The Amber System: Parallel Programming on a Network of Multiprocessors

Jeffrey S. Chase, Franz G. Amador, Edward D. Lazowska
Henry M. Levy and Richard J. Littlefield

Department of Computer Science
University of Washington
Seattle, WA 98195

Abstract

Microprocessor-based shared-memory multiprocessors are becoming widely available and promise to provide cost-effective high-performance computing.

This paper describes a programming system called *Amber* which permits a single application program to use a homogeneous network of multiprocessors in a uniform way, making the network appear to the application as an integrated, non-uniform memory access, shared-memory multiprocessor. This simplifies the development of applications and allows compute-intensive parallel programs to effectively harness the potential of multiple nodes.

Amber programs are written using an object-oriented subset of the C++ programming language, supplemented with primitives for managing concurrency and distribution. Amber provides a network-wide shared-object virtual memory in which coherence is provided by hardware means for locally-executing threads, and by software means for remote accesses. Amber runs on top of the Topaz operating system on a network of DEC SRC Firefly multiprocessor workstations.

1 Introduction

Small-scale shared-memory multiprocessors are becoming widely available in implementations ranging from single-user workstations to mini-supercomputers [Thacker et al. 88, Fielland & Rodgers 84, Downey 88]. Two factors working together are responsible for this trend. First, microprocessor performance has increased at a remarkable rate. Second, the cost of the microprocessor is a small

This paper was published in *Proc. of the 12th ACM Symposium on Operating Systems Principles*, December 1989, pp. 147–158. This material is based on work supported by the National Science Foundation (Grants CCR-8611390, CCR-8619663, CCR-8700106 and CCR-8703049), the Naval Ocean Systems Center, U S WEST Advanced Technologies, the Washington Technology Center, and Digital Equipment Corporation (the Systems Research Center, the External Research Program, and the Graduate Education Program).

part of total system cost. Within limits, adding microprocessors to a system can substantially increase performance at little additional cost.

The proliferation of multiprocessors means that local networks of these systems are likely to become common. This presents the opportunity to program a group of these machines to work together on a single application. In the same way that a small-scale multiprocessor has a cost advantage over a uniprocessor of equivalent performance potential, so a network of small-scale multiprocessors has a cost advantage over a single multiprocessor of equivalent performance potential. Networks of small-scale multiprocessors are likely to have greater performance potential than the fastest mainframes at significantly lower cost, and with greater flexibility.

This paper describes a programming system called *Amber* that exploits this technology trend. Specifically, Amber allows a parallel application program to treat a network of multiprocessors as an integrated, non-uniform memory access (NUMA), shared-memory multiprocessor. We assume that the goal of an Amber programmer will be to achieve maximum application speedup in this environment. To attain this goal, programmers will need to be aware of and will wish to take advantage of the network organization. Therefore, Amber's abstractions are intended to simplify communication, distribution, and parallelism, while supporting a dynamic program structure that can express and benefit from locality.

While we expect that a wide variety of applications will run effectively on Amber, the system is well suited to problems that partition into parallelizable clusters, where there is strong interaction within clusters and weak interaction between clusters. Amber also can be used as a general-purpose distributed programming system or as a parallel programming system on a single multiprocessor, and indeed owes parts of its heritage to systems of each of these types.

Amber is based on an object-oriented model of computation in which collections of dynamic, mobile objects distributed among nodes in a network interact through location-independent invocation. Amber programs are written using an object-oriented subset of the C++ programming language [Stroustrup 86], to which Amber adds primitives for thread management, synchronization, and object mobility. The Amber system is composed of a preprocessor to C++ and a run-time library, referred to as the Amber *kernel*, which is linked with the user's code. Amber runs under the Topaz operating system on a network of DEC SRC Firefly multiprocessor workstations [Thacker et al. 88].

While there are many dimensions to the Amber system, its essentials are easily grasped in terms of its relationships to several other systems:

- The distribution model of Amber is similar to that of Emerald [Jul et al. 88], a distributed object-oriented programming system that includes support for fine-grained object mobility. Amber differs from Emerald in its design for concurrency and virtual memory. It also relies on C++ rather than a special-purpose programming language, making Amber more attractive to application programmers and eliminating the need for a compiler.

- The concurrency model of Amber follows that of Presto, a C++-based system for building medium-grained parallel applications on shared-memory multiprocessors [Bershad et al. 88a]. Key properties of Presto include efficient support for threads, locking, and scheduling, and an open approach in which the application programmer can easily customize the programming model [Bershad et al. 88b]. Presto, however, has no facilities for distribution.

- The attractiveness of the architectural model – a large-scale shared-memory multiprocessor built in software from a network of workstations or small-scale multiprocessors – was first suggested by Kai Li, through his work on the Ivy distributed virtual memory system [Li 86, Li & Hudak 86]. Amber can be thought of as a distributed virtual memory system in which the unit of coherence is the object, while Ivy is a distributed virtual memory system in which the unit of coherence is the page. Section 4 discusses several important differences between these two views of distributed virtual memory.

- The organization of Amber programs into closely-cooperating clusters is similar to the task force structure in the Medusa [Ousterhout et al. 80] and StarOS [Jones et al. 79] operating systems for Cm* [Gehringer et al. 87], and to Argus guardians [Liskov 88]. However, on Amber this clustering is determined at run time and can change dynamically as a program executes. Locality is more crucial to Amber than to the Cm* systems because the ratio of remote invocation time to local invocation time is three orders of magnitude in Amber, compared to one order of magnitude on Cm*.

In many ways Amber is a synthesis of ideas; however, Amber is unique in that it specifically supports the programming of networks of multiprocessors through an object-oriented distributed virtual memory and explicit data and thread migration.

Amber explores issues in parallel programming at both the system level and the application level. At the system level, Amber investigates the viability of using a loosely-coupled network of small-scale multiprocessors as a large-scale machine. This raises issues in scheduling, virtual memory management, distribution, and coherency. At the application level, Amber examines issues in application structuring. Here we wish to understand the programming primitives needed to express both locality and parallelism. Overall, Amber explores the appropriateness of the object-oriented programming model to solve problems at both levels.

The remainder of this paper presents Amber's design, implementation, and performance in detail. Section 2 describes the Amber programming model. Section 3 presents a specific application to demonstrate how the Amber model is used. Section 4 discusses the design and implementation of the Amber distributed virtual memory system. Second 5 describes several other design and implementation issues that arise in building a system that is both distributed and parallel. Section 6 presents performance results. Section 7 summarizes our work.

2 The Amber Programming Model

An Amber program is structured as a collection of cooperating objects. An object may have private data, public operations that can be locally or remotely invoked, and private operations that can be invoked only by the object itself. Most objects are passive data entities; they execute only as a result of invocations. The active entities in the system are *thread* objects. Thread objects are the building blocks of parallel Amber programs; they are the units of execution and concurrency. A typical Amber application will contain a large number of threads executing concurrently on different processors within the same node and on different nodes in the network.

Amber provides the user with a set of primitive object classes that are the means by which distribution, concurrency and synchronization are controlled. The mechanisms for expressing parallelism follow the Presto model for integrating threads into an object-oriented language. Distribution in Amber is achieved by object mobility primitives and location-transparent object invocation. The Amber programming model is uniform in the sense that it is unnecessary for the programmer to deal explicitly with the locations of objects when they are invoked. Data objects move around the network under program control, while threads migrate automatically as invocations move from object to object.

Amber nodes share a single virtual address space, globally managed so that virtual addresses have the same meaning on all nodes. Object references can be transmitted across node boundaries and dereferenced on any node with consistent semantics, allowing programs to maintain and operate on distributed data structures in a uniform way. The use of a distributed global address space also allows shared memory paradigms to be used for addressing and communication in order to enhance performance. We expect the programmer to cluster threads and data appropriately to obtain maximum benefit from the use of shared memory.

Object-orientation is crucial to Amber because of the encapsulation that it provides. In an object-based system, the only way for a thread to manipulate or examine an object's state is to invoke an operation on the object. Runtime support code and checks inserted by the Amber preprocessor ensure that a thread which invokes a remote object is transparently moved to the node where the invoked object resides. This guarantees that all threads executing operations on a particular object can address the object's contents directly, with consistency maintained efficiently by hardware-based synchronization primitives and memory coherence protocols. The abstractions provided by object classes hide not only the representation of objects but also the internal details of their execution, synchronization, and location.

In contrast to many other object-oriented systems [Allchin & McKendry 83, Almes et al. 85], the goal of Amber is to execute a single application that performs a (parallel) computation, computes a result, and terminates. We do not attempt in Amber to provide an environment for persistent objects, stable storage, or communication and cooperation between programs.

2.1 Thread Objects

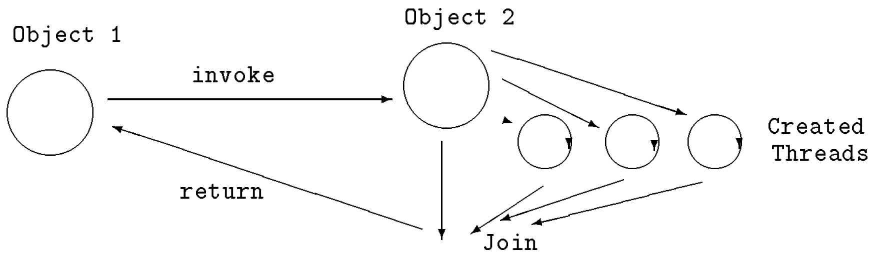
Most Amber objects are passive entities. Thread objects are unique in that they possess processor state and a runtime stack and can execute on a CPU. Like other objects, threads export operations that can be invoked to control their activity. The basic primitives for operating on threads are *Start*, which starts a newly-created thread executing an operation in a specified object, and *Join*, which blocks the caller until the thread terminates. The *Join* primitive also returns the result of the thread's execution; the result is simply the return value of the operation specified in the *Start* invocation. Threads also have operations to store and retrieve an arbitrary object reference, which is useful for maintaining a private block of data that is independent of the thread's stack.

Once a thread is started via the *Start* primitive, the runtime system places it on the run queue of the node on which the invoked object resides. From there it is executed as soon as a processor becomes available. The Amber system includes a customizable thread scheduling mechanism that supports timeslicing and can be used to implement priority-based or adaptive scheduling algorithms tuned to the specific application. When a new thread is scheduled it starts executing the body of the specified operation using the arguments passed to *Start*. The code in the object operation may invoke operations in other objects, which may create additional threads, and so on.

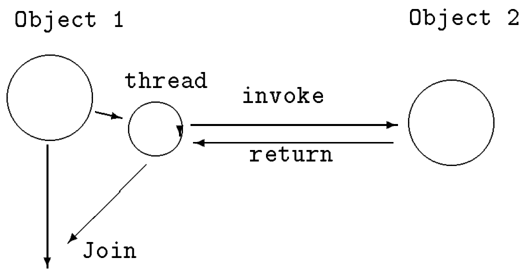
Threads are used for parallelism and asynchrony, while invocations are used exclusively for synchronous communication. An invoked object can exploit parallelism transparently to its invoker by creating and starting additional threads in response to an invocation. Alternatively, an invoking object can exploit parallelism by creating a thread to perform the invocation, effectively executing an asynchronous invocation through this mechanism. This is shown pictorially in Figure 1.

2.2 Synchronization Objects

Amber provides a flexible set of classes for controlling access to data shared by multiple threads: relinquishing and non-relinquishing locks, barrier synchronization, condition variables, and atomic integers. Programmers can take advantage of the extensibility properties of C++ to define customized higher-level mechanisms for concurrency control using the predefined primitives. The



Synchronous Invocation of Parallel Object



Asynchronous Invocation with Parallel Thread

Figure 1: Amber Invocations Using Threads

intent is that programmers will select an appropriate concurrency control scheme for each user object class, and encapsulate the details of the synchronization within the class definition. This serves the interests of modularity and abstraction because it allows invokers of objects to be isolated from the details of synchronization in the invoked object.

Because distribution is completely under the control of the programmer, it is possible for synchronization objects to be distributed and remote from their clients. Programmers must be careful to properly choose locking strategies even on a conventional shared-memory multiprocessor, but with Amber extra care is required because of the network latency involved in accessing locks remotely. Although it may seem strange to imagine spin-waiting on a remote lock, a remote non-relinquishing lock will work properly if invoked. The invoking thread simply moves to the node where the lock resides and spins on one of the processors of that node, relinquishing the processor it was using on the source node. The only real difference between a local and a remote non-relinquishing lock is that the expected time both to acquire and to hold the lock will change due to the cost of the network communication, making remote spin-waiting a poor idea.

2.3 Controlling Object Location

Amber programmers take advantage of locality by using mobility primitives to control the locations of objects. In general, the advantage of object mobility over other data transfer schemes is that it does not place any data formatting or communication burden on the programmer. Objects can be moved dynamically as the program executes, and threads executing operations on a moving object are moved transparently with the object. This is useful because some applications will need to reorganize object locations following different computational phases of a program, although static object placement (i.e., creation-time placement without further mobility) might be sufficient for many applications.

Amber's mobility primitives are modeled after mobility in the Emerald system [Jul 88]. All Amber objects have a number of basic mobility operations defined for them. An object can be moved with *MoveTo* and its location can be determined with *Locate*. The programmer can also *Attach* an object to another object or *Unattach* an object which is attached. Moving an object automatically causes all of its attached objects to move along with it. The attachment primitives allow a programmer to dynamically create structures of objects that move together and are always guaranteed to be co-located. The attachment mechanism in Amber is more dynamic than in Emerald, where it is specified at compile time.

When an ordinary object is moved from a node it ceases to be resident there, causing future invocations of the object on that node to incur network communication overhead. This is necessary to avoid the difficulties of maintaining multiple copies of mutable objects. However, read-only objects can be safely replicated on multiple nodes. Amber allows objects to be designated as *immutable* for this purpose. The system assumes that immutable objects are never modified once they are initialized, although this restriction is not enforced. Invoking *MoveTo* on an immutable object causes the object to be copied rather than moved. This facility is useful for propagating read-only data across nodes in order to reduce communication costs. Immutable objects export an operation that replicates them on all nodes.

The notion of location in the Amber system is encapsulated in the *node* object. The kernel on each machine creates a resident node object at startup time to represent the node. The node class provides primitives for finding out how many nodes there are and for obtaining references to node objects. Node objects can be remotely invoked to obtain information about the node they are associated with. More importantly, references to node objects can be used to specify location to the mobility primitives. The *MoveTo* operation takes a target object reference as an argument, and moves the invoked object to the location of the target. Although any object can be used, specifying a node object as the target guarantees that the moving object will be placed on the named node. The *Locate* primitive returns a reference to a node object.

2.4 Achieving Good Application Performance

The goal of Amber is to provide a uniform model of programming, while simultaneously allowing the programmer to use program-specific knowledge to fine-tune program organization for a parallel and distributed environment. Of course, the means by which performance is tuned may require the programmer to deal explicitly with object location or the details of system functions such as remote invocation and thread scheduling. This tension between uniformity and performance is more pronounced in Amber than in conventional distributed systems because high performance for parallel applications is the primary purpose of the system.

Amber provides a number of mechanisms for tuning application performance. Examples include customizable thread scheduling, primitives for managing object placement and locality, and optimized forms of communication for objects known to be co-located. Amber also allows the programmer to control tradeoffs between latency and throughput in the distribution primitives. These facilities compromise the uniformity of the programming model somewhat and they place more burden on the programmer to use system primitives properly. In certain cases an inappropriate use of the system can cause incorrect behavior.

We believe that the drawbacks to these facilities are outweighed by their benefits for performance-critical parallel programming. This view of what a programming environment should do is consistent with accepted systems design goals of separating policy from mechanism and not hiding power [Levin et al. 75, Lampson 84]. The underlying assumption is that the needs of different applications vary so much that a more restricted model cannot possibly perform well in all situations.

We believe that programmers are willing to devote careful thought and effort to achieving the best performance, but that they want to think in terms of the problem structure rather than the low-level details of the hardware configuration the program is running on. Amber programmers can think in terms of problem-oriented units such as objects and threads rather than hardware-related units such as processors, memory pages and network packets. For example, Amber's cheap threads allow the programmer to manage more control streams than there are processors, making it easy to structure applications cleanly and to overlap local computation with remote communication. Also, object mobility and transparent remote invocations isolate the programmer from the details of message passing between nodes. Other examples of this principle will emerge in later sections.

In Amber, objects are the units of program locality as well as the units of data. Since the node that executes each object operation is determined by the location of the invoked object, the distribution of computational load between the machines is completely dependent on the locations of the objects involved in the computation. Interactions between co-located objects are several orders of magnitude less expensive than interactions that cross node boundaries, so the locations of objects have a significant effect on the communication cost incurred by the program. Because most objects are relatively small, it is often beneficial to move an object in order to avoid several remote invocations. In general, interacting objects should be co-located whenever it is reasonable to do so in order to avoid paying the cost of a remote procedure call on each invocation. This often conflicts with the goal of locating objects so as to evenly distribute the computational load between machines.

Effectively partitioning a program's data into objects and determining appropriate locations of those objects is a nontrivial task. Although Amber attempts to provide location-transparency wherever possible, it allows the programmer to exploit knowledge of program locality by explicitly controlling the locations of objects. In the future, automated tools may be capable of partitioning an application across multiprocessor clusters, but we believe that for most applications the best performance will be achieved by allowing the programmer to manage object location.

2.5 The Programming Environment

Thread objects, node objects, and synchronization objects form the basis of the facilities provided by Amber. All system-defined interfaces belong to a C++ class hierarchy made visible to the application by definition files that it imports. User programs are preprocessed, compiled by C++, and linked with the Amber kernel to produce an executable image.

To execute an Amber program, the user runs the executable image on a single node, specifying any command line arguments that may be needed. These arguments, which are passed to an initialization routine provided by the programmer, can specify run-time parameters such as the number of nodes to use. The Amber kernel then contacts the selected nodes; each node retrieves a copy of the program image from the Topaz remote file system and executes the image. Each node knows whether it is the originating node or a secondary node. As the nodes start, they execute a node initialization routine provided by the programmer. Once all the secondary nodes are initialized, the originating node creates a single thread to execute the user's main program. The user's main program can then create objects, move objects to other nodes, and start additional threads for parallel execution.

The Topaz operating system, on which Amber is implemented, supports multiple threads of control in a single task (address space). Amber programs execute as a set of cooperating Topaz tasks distributed across the network. One Topaz task executes the Amber image on each node. These tasks use the Topaz RPC facility to communicate. In each task, the Amber kernel creates enough Topaz threads to allow it to effectively use the processors on its node; that is, there is one

```
float T[NROW,NCOL], maxdif, omega, tol;
int color;
do {
  maxdif = 0.0;
  for (color = 0; color < 2; color++) {
    forall (r = 1; r < NROW-1; r++) {
      forall (c = 1+(r+color)%2; c < NCOL-1; c+=2) {
        float Tnew, diff;
        Tnew = (T[r-1,c] + T[r+1,c] + T[r,c-1] + T[r,c+1]) / 4.0;
        diff = ABS (Tnew - T[r,c])
        if (diff > maxdif) maxdif = diff;
        T[r,c] = T[r,c] + omega * (Tnew - T[r,c]);
      }
    }
  }
} while (maxdif > tol);
```

Figure 2: Red/Black SOR Algorithm

Topaz thread per processor on a node. However, there will typically be many more Amber thread objects than Topaz threads on a single node; Amber schedules these threads for execution by Topaz thread objects, so at any time one Topaz thread is executing one Amber thread.

3 An Amber Application

To make the discussion of Amber more concrete, we now describe an application: a program that computes the steady-state temperature distribution over the interior of a square plate, given the temperatures around the plate's boundary. The behavior of this system is governed by Laplace's equation, which states that the value at each point must be the average of the values of its neighbors. The specific algorithm that we employ is Red/Black SOR (Successive Over-Relaxation), an iterative method that parallelizes well and that is commonly used in practice.

This application constitutes something of a "stress test" for the Amber programming model. Numerical applications of this type are not naturally expressed in terms of multiple objects, and forcing the application into this framework was somewhat tedious. However, we shall show in a later section that excellent performance was achieved, and the program is a good tool for evaluating the system because it can be configured easily to exercise various features of Amber.

3.1 The Algorithm

The Red/Black SOR (Successive Over-Relaxation) algorithm can be understood by analogy to a checkerboard. Each point of the problem grid corresponds to a square on the checkerboard. During each iteration, all of the black points are updated first, followed by all of the red points. Since black points have only red neighbors and vice versa, each of the update phases is highly parallelizable. The algorithm is easily described in C-like pseudocode, shown in Figure 2.

3.2 Design Decisions

The obvious way to partition this algorithm for parallel execution is to break the grid into sections, assign each section to a different processor, and let the processors run simultaneously, synchronizing between phases. Given true shared memory this can be done quite easily. However, on a typical network with high latency and limited bandwidth, one must pay close attention to the data reference patterns in order to get good performance.

Some partitionings for the Amber implementation are clearly inappropriate. Placing the entire grid in one object would result in poor performance. Under the Amber paradigm, each object lives on only one node and threads migrate to that node to operate on the object; if the entire grid were in one object, then only the processors belonging to that node would do significant work. Placing each point in a separate object and locating the points belonging to each section of the grid on a single node also would result in poor performance. The implementation would work well over the interior of each section, since the objects corresponding to the adjacent points would be local, but on section boundaries, each reference to a point in a neighboring section would involve a separate remote invocation.

An effective partitioning is to place each section of the grid in an object, and to distribute these section objects in the network. The processing for each section object consists of (1) exchanging edge data with neighboring sections, (2) updating the points within the section, and (3) contributing to the global determination of `maxdif`. This organization makes it easy to transfer data for an entire edge of a section in a single invocation, which is crucial to performance. It would be difficult to implement a similar optimization using the point per object partitioning.

3.3 The Amber SOR Implementation

Our Amber implementation of this partitioning has several sets of threads associated with each section object. One set of threads is responsible for performing the computation within a section (since parallelism can be utilized by subdividing the section). Another set of threads is responsible for exchanging edge data with neighboring sections; for simplicity we use one thread per neighbor. Finally, one thread per section is responsible for communicating with a single *master thread* regarding the computation of `maxdif`. Figure 3 displays this structure for a three-section decomposition.

This design is quite efficient for several reasons. First, as noted previously, only a single function call is required to transfer an entire row or column of data between sections. This guarantees that each transfer needs only one network transaction, regardless of how data happens to be laid out in the address space. Second, data transfers can be overlapped with computation by running the respective threads in parallel. This reduces the effect of network latency. Third, computation threads within a section can freely divide work among themselves, without danger of causing network activity. This makes it easy to balance the computational load across multiple processors on a node. Fourth, the formulation has fairly large granularity, which reduces the effect of overheads such as thread scheduling.

The performance of this implementation on a network of eight DEC SRC Firefly multiprocessor workstations will be discussed in a later section.

4 The Distributed Virtual Address Space

An Amber application program executes within a single network-wide virtual address space. All code and read-only data objects are replicated at the same addresses on all nodes. All objects

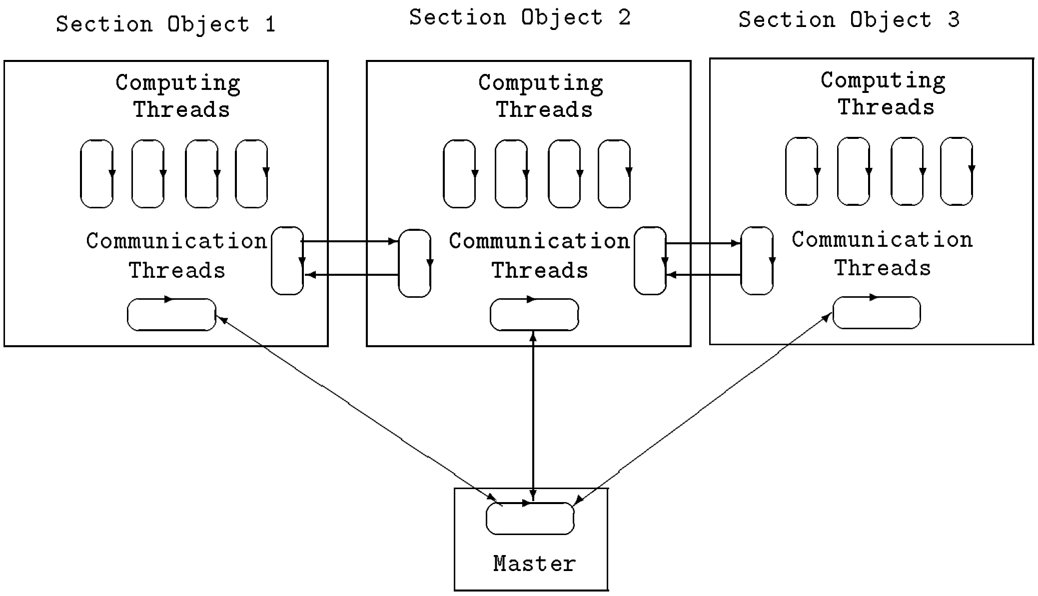


Figure 3: Structure of the Amber Red/Black SOR Implementation

including thread objects and their stacks are assigned a distinct segment of the global address space when they are created, and each object occupies this same virtual address range on any node that it visits during its lifetime. The segment of virtual memory occupied by an object on one node is guaranteed to be reserved for that object on all other nodes. This general structure is pictured in Figure 4.

The distributed address space allows direct addresses to be used as unique location-independent object references, greatly simplifying the implementation of key features such as remote invocation and object mobility. The memory organization is closely related to issues of consistency of the data shared by multiple nodes. At the hardware level each node can address only its private physical memory, and coherence of these private memories is difficult to maintain efficiently in a distributed environment. Since remote references are trapped at object invocation time, objects in effect define the granularity of memory coherence. We believe that the object is a natural and efficient unit for maintaining coherence of the global address space.

4.1 Implementation of The Distributed Address Space

The global address space is implemented by arranging the virtual address space of each participating Topaz task identically. Code and statically initialized program data are handled automatically because the tasks are activations of exactly the same program image read from the distributed file system and executed on homogeneous machines. The difficulty is in forcing nodes to use disjoint regions of the address space for heap allocations of dynamic objects. This section describes how the use of the shared address space is negotiated and how remote references are trapped and handled efficiently.

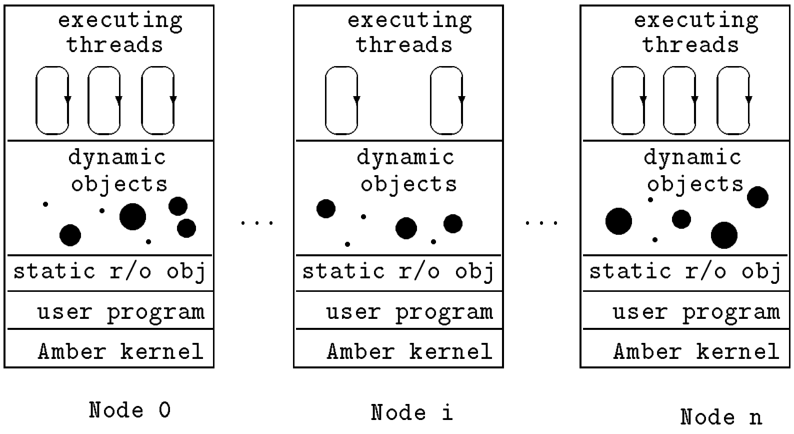


Figure 4: Amber Memory Organization

4.1.1 Address Space Partitioning

Most Amber objects are dynamically allocated from a heap. Because all nodes share a single virtual memory, nodes must coordinate the use of address space so that two nodes do not allocate the same heap block. However, the system would have serious performance problems if each object allocation required distributed agreement. Instead, the address space is apportioned by regions large enough to handle many object allocations. Each node is assigned an initial region (on the order of 1M bytes) at startup time for its local heap allocations. Statically partitioning the entire virtual address space in this way would be a poor idea because objects are not allocated uniformly across nodes. For this reason, a large part of the virtual address space is left unallocated at startup and is handed out later as nodes exhaust their initial pool. These later region allocations can be coordinated by a centralized (or possibly distributed) address space server, as in Ivy. The cost of extending the address space is not excessive because the regions are large enough that extensions are needed relatively rarely.

Amber's distributed virtual memory system guarantees that no region of the dynamic address space is ever owned by more than one node and that any assigned region is mapped into the address spaces of all the nodes. One shortcoming of this approach is that it requires the sparse use of a large address space, which is somewhat expensive in terms of page table overhead on systems like the VAX. Also, it may be prohibitive for huge numbers of nodes with small address spaces per node. For now, we are limiting ourselves to relatively small networks (e.g., tens of nodes). Microprocessors with 48- or 64-bit address spaces and inverted page tables are likely to become common in the next several years, which will eliminate this concern [Chang & Mergen 88, Mahon et al. 86].

4.1.2 Handling Non-local References

Although Amber objects are referenced by virtual address, location-independence requires the system to determine whether or not an object is local when it is invoked. To provide this information, each object has an *object descriptor* on every node that indicates whether or not the described object is locally resident. The descriptor may contain other information about the object, such as where to look for it if it is remote. An object's local descriptor is examined on each potentially remote invocation by checks inserted by the preprocessor. If the descriptor indicates that the object is remote, the invocation traps to the kernel and is handled by a remote procedure call. This scheme adds only marginally to the cost of local invocations because a descriptor check can be done in a single VAX instruction.

Each Amber object is implemented as a record, the first part of which is its descriptor, and the remainder of which is its representation (the data local to the object). The virtual address of an object is therefore the address of its descriptor. When a new object is created it is allocated from the heap on a particular node. The descriptor for the object is initialized on that node to indicate that the object is resident so that it can be invoked. If a mutable object is ever moved, its descriptor is changed to indicate that it is not resident, and a forwarding address is inserted in the descriptor. Any attempt to invoke the object will then trap to the kernel.

Objects and their descriptors are managed so that an uninitialized descriptor is detected and interpreted to mean that the object is remote. Suppose that a reference to a newly-created object A is passed to a node N which A has never visited, and that some thread on N attempts to use the reference to invoke A. In this case, the invoking code on N will use the reference to A to check A's descriptor in order to see if it is resident on N. That check will access a portion of N's address space that is uninitialized and may not even be mapped. If the address is not mapped then Topaz notifies the Amber kernel of the address fault and the kernel responds by mapping the region containing the address. An uninitialized descriptor is handled correctly because all uninitialized pages of virtual memory are zero-filled by the Topaz operating system, and object descriptors are defined so that the resident flag is a zero-valued bit. Therefore, the check for residency will see a zero bit and will trap to the Amber kernel to perform a remote invocation. The system also properly handles references to objects occupying heap blocks which were previously deallocated and reused, but this mechanism is not described here.

When the kernel handles a trap on an invocation of a remote object it retrieves a forwarding address for the object from the object's descriptor. (The use of forwarding addresses is described in [Fowler 85].) The forwarding address may be out of date if the object moves frequently. In this case the object's location can be determined by following a chain of forwarding addresses, since the object leaves a new forwarding address on each node through which it passes. It is costly to locate an object by following a forwarding chain, but this happens rarely because information about the object's latest location is cached on all nodes along the chain so that subsequent references to the object can be handled quickly.

A slightly different approach is needed in the case of a trap on an object with an uninitialized descriptor. A reference to the node which owns a region is obtained from the address space server and cached when the region is first mapped by a task, so each task has complete knowledge of the assignment of heap regions to nodes. This allows the system to identify the node on which a heap object was created from the object's virtual address. When a reference to an object with an uninitialized descriptor is trapped, the kernel contacts the node on which the object was created. The home node of an object can determine where the object resides by following the chain of forwarding addresses.

4.2 Comparison with Previous Systems

The issues surrounding the use of a distributed address space can be illustrated by comparing the Amber scheme with approaches used by other systems. From the programmer's perspective the Amber model is quite similar to the shared object space of the Emerald system. The implications of this model for application design and performance are best understood by comparison to the more general shared virtual memory of the Ivy system. This section explores the relationships between the three approaches to data sharing. One important difference is that the shared object space of Emerald and Amber presents a more restrictive programming model than the Ivy approach.

Another difference is that non-local references in the shared object model are handled by moving the control stream to the location of the data rather than moving the data to the location of the faulted reference. This contrasts with the Ivy approach to distributed coherence, which moves the referenced data to the node on which it is referenced using ownership schemes analogous to hardware cache consistency protocols. The third difference is seen in the units used for maintaining memory coherence. In Amber the granularity of coherence is the data object, a problem-oriented unit which has a number of advantages over the page-level coherence used by Ivy.

4.2.1 The Ivy System

Ivy was one of the first systems to use a network, originally of uniprocessor workstations, to construct a loosely-coupled multiprocessor. Ivy also pioneered the use of a network-wide virtual memory to simplify communication through the use of shared-memory semantics. Coherence of Ivy's shared memory is maintained by memory managers on each node, which use page faults to detect shared accesses and exchange coherency messages with the other memory managers [Li & Hudak 86]. This shared virtual memory system eliminates the need for programmers to explicitly manage binding and remote communication, as they would with a conventional remote procedure call system [Birrell & Nelson 84]. From the programmer's perspective data distribution in Ivy is completely transparent because memory is guaranteed to be consistent at the byte level across all the nodes.

A major difference between Amber and Ivy is that Amber takes a function-shipping rather than a data-shipping approach to coherence. Rather than attempting to maintain the consistency of mutable objects across nodes, each object is placed on a single node where its access is controlled through its operations. Threads that address a remote object are faulted and moved to the referenced object's location, where the consistency of the object's internal state can be efficiently managed by hardware means. For this reason Amber programmers do not use process migration primitives as they do in Ivy to explicitly specify which node a given control stream is to execute on. The location of each computation is determined transparently by the system from the locations of objects. In contrast, Ivy manages the location of data transparently but requires the programmer to explicitly control the location of computation. One assumption of Amber is that the performance effects of the function-shipping approach are more predictable than the data-shipping approach. Ivy can thrash when a memory page is repeatedly referenced by difference nodes. Similarly, Amber can thrash when a thread repeatedly invokes the same remote object, but this effect is less dependent upon the orders of events and the timings of concurrent operations.

Ivy's unit of distribution and coherence is the memory page. The performance of page-level coherence is dependent upon the degree of locality in the memory references made by the program. In contrast, Amber maintains consistency at the object level. An underlying hypothesis of Amber is that the references to a given object in an object-oriented program exhibit locality characteristics naturally suited to maintaining consistency. The body of an object operation can reference only the thread stack and the contents of the object itself, so an executing member function is likely to make a sequence of highly localized memory references. Another advantage is that knowledge implicit in the way the data area is divided into objects can be exploited to make the coherence algorithm more efficient.

Since maintaining coherence involves network traffic, programmers must be aware of the coherence scheme in order to reduce artificial sharing and avoid unnecessary network traffic. Programs need to control locality in NUMA architectures, and the page is a somewhat artificial unit of locality, unrelated to program structure. Ivy may incur unnecessary coherence overhead when logically unrelated data items that happen to reside in the same page are referenced repeatedly by multiple nodes. In a scheme with page-level coherency, programmers will need to become aware of page sizes and boundaries in order to achieve the best performance, just as programmers of current multiprocessors need to be aware of cache line sizes to reduce artificial sharing.

This tradeoff between pages and objects will depend somewhat on the page size and the average object size. If objects are huge, migrating an object may waste latency and network bandwidth if only part of the object will ever be accessed. On the other hand, if objects are small, then moving objects and remotely invoking them may be cheaper than sending coherency messages. Performance of the Amber approach is also affected by the number and duration of remote object invocations: many short remote invocations will reduce the performance of an Amber application.

4.2.2 Emerald

The global data space of Emerald and Amber can be thought of as a shared object space in which coherence is maintained at the object level by trapping invocations of remote objects. This contrasts with Ivy, where consistency of arbitrary bytes is guaranteed across multiple nodes, allowing a wide range of conventional programming techniques to be used transparently by a distributed application. Emerald's distributed object address space operates in a significantly different fashion from Amber's. In Emerald, each object has a unique global identifier that is distinct from its virtual address. This global identifier is used to reference an object remotely. However, each node manages its own private virtual address space; within a node, an object is referenced by the local virtual address of its object descriptor. In Amber, nodes cooperate in the management of their address spaces, and direct addresses are used to reference objects.

When an Emerald object A moves from node N to node M, its virtual address will change. Other objects on N that reference A do not need to change because A's descriptor will remain on N and will be modified to indicate that A has moved. References within A, however, must be modified because they contain virtual addresses of other object descriptors on N. Those descriptors will exist at different virtual addresses on M. Emerald uses special compiler support to aid in the translation of stored object references.

Emerald implements node-local virtual address spaces and global identifiers because its user model assumes a universe of long-lived objects created by multiple users. The multiple private address spaces provide a larger virtual address space and permit each node to do independent address space management. By contrast, Amber assumes a single application program with a relatively short lifetime and moderate total memory requirements. Another motivation for the use of a distributed address space in Amber is the difficulty of translating object references in a C++ environment, since C++ lacks the rigidly enforced typing constraints of Emerald.

5 Other Implementation Issues

5.1 Implementing Object Mobility on a Multiprocessor

Conceptually, moving an Amber object is a simple operation because of the global virtual address space. No translations of addresses are required either inside or outside of the moving object. No code is ever moved because all code is replicated on every node at initialization time. Moving an object involves modifying the object's descriptor on the source node to indicate that it is no longer resident, copying the object's data bytes to the same memory region on the destination node, and modifying the object's descriptor on the destination node to indicate that the object is local. The system must also identify, stop, and move all threads which are executing member functions of the moving object. When a thread is executing a member function of some object it is said to be *bound* to that object. It is easy to determine which object a thread is bound to by examining the top frame of the thread's runtime stack.

From an implementation standpoint, including object mobility in a system with real concurrency creates numerous difficult interactions between threads and objects. The coherence scheme requires that all references to the internals of a moving object be made on the node where the object resides. If the object is in transit, no references to its internals can be made until it arrives at its destination. Threads must be prevented from invoking or returning into a moving object on the source node after a decision has been made to move it. Any threads already executing within a moving object must be identified and moved with the object. The central problem is that user threads may be running concurrently with the mobility code on another processor and attempting to manipulate the moving object. This makes it difficult to eliminate race conditions that would cause a thread to execute on the wrong node during or after a move.

The Emerald implementation of mobility is much simpler because Emerald is designed for uniprocessor systems. When the Emerald kernel moves an object, all processes on the node are implicitly suspended while mobility code in the kernel is in control of the single processor. It is a simple matter to determine which processes are bound to the moving object by examining their stacks. In addition, Emerald is coded so that the invocation sequence itself is atomic; no race condition can exist between checking an object for residency and building the frame to indicate that an invocation is active.

The Emerald scheme is inadequate for multiprocessor hardware because one thread may invoke an object just as another thread on a different processor is trying to move it. The first major problem is that the system must guarantee the atomicity of descriptor checks made on invocations and returns. In a naive implementation, a thread might check the descriptor and find that the object is local but not actually complete the local invocation until after the object has moved, and the system might fail to recognize that the thread is bound to the object and must move with it. This race condition will always exist if the descriptor is checked before the stack modifications associated with the invocation are made. An analogous race condition can occur on returns: a thread could check that an object is still resident before its return, but the object could move after that check and before the actual control transfer.

The solution used in Amber is to modify the invocation and return sequences. The invocation descriptor check is made at the beginning of each operation, after the invocation stack frame is pushed. Likewise, the return check is made after the invocation frame that the thread is returning from has been popped. This is sufficient to ensure that the executing thread will always be identifiable as executing within the object before it checks the descriptor. No new threads can enter or return into the object once the descriptor is marked as non-local.

A related problem is determining which active threads are already bound to a moving object and must move with it. In general there are two approaches to a solution. One approach is to keep a data structure describing those threads currently executing within each object. This solution simplifies mobility but it makes invocations expensive because the structure must be updated atomically on each invocation and return. The other approach is to freeze all activity on the node and examine the stacks of all suspended threads to determine which are bound to the moving object.

The Amber solution is to preempt all running threads and force them to be rescheduled after the descriptor of a moving object has been marked as non-resident. An additional check for residency of the current object is made on each context switch to a preempted thread. The thread is immediately moved to the appropriate node if this check indicates that the object has moved.

This scheme adds marginally to the cost of context switches, and it makes the *MoveTo* primitive rather costly since it is expensive to preempt a running thread. It has the added disadvantage that the cost of mobility increases as processors are added to a node. Also, some concurrency may be lost if the destination node is idle but the source node is busy, since suspended threads which are bound to the object will not move to the destination node until they are rescheduled on the source node. We believe that these costs are acceptable because we expect object moves to be much less frequent than object invocations. Local invocations and returns are very fast in Amber because the descriptor checks inserted by the preprocessor can be done with a single VAX branch-on-bit-clear instruction. In addition, we expect that objects will often be moved at "quiet" synchronization points in Amber programs; as an optimization, an object can be moved more efficiently if the program can assert to the move operation that no threads are active in the moving object.

5.2 Using C++ for Distributed Systems

The major advantage of C++ is that it is flexible enough to integrate Amber primitives into the programmer's world without modifying the language. Amber's facilities and interfaces are defined cleanly by a hierarchy of C++ classes made visible to the program by definition files that it imports. Object descriptors are allocated and managed by deriving all user classes from a single base class called *Object* whose private data items include the object descriptor. The constructor and destructor functions for the *Object* class initialize the descriptor and ensure that object deallocation is performed on the correct node. The mobility primitives are supplied as operations (*member functions* in the C++ parlance) on members of class *Object*. Other facilities such as threads, immutable objects and synchronization objects are provided by introducing new subtypes of *Object* using the C++ derived class mechanism. Derived classes and C++ virtual functions are also a convenient vehicle for supporting customizability of the system, allowing programmers to tailor system behavior to the needs of their specific application. For example, a customized scheduling discipline can be installed at runtime by replacing the system run queue object with a similar object that behaves slightly differently. In general, user subtyping of system classes is a clean mechanism for structuring upcalls from the kernel to user code.

The major problem with using C++ is that Amber's distribution model is dependent upon the regularity of an object-oriented programming language. The system assumes that remote objects will only interact through invocation, since references to remote objects are recognized and trapped only on invocations. In an Amber program, threads executing within an object cannot directly manipulate the internals of some other object with safety since that object might be remote. Furthermore, all data items which may be referenced remotely must be encapsulated in an object. The C++ language supports object-based programming, but it cannot be regarded as an object-oriented language because it includes many language features which violate the precepts of the object paradigm. Examples of such features are friends, public member elements and unprotected structures. Other features of C++, such as static class member elements and inline member functions, provide a syntactic illusion of object-orientedness but violate deeper constraints that Amber is dependent upon for correctness. As a rule, these features were included in the language for performance reasons. Any of these features can result in incorrect behavior if they are used to implement interactions between objects which do not reside on the same node. Another problem with C++ is that it allows the programmer to violate typing constraints or lapse into C code, which is decidedly not object-oriented.

These dangerous features of C++ present opportunities for the programmer to optimize interactions between co-located objects. In some situations the programmer can depend on objects being co-located. Co-location can be explicitly requested using Amber's attachment primitives. Also, C++ allows an object to be directly contained within some other object. These *member objects* always move with their containing object and are guaranteed to be co-resident with it. Intelligent use of the performance features of C++ in situations where co-residency is assured can significantly improve program performance. For example, consider a multithreaded object whose internal state is protected by a non-relinquishing lock. If the lock is a member object of the protected object then it can be safely acquired and released using fast inline function calls. Co-residency guarantees can also be exploited to optimize invocations of functions in base classes or invocations of objects allocated from a thread's stack.

It might be questioned whether the presence of efficient but dangerous C++ features in Amber is an overall liability or an asset. Certainly these features can introduce subtle bugs if used improperly. However, they can also substantially improve program performance when used correctly. There is a strong suggestion that the class of programmers served by Amber prefers performance and flexibility over security. For this reason we have been reluctant to modify the Amber preprocessor to restrict the use of questionable language features.

6 Performance

To explore the performance of Amber, we return to the implementation of Red/Black SOR discussed in Section 3 and diagrammed in Figure 3. Recall that the implementation divides the rectangular grid of points into some number of sections, each of which is a multi-threaded Amber object.

For the purposes of our experiments, we selected a specific problem with a grid size of 122 by 842 points. As a baseline for comparison, we treated the entire grid as a single section and solved it with a single thread. This partitioning has virtually no overhead, and represents the "non-Amber" C++ sequential execution time. All parallel speedups are reported relative to this implementation.

The parallel implementations all involved a partitioning of the grid into 8 section objects. In various experiments, these 8 section objects were distributed among 1 to 8 DEC SRC Firefly multiprocessor workstations, using 1 to 4 processors per workstation.[1] The results of these experiments are shown in Figure 5. Each point in this figure represents the measured speedup for a particular experiment, relative to the sequential C++ implementation described above. Each point is labeled to indicate the number of Firefly nodes employed, and the number of processors per node. For example, the point labeled "4Nx2P" corresponds to an experiment in which the 8 sections of the grid were distributed among 4 Fireflys (2 per Firefly) and two processors per Firefly were used (for a total of 8 processors). A number of conclusions can be drawn from Figure 5:

- The overhead inherent in using the Amber object model is acceptably low. This is demonstrated by the fact that the speedup of the Amber implementation is very close to the ideal speedup relative to the sequential implementation, for 8 or fewer processors. (Recall that all Amber results are based on an 8-section-object partitioning of the grid.)

- The overhead of *remote* Amber object invocation also is acceptably low. This is demonstrated by the fact that identical speedups are achieved for all the experiments that involve a total of 4 processors: 1 node with 4 processors, 2 nodes with 2 processors each, and 4 nodes with 1 processor each. Identical speedups also are achieved for two 8-processor experiments: 2 nodes with 4 processors each, and 4 nodes with 2 processors each.

- The fundamental goal of Amber – to allow the power of a network of small-scale multiprocessors to be harnessed for a single parallel application – has been achieved. This is demonstrated by the fact that a speedup of 25 is attained for the 8Nx4P case – 8 Firefly workstations, each contributing 4 processors to the overall solution.

[1]In truth, 2 of the 11 experiments – those using 3 and 6 Firefly nodes – used a partitioning into 6 section objects rather than 8.

Figure 5: Measured Performance of Amber Red/Black SOR Implementation

- Significant performance benefit comes from structuring the implementation so that transfers of edge data are overlapped with computation over the interiors of sections. This is demonstrated by the significantly different performance of the two 8Nx4P cases. Note that a page-oriented distributed virtual memory utilizing demand paging would not benefit from this overlap.

We have not implemented this application under a system with a page-oriented distributed virtual memory, so it is impossible to make exact comparisons. Certainly the page-oriented model would have required less coding effort initially. However, the performance achieved by programs such as this ultimately depends on how efficiently data can be transferred between processors. One must pay close attention to the partitioning of data reference patterns to obtain good performance. The methods for doing this on Amber, with its object-oriented distributed virtual memory, and using an Ivy-like system with a page-oriented distributed virtual memory, are quite different.

Using a page-oriented system, the programmer would optimize data reference patterns by laying out data structures and partitioning the work so as to make each processor reference different sections of the linear address space. If two processors both start writing to the same block of addresses, the virtual memory system will thrash. The fact that this can happen may not be obvious at the source code level. For example, any naive partitioning of the SOR program (Figure 2) will thrash miserably because of repeated references to the shared variable `maxdif`.

Using Amber, the programmer optimizes data reference patterns by organizing the object definitions and locations so as to minimize the number of remote invocations. This has both advantages and disadvantages. On the minus side, the process can be tedious for some applications. On the plus side, the resulting performance is more likely to be predictable and understandable. We believe that the explicitness with which decomposition must be addressed in Amber is a net asset because the programmer has complete control over which data is transferred and when.

7 Summary

This paper described the Amber system, which permits a single application program to view a network of multiprocessors as an integrated, non-uniform memory access, shared-memory multiprocessor. We believe that the underlying hardware architecture makes sense from a cost/performance point of view. Processors can be added to a computer system at small marginal cost, but packaging constraints limit the practical size of a single system. Therefore programmers will need to build parallel programs that cross machine boundaries.

The object-oriented programming model of Amber strikes a balance between the ease of programming afforded by a page-oriented distributed virtual memory, and the performance gains that can be achieved through explicit management of location. We believe that, at least for the near-term, programmers will need to explicitly control locality of data and processing in order to maximize speedup. Amber provides programmers with high-level abstractions for program structuring, while at the same time giving them control over locality.

We have presented a specific Amber application – an implementation of Red/Black SOR – to illustrate the Amber program design process and to demonstrate the performance achieved by the system. The results demonstrate that the fundamental goal of Amber – to allow the power of a network of small-scale multiprocessors to be harnessed for a single parallel application – has been achieved.

8 Acknowledgements

We would like to thank Brian Bershad and Jan Sanislo for helping us to understand the operating system and hardware of the Firefly. We would also like to thank the DEC Systems Research Center for providing the Firefly workstations, without which this work could not have been done.

References

[Allchin & McKendry 83] J. Allchin and M. McKendry. Synchronization and recovery of actions. In *Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing*, pages 31–44, August 1983.

[Almes et al. 85] G. T. Almes, A. P. Black, E. D. Lazowska, and J. D. Noe. The Eden system: A technical review. *IEEE Transactions on Software Engineering*, SE-11(1):43–59, January 1985.

[Bershad et al. 88a] B. N. Bershad, E. D. Lazowska, and H. M. Levy. PRESTO: A system for object-oriented parallel programming. *Software – Practice and Experience*, 18(8), August 1988.

[Bershad et al. 88b] B. N. Bershad, E. D. Lazowska, H. M. Levy, and D. Wagner. An open environment for building parallel programming systems. In *Proceedings of the ACM SIGPLAN Symposium on Parallel Programming Environments, Applications, and Languages*, July 1988.

[Birrell & Nelson 84] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[Chang & Mergen 88] A. Chang and M. F. Mergen. 801 storage: Architecture and programming. *ACM Transactions on Computer Systems*, 6(1):28–50, February 1988.

[Downey 88] W. J. Downey. Inside the GP1000. Technical report, BBN Advanced Computers, Inc., Cambridge, Massachusetts, May 1988.

[Fielland & Rodgers 84] G. Fielland and D. Rodgers. 32-bit computer system shares load equally among up to 12 processors. *Electronic Design*, pages 153–168, September 1984.

[Fowler 85] R. J. Fowler. *Decentralized Object Finding Using Forwarding Addresses.* PhD dissertation, University of Washington, December 1985. Department of Computer Science technical report 85-12-1.

[Gehringer et al. 87] E. F. Gehringer, D. P. Siewiorek, and Z. Segall. *Parallel Processing: The Cm* Experience.* Digital Press, Bedford, Massachusetts, 1987.

[Jones et al. 79] A. K. Jones, R. J. Chansler, I. Durham, K. Schwans, and S. R. Vegdahl. StarOS, a multiprocessor operating systems for the support of task forces. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles*, pages 117–127, December 1979.

[Jul 88] E. Jul. *Object Mobilty in a Distributed Object-Oriented System.* PhD dissertation, University of Washington, December 1988. Department of Computer Science, TR-88-12-06.

[Jul et al. 88] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[Lampson 84] B. W. Lampson. Hints for computer system design. *IEEE Software*, 1(1):11–28, January 1984.

[Levin et al. 75] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in hydra. In *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*, pages 132–140, November 1975.

[Li & Hudak 86] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. In *Proceedings of the 5th ACM Symposium on Principles of Distributed Computing*, pages 229–239, August 1986.

[Li 86] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD dissertation, Yale University, September 1986. YALEU/DCS/RR–492.

[Liskov 88] B. Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[Mahon et al. 86] M. J. Mahon, R. B.-L. Lee, T. C. Miller, J. C. Huck, and W. R. Bryg. Hewlett-Packard precision architecture: The processor. *Hewlett-Packard Journal*, 37(8), August 1986.

[Ousterhout et al. 80] J. K. Ousterhout, D. A. Scelza, and P. S. Sindhu. Medusa: An experiment in distributed operating system structure. *Communications of the ACM*, 23(2):92–105, February 1980.

[Stroustrup 86] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Massachusetts, 1986.

[Thacker et al. 88] C. P. Thacker, L. C. Stewart, and E. H. Satterthwaite, Jr. Firefly: A multiprocessor workstation. *IEEE Transactions on Computers*, 37(8):909–920, August 1988.