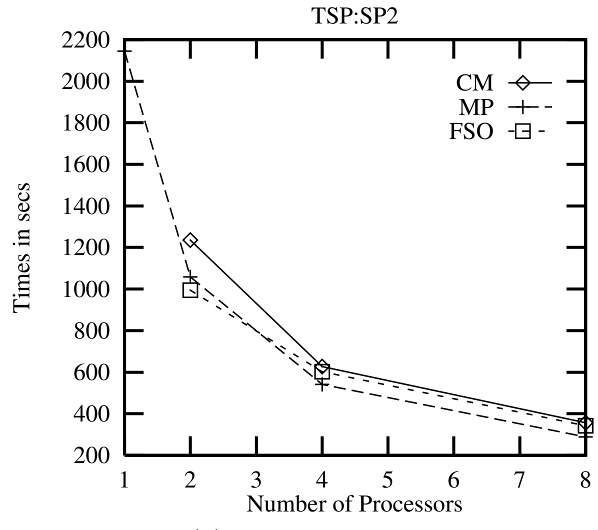
4.3 Results on the SP2

The results on the IBM SP2 are still preliminary. It appears at this point that due to the faster processor and communication speeds, a number of the results from the Sun workstations are somewhat modified on the SP2 though the general characteristics do not change significantly. This section presents a summary of the results of executing the same four applications on an 8-(RS6000-)processor SP2. Ethernet was used as the communication medium and not the high performance switch due to its unavailability when these experiments were being performed. The hardware page size was 4 Kbytes and the main memory size was around 100 Megabytes on each node. The applications were executed on 1-8 nodes.

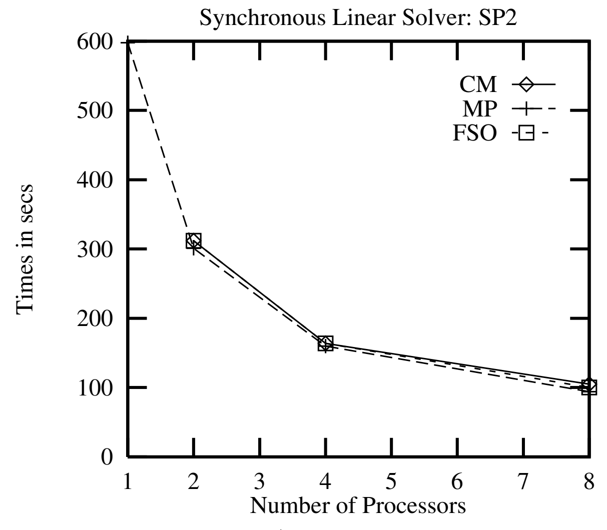
Figure 12 presents completion times for the four applications implemented on ECM, FSOs and also directly on PVM, all running on the SP2. As the RS6000s are much faster than the Sparc2s, we have implemented larger problem sizes for the SOR and LIN applications. SOR is now being computed for a 2048×2048 (up from 512) matrix, and LIN for 4096 unknowns (up from 2048).



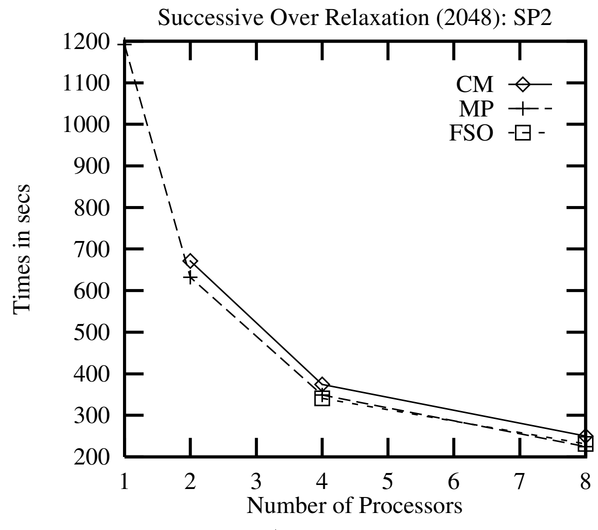
(a) Embarrassingly Parallel



(b) TSP - 17 cities



(c) LIN



(d) SOR

Figure 12: Completion Times

As seen from the figure, all the curves now are quite close to each-other. For TSP, ECM is 15-23% slower than MP. The FSO does much better again - it is 6% *faster* in the best case (two nodes) and 18% slower than MP, in the worst case. For LIN, ECM varies between 2-12% of the MP implementation, and the FSO lies between 2-6%. For SOR, the corresponding numbers are 6-11% and $\pm 2\%$ - the FSO is up to 2% *faster* than

the corresponding MP implementation, mainly due to a computation-communication overlap.
The SP2 effect. The performance of the applications is closer to each-other on the three systems on the SP2 compared to the workstation cluster. The gap between the various systems is reduced because it is primarily due to the communication time which is much smaller on the SP2 (mainly because a large part of the cost of Ethernet communications in the software layers). These results again demonstrate that using type and structure information as in FSOs leads to better performance than using DSM.

5 Related Systems

A number of systems that can support state sharing on a workstation cluster (for example, Ivy [28], Treadmarks [22], Midway [7] and Distributed Filaments [15]) have been proposed and implemented. The goal of these systems has mainly been to implement a particular memory model (like Entry Consistency in Midway) and not to provide a general framework like Indigo that allows many different consistency models as well as general shared abstractions to be implemented. We have implemented shared abstractions which go beyond just memory models, thus allowing us to quantify the gains in the performance of the applications when type information is exploited. Thus we are able to provide system support that matches the needs of various state sharing techniques that range from memory with various consistency needs to FSOs. Moreover, Indigo is portable as it runs on PVM which has been ported to a number of systems.

Shapiro [34] proposed a system in which objects fragmented across nodes of a distributed system could be implemented. This work is similar to topologies [33], which is the approach we take for implementing distributed shared abstractions. Indigo provides lower level support for data movement between shared abstraction caches and events that could be used to keep the caches consistent. The object fragments and the communication structure between them can be implemented using Indigo calls. In addition, we explored shared abstractions for computational problems and compared their performance with message passing and a DSM system. In contrast, coarse-grain objects such as replicated files are investigated in [34].

Tempest [19] also provides mechanisms for communication and synchronization in a parallel program. It differs from Indigo which considers access detection techniques needed by DSM implementations to be a matter of policy and thus a responsibility of the DSM layer whereas Tempest provides support for it. Nexus [14] supports distributed objects but it does not provide support for programming shared abstractions over a distributed system. It also does not address caching and consistency issues because an object resides at a single node.

6 Conclusion

This paper presents the design and implementation of a user-level library called Indigo that facilitates sharing of state across nodes in a distributed system. Indigo provides various primitives for data movement and active handling of messages. To evaluate it, a distributed shared memory system and several shared abstractions were programmed and used to explore several issues that impact the performance of applications programmed on a DSM system, such as the effect of consistency and granularity mismatch between the application and the communication layers. In addition, a new approach is evaluated for handling false sharing, in which programs which do not exhibit false sharing do not pay any penalty for handling it. Our results with fragmented shared objects also demonstrate that utilizing type and structure information leads to significant gains in performance of distributed applications.

There are many interesting ways in which the functionality of Indigo can be extended. We are presently considering whether to add thread support, or to add RPC mechanisms. There is also a plan to run the system on a larger number of nodes to test it for scalability and to observe any effects it has on the performance of our applications. Platforms other than PVM will also be considered in our future implementations. We are currently in the process of designing and building a system that supports configurable objects for high-performance applications over a distributed platform.

References

[1] Sarita V. Adve and Mark D. Hill. Weak ordering - a new definition. In *Proceedings of the 17th Annual International Symposium on C omputer Architecture*, pages 2–14, May 1990.

[2] Sarita V. Adve and Mark D. Hill. A unified formalization of four shared-memory models. Technical Report Computer Sciences Technical Report 1051, University of Wisconsin, Madison, September 1991.

[3] Mustaque Ahamad, Rida Bazzi, Ranjit John, Prince Kohli, and Gil Neiger. The power of processor consistency. In *Proceedings of the 5th ACM Symposium on Parallel Algorithms and Architectures*, 1993.

[4] Mustaque Ahamad, Gil Neiger, Prince Kohli, James E. Burns, and Phillip W. Hutto. Causal memory: Definitions, implementation and programming. *Distributed Computing*, 9(1):37–49, Aug 1995 appear.

[5] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. Experience with distributed programming in Orca. In *In Intl. Conf. on Computer Languages*, 1990.

[6] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the 2nd ACM Symposium on Principles and Practice of Parallel Programming*, pages 168–177, March 1990.

[7] Brian N. Bershad and Matthew J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, Carnegie Mellon University, September 1991.

[8] William J. Bolosky and Michael L. Scott. False sharing and its effect on shared memory performance. In *4th Symposium on Experimental Distributed and Multiprocessor Systems*, pages 57–71, September 1993. Also available as MSR-TR-93-1, Microsoft Res. Lab., Sep. 1993.

[9] J. S. Chase, F. G. Amador, E. D. Lazowska, H. M. Levy, and R. J. Littlefield. The amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the 12th Symposium on Operating System Principles*, 1989.

[10] Christian Clemencon, Bodhisattwa Mukherjee, and Karsten Schwan. Distributed shared abstractions (DSA) on large-scale multiprocessors. In *Proc. of the Fourth USENIX Symposium on Experiences with Distributed and Multiprocessor Systems*, pages 227–246. USENIX, September 1993. Also as TR# GIT-CC-93/25.

[11] A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwaenepoel. Software versus hardware shared memory implementation - a case study. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, 1994.

[12] Partha Dasgupta, Richard J. LeBlanc, and William F. Appelbe. The Clouds distributed operating system: A functional description, related work and implementation details. In *Proceedings of the Eighth International Conference on Distributed Computing Systems*, June 1988.

[13] Ed Felton. Best-first branch-and-bound on a hypercube. In *Third Conference on Hypercube Concurrent Computers and Applications, Pasadena, CA*. ACM, Jet Propulsion Laboratories, Jan. 1988.

[14] I. Foster, C. Kesselman, and S. Tuecke. The Nexus task-parallel runtime system. In *Proceedings of the First International Workshop on Parallel Processing*, 1994.

[15] Vincent French, David Lowenthal, and Gregory Andrews. Distributed filaments: Efficient fine-grain parallelism on a cluster of workstations. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, November 1994.

[16] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[17] James R. Goodman. Cache consistency and sequential consistency. Technical Report 1006, University of Wisconsin, Madison, February 1991.

[18] X Consortium Working Group. Fresco specification draft version 0.7, April 1994.

[19] Mark D. Hill, James R. Larus, and David A. Wood. Tempest: A substrate for portable parallel programs. In *COMPCON*, March 1995.

[20] Ranjit John and Mustaque Ahamad. Evaluation of Causal Distributed Shared Memory for Data-race-free Programs. Technical Report GIT-CC-94/34, College of Computing, Georgia Tech, Atlanta, GA 30332-0280, 1994.

[21] Peter Karlin, Mani Chandy, and Carl Kesselman. The compositional C++ language definition. Technical Report CS-TR-92-02, Califonia Institute of Technology, March 1993. Revision 0.95.

[22] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. Technical Report Rice COMP TR93-214, Department of Computer Science, Rice University, 1993.

[23] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th International Symposium of Computer Architecture*, 1992.

[24] Prince Kohli, Gil Neiger, and Mustaque Ahamad. A characterization of scalable shared memories. In *Proceedings of the 22nd International Conference of Parallel Processing*, August 1993.

[25] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[26] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[27] R. Lea, C Jacquemot, and E. Pillevesse. COOL: system support for distributed programming. *Communications of the ACM*, 36(9):37–46, Sept. 1993.

[28] Kai Li. Ivy: A shared virtual memory system for parallel computing. In *International Conference on Parallel Processing*, pages II 94–101, Aug 1988.

[29] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM TOCS*, 7(4):321–359, November 1989.

[30] Richard J. Lipton and Jonathan S. Sandberg. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Princeton University, Department of Computer Science, September 1988.

[31] F. Mattern. Virtual time and global states of distributed systems. In *Proceedings of the 1988 International Workshop on Parallel and Distributed Algorithms, Bonas, France*. North Holland, 1989.

[32] J. G. Mitchell, J. G. Gibbons, G. Hamilton, P. B. Kessler, Y. A. Khalidi, P. Kougiouris, P. W. Madany, M. N. Nelson, M. L. Powell, and S. R. Radia. An overview of the spring system.

[33] Karsten Schwan and Win Bo. Topologies — Distributed objects on multicomputers. *ACM Transactions on Computer Systems*, 8(2):111–157, 1990.

[34] M. Shapiro. Structure and encapsulation in distributed systems: The proxy principle. In *Sixth International Conference on Distributed Computing Systems, Boston, Mass.*, pages 198–204. IEEE, May 1986.

[35] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2:315–339, December 1990.

[36] A. Tanenbaum and S. Mullender. An overview of the amoeba distributed operating system. *Operating Systems Review*, 15:51–64, July 1981.

[37] Deborah Wallach, Wilson Hsieh, Kirk Johnson, Frans Kaashoek, and William Weihl. Optimistic active messages: A mechanism for scheduling communication with computation. 1994.

[38] M. Zekauskas, W. Sawdon, and B. Bershad. Write detection for distributed shared memory. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, November 1994.