SHIVA:
AN OPERATING SYSTEM TRANSFORMING A HYPERCUBE INTO A SHARED-MEMORY MACHINE

Kai Li
Richard Schaefer

CS-TR-217-89

April 1989

Shiva: An Operating System Transforming A Hypercube into a Shared-Memory Machine*

Kai Li Richard Schaefer

CS-TR-217-89

April 10, 1989

Abstract

The Shiva project at Princeton aims to develop an operating system supporting both the shared-memory and message-passing models of parallel computation for the second-generation message-passing multicomputers. Our initial system was designed and implemented for the Intel iPSC/2 hypercube multicomputer. It provides a large, coherent, shared virtual memory and a multithread interface—transforming a hypercube multicomputer into a shared-memory multiprocessor. The Shiva system also supports message-passing among threads at low cost. Our preliminary performance measurements indicate that shared virtual memory is an effective strategy for implementing the next generation operating systems for hypercube multicomputers.

*This research was supported in part by the National Science Foundation under grants CCR-8814265 and by the Intel Corporation.

1 Introduction

High-performance message-passing multicomputers with hundreds of processors have become commercially available over the past few years. There are more than one hundred first-generation multicomputers in use and during the past two years second-generation multicomputers with faster processors and much faster message-passing networks have appeared[Sei85, AS88]. Although these machines exhibit performance comparable to that of conventional supercomputers on many computing problems, their operating systems have not been able to take full advantage of their massive physical resources to effectively support a large domain of applications.

The existing systems for the multicomputers have a number of drawbacks:

- *Lack of support for the shared-memory model of parallel programming.*

  We are convinced that no single model of parallelism is appropriate for all applications. Solely providing the message-passing model restricts the domain of applications. Whether a large-scale multicomputer can provide a true shared-memory efficiently and effectively is an open problem.

- *Small memory size of each node.*

  The limit of the memory size on each node is determined by the cost and convenience of packaging. The second-generation multicomputers typically have 1 to 16 megabytes of memory per node. Although such a machine may have gigabytes of aggregate memory, applications whose processes require data spaces larger than a node's physical memory are difficult to run. The small amount of memory on each node further restricts the application domain of multicomputers.

- *Difficulty of passing complex data structures between nodes.*

  Message-passing systems restrict the programmer to passing data by value. There is no way to share pointers since processors have distinct address spaces. Passing complex data structures containing pointers requires copying the entire structure. Also both the sender and receiver processes must keep track of all elements that have ever been sent and received because pointers in the current data structure may point to data structures previously passed[HL82]. Furthermore, if multiple read-only copies of shared data are used to promote concurrency, a coherence strategy must be employed to ensure each node accesses valid data.

- *Large cost of process migration.*

  When migrating a process, all resources allocated to the process have to be moved together. This is expensive [PM83]. The inconvenience of process migration reduces the flexibility of parallel applications.

These drawbacks have seriously limited the application domain of multicomputers. The goal in the Shiva project is to develop an operating system that effectively utilizes the abundant physical resources and overcomes these drawbacks.

The most distinctive feature of Shiva is the implementation of a large, coherent, shared virtual memory address space spanning almost the entire physical memory of the multicomputer. Unlike facilities presented in parallel langauge research, the shared virtual memory supports the shared-memory model at the lowest system level: instructions and memory references. The shared virtual memory not only provides the same interface as that of a shared-memory multiprocessor, but also offers paging between physical memories to support applications requiring massive amounts of data.

Another feature of Shiva is a multithread interface. As in the thread interface in operating systems for shared-memory multiprocessors [MS87], a Shiva threads can reside on any node and coherently access any memory location in the shared virtual memory address space at any time. Unlike threads in existing operating systems, Shiva threads can send messages to each other. We simply view message-passing between threads as memory copying within the shared virtual memory address space with thread synchronization.

Shiva also provides a langauge-independent remote-procedure-call (RPC) mechanism. RPC allows a procedure to be executed on a specified node. The arguments of the procedure can be passed by either value or reference. The remote procedure can also access global variables and outer-scope variables.

This paper describes our design and implementation of a prototype version of Shiva for the Intel iPSC/2 hypercube multicomputer. Our first version, designed for testing system algorithms, has been implemented on top of the NX/2 operating system [Pie88]. The preliminary performance results indicate that the overhead of coherent mapping in the Shiva system is marginal and our future kernel implementation should effectively support the message-passing model as well as the shared-memory model of parallel computation. We expect Shiva to significantly enhance the hypercube multicomputer architecture for a wider range of application problems.

2 The Shiva System

The current Shiva system is designed for the Intel iPSC/2 hypercube multicomputer and runs on each node of the machine. Since the iPSC/2 hypercube is primarily used by single users for large computing problems, its operating system does not need to address protection problems among users. Such a feature greatly simplies the design of Shiva. The main components of Shiva are a shared virtual memory mapping mechanism, a thread control and synchronization module, a message-passing implementation, a language-independent RPC facility, and memory management.

Intel iPSC/2 Hypercube

An Intel iPSC/2 hypercube system consists of an iPSC/2 hypercube multicomputer and a system resource manager (a 80386-based workstation) which is directly connected to one node of the hypercube. As the name indicates, the iPSC/2 is a hypercube connected multicomputer [Arl88], the successor to the Intel iPSC Hypercube. The largest configuration is 128 nodes. Each node consists of an Intel 80386 paired with 64Kbytes of high speed instruction and data cache for 0-wait state operation, a numeric co-processor (an Intel 80387, SX scalar extension, or a VX vector unit), and up to 16 Mbytes of memory [Clo88].

Each node of the iPSC/2 also has a routing logic module that implements a variation of wormhole routing[Nug88] in which transmission is performed after the route has been constructed. Messages are handled independently by the routing units without interrupting processors on the route. The latency of message passing has been significantly reduced to tens of microseconds. With such high performance, the current operating system, NX/2, is able to deliver 350 microsecond latency for short messages between any two nodes in the machine.

The Intel 80386 processor has a Memory Management Unit (MMU) on chip. The page table entry used by the MMU has protection bits, a reference bit, and an invalid bit. The page size of the MMU is 4,096 bytes, which is large enough to have a small page table and to amortize the page fault overhead. These features of the 80386 and the high-performance message transmission allow us to implement shared virtual memory conveniently and efficiently.

SVM Mapping

Shiva provides a single, flat, shared virtual memory address space. The size of the address space is the minimum of 1 gigabyte and the sum of all physical memories of the hypercube minus the space used by the Shiva kernel. A memory mapping manager on each node maps a virtual address space to the shared virtual memory address space, as shown in Figure 1. The address space is kept coherent at all times. That is, the value returned by a read operation is always the same as the value written by the most recent write operation to that particular address. To client programs, the shared virtual memory address space appears the same as a shared-memory space on a shared-memory multiprocessor.

The mapping managers implement the coherent mapping by using the existing MMU hardware on the 80386 processor chip. The mapped address space on each node has a corresponding page table containing one entry for each page. For each virtual address, the address translation unit of the MMU indexes the page table and uses the corresponding page entry to find the physical address and current type of access allowed to the page (nil, read-only or writable). When access right violations occur, the protection mechanism of the MMU triggers page faults and traps the faults in appropriate fault handlers. Our memory mapping manager manipulates the access bits in the page table entries of all pages on the hypercube to keep the shared virtual memory address space coherent. Hence, coherence of the shared virtual memory address space

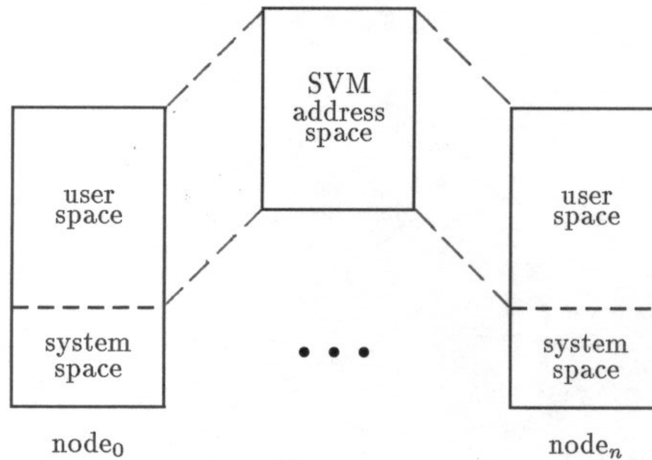

Figure 1: Shared virtual memory mapping.

is solved at the page level with page fault handlers and their servers. To client programs, this mechanism is completely transparent.

Our page coherence strategies are based on invalidation techniques [Li86]. Since the shared virtual memory address space is organized in pages, the memory mapping manager on each node views its local memory as a large cache of pages for its associated processor. Pages that are marked read-only can have copies residing in the physical memories of many processors at the same time. A page currently being written can reside in the physical memory of only one processor. When a processor writes a page that is currently residing on other processors, it must get an up-to-date copy and then invalidate all copies on other processors. A memory reference causes a page fault when the page containing the memory location is not in a processor's current physical memory. When the fault occurs, the memory mapping manager retrieves the page from the memory of another processor.

In general, when a request is made for a page, three roles are involved in a coherence strategy: the requester, the owner and the manager. The requester is the node whose memory reference violated the access rights specified in its corresponding page table entry causing a page fault. The owner of the page is the current or last node to have write access to the page. The manager node of the page maintains the coherence information regarding the page including the current owner, access type, copyset (for read access) and a list of requesting threads. We compact most data structures in the page table to minimize the space required for the system.

Our prototype uses a fixed distributed manager strategy which statically assigns pages to managers by a function of the number of nodes and pages. Upon receiving a page fault, the requester uses this function to identify the manager of the faulting page and sends a request to that node. The manager refers to the ownership information and asks the owner of the page to deliver the page or a copy to the requester. The requester receives the page from the owner

and notifies the manager if necessary. The manager updates the information regarding the page by adding the requester to the copyset of the page, for a read fault, or recording it as the new owner for a write fault. The manager locks the ownership information during the entire protocol to ensure that the messages servicing successive requests for a page do not overlap and corrupt the coherence information.

Of course, if any of the roles involved are performed by the same node, then the protocol can be simplified to minimize the number of messages. In the case where a page fault occurs on the managing node of the page, the requester (and manager) asks the owner for the page directly. When the manager and owner are the same node, the requester again asks the owner directly for the page.

The request bit in the page table entry provides a flag which prevents other unnecessary messages. When a thread has a page fault, the request bit is checked to determine whether the node has already sent a request, yet to be fulfilled, for this page. If so, no message is sent. The bit is cleared upon receipt of the page. Another optimization can reduce the message traffic if a distinction is made between local and remote threads' requests for a page. Upon receiving a page, the manager first allows all requesting local threads to access the page before responding to any remote requests. This strategy avoids extra messages and helps prevent thrashing.

The dynamic distributed manager algorithm [Li86] can further reduce message traffic for maintaining coherency. This algorithm is being implemented in the next version of our prototype system.

Page Replacement

When a page fault occurs, the fault handler sends out a request to the manager of the page and then prepares to receive a copy of the page from the current owner. If all page frames are in use, a page replacement mechanism will be invoked to select one and save it elsewhere to make space for the new page. At present, Shiva only considers page replacements between physical memories since most iPSC/2 configurations do not have secondary storage yet. This is why the size of the shared virtual memory address space is bounded by the total physical memories minus the system space. It is well understood that there is no ideal solution for page replacement because an ideal algorithm requires data about future memory references which are impossible to predict. Therefore, in Shiva, we have designed a page replacement priority mechanism whose parameters can be adjusted during future experiments.

The Least-Recently-Used (LRU) algorithm has been the most popular page replacement method in traditional virtual memory implementations. For shared virtual memory implementations, LRU is not applicable since a recently referenced page may have nil access due to the memory coherence protocol and it should be replaced before a writable page. Our method of calculating the replacement priority of a page is by its page-type and its last reference time:

$$prio_p = type_p \alpha + t(1 - \alpha)$$

where *prio* is the replacement priority of page p, $type_p$ is the page-type value of page p assigned by the system designers, t is calculated by the current time minus the last reference time, and α is a weight parameter to be adjusted through experimentation.

The Least-Recently-Used (LRU) page replacement policy can be viewed as a special case of the page replacement priority, when α is 0. Such a page replacement priority calculation ensures that LRU is used for each page type while the priorities are preserved among different page types for a range of last reference time.

There are five kinds of page frames: *writable, owned read-only, read-only, nil access*, and *unused*. A writable page is obviously owned by the processor. An owned read-only page is also owned by the processor but it is read-only. A read-only page is not owned by the processor, but the processor knows who owns the page. A nil access page is a memory page invalidated by the memory coherence protocol. An unused page is a free page frame. Table 1 shows a page type priority assignment for the five page types in Shiva. The value of α is to be adjusted through experimentation.

Page	**Type value**
writable	10
owned read-only	20
read-only	30
nil access	max −1
unused	max

Table 1: A page-type priority assignment

With such an assignment, unused pages have the highest priority for page replacement and nil access (or invalidated) pages have the second highest priority. Read-only copy pages have the next highest priority because replacing the page frame of a read-only page needs only a single message to inform the owner of the page. Owned read-only pages have higher priority than writable pages because replacing the page frame of a owned read-only page requires a only transfer of the ownership to a page copy holder. Replacing a writable page frame requires a transfer of both the ownership and the content of the page.

Shiva uses a clock hand and the reference bits in the MMU page table to approximate the value of last reference time for page replacement priority calculation. In order to reduce the latency of a page fault, Shiva uses watermarks on the unused pages. When the number of unused pages is below the low watermark, the system does active page replacements while the CPU is idle. When the number of unused pages is greater or equal to the high watermark, the active page replacement is stopped.

Thread and Synchronization

Similarly to the threads in other systems [MS87], Shiva threads can address any location in the shared virtual memory address space and their execution can be overlapped with page fault servicing to maximize processor utilization. Since threads share the same address space, there is no need to set up page tables or flush caches for a thread context switch; thread context switches can be very fast. Unlike other thread implementations, threads in Shiva support both the shared-memory model and the message-passing model.

Thread scheduling is driven by page faults and the sending and receiving of messages. Each node of the hypercube has its own ready queue. When a thread migrates from one machine to another, its thread control block is deleted from the ready queue of the source node and inserted into the ready queue of the destination node.

Shiva provides binary P/V operations as the basic primitives for thread synchronization. Unlike the thread management in IVY [Li88] in which thread synchronization primitives are implemented based on shared virtual memory, all synchronization primitives for the Shiva system are implemented using simple messages. This design decision is based on the low latency of short message transmission and the large page size (4,096 bytes) on the iPSC/2.

There are two strategies for managing semaphores which achieve global access, the static distributed and dynamic, distributed approaches [Li86]. These approaches are analogous to the page coherence strategies. Semaphores can be statically assigned to the node from whose pool they were initialized or they can migrate from node to node. The former strategy is simpler. When a thread accesses a remote semaphore, it must send a message to perform a P operation and receive one acknowledging that it has possession of the semaphore. To perform a remote V operation, the thread sends another message. The alternative approach migrates a semaphore to the node where the thread resides upon the success of its remote P operation. This mechanism guarantees that the V operation is local. It also takes advantage of locality of reference to semaphores. If several threads on a single node need access to the semaphore at nearly the same time, then the semaphore operations are all local after migration. Our prototype system has employed the dynamic, distributed strategy.

The semaphore structure contains a lock bit, an awaiter bit and a location field. A test and set instruction is used on the lock bit during the P operation. The awaiter bit indicates whether there are threads blocked on the semaphore. The location field contains the number of the node where the semaphore is located. Each node has a data structure for the same semaphore. There is only one holder of the semaphore at any time. The holder maintains a queue of threads waiting on the semaphore. Initially, the lock bit of the holder is cleared and the lock bits on other nodes are set so that any P operations on non-holder node will fail to test-and-set the lock bit. The fast path of P and V need only three or four instructions[2]. When a P operation fails to test-and-set the lock bit, it looks for the semaphore holder, according to the location field, to put itself onto the queue. Just as in the dynamic, distributed manager

[2]This method was due to a discussion with John R. Ellis in July 1988.

algorithm for page coherence [Li86], the location field is a hint indicating the possible semaphore holder.

The V operation checks the awaiter bit. if there are threads waiting when a V is performed, the first thread in the waiting queue is started and the semaphore is migrated to the node where the waiting thread resides without clearing the semaphore. This method preserves fairness and prevents a newly migrated semaphore from being remigrated before the restarted thread gets a chance to lock the semaphore.

Message Passing

Since the message-passing model of parallel computation fits many applications, Shiva allows message passing between threads. The basic primitives provided for message passing are similar to those in the standard C library for the iPSC/2 [Pie88]. Existing application programs can run under Shiva with very few modifications.

There are three simple blocking calls for message passing:

- `csend( type, tid, buf, length )`

  It assigns a type and destination to a message, then sends it. The call does not return until the message has entered the communications network and the send buffer is no longer needed.

- `crecv( typesel, buf, length )`

  It selects an incoming message by type and receives it into a buffer. It does not return until the message arrives in the buffer.

- `cprobe( typesel )`

  It waits for a selected type of message to arrive at the node.

When `cprobe()` returns, `crecv()` can be used to receive the message into a buffer.

There are five non-blocking primitives for sending and receiving messages asynchronously:

- `mid = isend( type, tid, buf, length )`

  It initiates transmission of a message and immediately returns a message id.

- `irecv( typesel, buf, length )`

  It sets up a buffer for receiving an incoming message of the selected type, and returns immediately.

- `iprobe( typesel )`

  It determines whether a message is ready to be received.

- `msgwait( mid )`

  It waits until either an `isend()` or `irecv()` operation on message `mid` completes.

- `msgdone( mid )`

  It returns the status of `isend()` or `irecv()` operation on message `mid` immediately.

Unlike the standard routines for the iPSC/2 which use node and process identifiers to specify a destination, the corresponding primitives in Shiva use thread identifiers instead. This approach frees the programmer from keeping track of the location of threads and allows the use of system provided load balancing mechanisms. Also, the use of thread identifiers offers more flexibility in the implementation of thread migration.

Since all nodes have access to the shared virtual memory address space on the hypercube, sending and receiving messages are viewed as copying from one buffer to another within the shared virtual memory address space and synchronizing the sending and receiving threads. We could implement the message passing primitives with the shared virtual memory coherence mapping. We did not choose this approach because sending a short message would lead to copying an entire, fairly large (4,096 bytes) page of virtual memory. Instead, we implement send and receive using short hypercube messages with a much shorter latency time.

RPC Mechanism

Shiva supports a simple, language-independent RPC mechanism. There is only one primitive:

```
rpc( node, proc, args );
```

where `proc` is the procedure to be executed on node `node` and `args` are the arguments of the procedure. The RPC mechanism has exactly the same semantics as local procedure calls except that the execution is performed on another node.

The primitive is implemented as two thread migrations. The thread first migrates to the remote node, executes the procedure, and then migrates back to the original node. Since the shared virtual memory address space is coherent on all nodes, there is no need to do data structure marshalling or copying for argument passing.

Such a mechanism provides more powerful semantics than the traditional RPC mechanism [Nel81, BN84]. Shiva's RPC can pass arguments by reference in addition to by value. The body of the procedure can reference global and outer scope variables. The traditional RPC mechanism limits the remote procedure to passing arguments by value and the remote procedure cannot access global or outer scope variables since the procedure will be executed in another address space.

The send and receive primitives and the RPC mechanism enable the shared virtual memory system to support the message-passing model of parallel programming in addition to the shared-memory model. Parallel programming languages based on either model or a combination of both can be implemented on this system conveniently and efficiently.

3 Performance

We've performed some preliminary measurements on an 128-node hypercube to determine the cost of various basic operations of the Shiva system. We were primarily interested in the performance of the page coherence and context switching mechanisms and the potential for improvement obtained by moving Shiva facilities into the kernel. Timings were also taken of various primitive operations of the NX/2 operating system which are relevant to the analysis of Shiva facilities. The mclock system call of NX/2 was used to obtain results with 1 millisecond precision. Each test performed 10,000 iterations of the operation being measured to give us microsecond accuracy.

The cost of Shiva's page fault and coherence mechanism was tested by slightly modifying the code. Access to a page received over the network after a fault is only granted on the 10,000th iteration. Hence, the mechanism works exactly as usual except it counts faults and sets the page access to nil rather than read or write until the final iteration. The general and special cases described above were tested for both read and write faults. The special case when a page fault occurs on the manager neighboring the owner represents the best case. The worst case occurs when the three entities are distinct nodes, the faulter and owner are as far apart as possible (7 hops) and the manager is a neighbor of the owner. A full page must travel across the width of the network and two of the three short messages must travel almost that far (6 hops). The cost of context switches was measured by creating two threads that simply take turns suspending themselves. The first thread keeps a count of the number of iterations performed and exits upon completion of the test.

Page fault	Faulting Node	Manager Node	Costs (ms)
Read fault on manager	1	1	3.828
	127	1	4.097
Read fault on nonmanager	1	0	4.035
(manager = owner)	127	0	4.286
Read fault on nonmanager	2	1	4.467
(manager $\neq$ owner)	127	1	4.779
Write fault on manager	1	1	3.828
(manager =	127	1	4.084
Write fault on nonmanager	1	0	4.059
(manager = owner)	127	0	4.296
Write fault on nonmanager	2	1	4.487
(manager $\neq$ owner)	127	1	4.760

Figure 2: Shiva system page fault costs.

Figure 2 shows the page fault costs in our prototype system. The memory coherence mapping uses the fixed distributed algorithm. For each type of page fault, we tested two cases:

manager adjacent to the faulting node and manager far away from the faulting node. In all cases, the worst possible situation on 128 nodes takes less than 4.8 milliseconds to service the fault. The performance of this relatively unoptimized, user mode implementation is considerably better than the raw access time of disks used to support traditional virtual memory.

Two improvements to the current implementation are in progress. The first is to implement a dynamic distributed coherence algorithm. The second is to move Shiva into the kernel. To predict the effect of moving the current implementation into the kernel, we tested the costs of related operation in the kernel. Figure 3 shows the time spent at each stage when there is a read page fault on the manager and the predicted improvement when the coherence mapping is moved into the kernel.

	Current (ms)	Kernel (ms)
Reinstall fault handlers	0.363	0.000
Communication with manager	0.661	0.516
Copy page to send buffer	0.134	0.042
Send and receive the page	2.585	2.440
Copy page from receive buffer	0.201	0.042
Set accesses	0.130	0.080
Other cost	0.023	0.023
Total cost	4.097	3.143

Figure 3: Analysis of a read page fault on manager.

For the current coherence mapping algorithm, (fixed distributed manager), our conservative analysis shows that a kernel implementation can provide at least a 23% improvement. The current prototype runs in user mode requiring the reinstallation of trap handlers after each page fault. A kernel mode would install trap handlers only once at the initialization stage. Although the current prototype implements page copying to buffers by flipping page table entries, it is not done in the most efficient way due to system call constraints. We have been informed that further improvement in the latency of short messages can be gained by disabling the flow control mechanism[3]. This may give us another 5 to 7 % improvement.

4 Related Work

The concept of shared virtual memory was originally proposed in [Li86]. Experience with the first prototype, IVY, on a network of workstations indicates that shared virtual memory has the potential for a large scale multicomputer [Li88]. Another paper [LH86], provides a detailed

[3]Private communication with Paul Pierce in March 1989

analysis of the memory coherence strategies for shared virtual memory. The Shiva system is the first design and implementation for a large-scale multicomputer.

Research on traditional virtual memory management for uniprocessor architecture [Den70, DD68, BBMT72] has had significant impact on the ideas underlying shared virtual memory. An important observation was the *locality* of reference exhibited by sequential programs [Den72].

Spector proposed a *remote reference/remote operation model* [Spe82] in which a master process on one processor performs remote references and a slave process on another performs remote operations. This model allows a loosely coupled multicomputer to behave in a way similar to CM* [FOR*78, JCD*79] or BBN Butterfly [BBN85] in which a shared memory is built from local physical memories in a static manner.

Among the distributed operating systems for loosely coupled multicomputers, Apollo Aegis [Apo81, LLD*83] and Accent and Mach [RR81, FR86, RTY*87] have had strong impact on the integration of virtual memory and interprocess communication. These systems permit mapped access to data objects that can be located anywhere in a distributed system. They view physical memory as a cache of virtual storage. Aegis uses mapped read and write memory as its fundamental communication paradigm. Accent and Mach has a similar facility called *copy-on-write* and a mechanism that allows processes to pass data *by value*.

Another approach has been to have processes use a set of primitives to access a global space containing shared data structures [CS86, CG86]. Process synchronizations are done also with the primitives. Passing complex data structures and process migration are as difficult as in message passing systems because clients cannot pass data structures by address reference.

A remote procedure call (RPC) mechanism [Nel81, BN84] allows synchronous, language-level transfer of control between two programs in disjoint address spaces. Although an RPC mechanism provides syntax and semantics similar to local procedure calls in the application program's high-level language, it requires clients to pass data by value.

The VMP project at Stanford implements a software virtual addressed cache [Che88] to provide multicomputers with a coherent shared memory space. Their initial experience shows that a cache line size can be as large as 128 or 256 bytes without performance degradation. The cache consistency protocol is similar to the dynamic distributed manager algorithm for shared virtual memory.

5 Conclusions

We have described the design and implementation of Shiva for the Intel iPSC/2 hypercube multicomputer. The Shiva system provides clients with a large, coherent, shared virtual memory and a multithread interface. Threads operating within the same shared virtual memory address space can send messages to each other and migrate from one node to another at low cost.

These facilities enable Shiva to support both the shared-memory and message-passing models of parallel programming.

The cost of maintaining memory coherence while servicing page faults in this preliminary version of the system is encouraging. The delay incurred including operating system overhead is nearly an order of magnitude smaller than the typical disk page transfer time. We think remote physical memory can and should be used to support virtual memory on the nodes of a hypercube even when disks are present.

The ability to use the shared memory model is a major attraction of Shiva. This model fits many applications better than message passing and broadens the spectrum of problems which can be solved on the hypercube. Since shared virtual memory systems can easily support the message passing model, algorithms which already run efficiently are not hampered by the system.

Transparency has been the focus of the Shiva system. It provides clients with an address space almost as large as the sum of all physical memories of the entire hypercube. The large address space enables the iPSC/2 to run many applications that require massive data.

We are currently adjusting parameters of page replacement, and implementing a dynamic distributed algorithm for memory coherence. We are doing more experiments on the prototype and in the progress of moving our implementation into the kernel.

Acknowledgements

We would like to thank Mojy Mirashrafi and Paul Pierce for their help with the NX/2 operating system, and Justin Rattner for his support.

References

[Apo81] Apollo. *Apollo DOMAIN Architecture.* Apollo Computer Inc., Chelmsford, Mass., 1981.

[Arl88] Ramune Arlauskas. *iPSC/2 System: A Second Generation Hypercube*, pages 9–13. Intel Corporation, 1988.

[AS88] W.C. Athas and C.L. Seitz. Multicomputers: Message-Passing Concurrent Computers. *IEEE Computer*, 21(8):9–24, August 1988.

[BBMT72] D.G. Bobrow, J.D. Burchfiel, D.L. Murphy, and R.S. Tomlinson. TENEX, a paged time-shaing system for the PDP-10. *Communications of the ACM*, 15(3):135–143, March 1972.

[BBN85] BBN. *Butterfly Parallel Processor Overview.* Bolt Beranek and Newman Adv. Computers Inc., Cambridge, Mass., 1985.

[BN84] A.D. Birrell and B.J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[CG86] N. Carriero and D. Gelernter. The S/Net's Linda Kernel. *ACM Transactions on Computer Systems*, 4(2):110–129, May 1986.

[Che88] David R. Cheriton. The VMP Multiprocessor: Initial Experience, Refinements and Performance Evaluation. In *Proceedings of the 14th Annual Symposium on Computer Architecture*, 1988.

[Clo88] Paul Close. *The iPSC/2 Node Architecture*, pages 43–50. Intel Corporation, 1988.

[CS86] D.R. Cheriton and M. Stumm. The Multi-Satellite Star: Structuring Parallel Computations for A Workstation Cluster. *Journal of Distributed Computing*, 1986.

[DD68] R.C. Daley and J.B. Dennis. Virtual Memory, Processes, and Sharing in MULTICS. *Communications of the ACM*, 11(5):306–312, May 1968.

[Den70] Peter J. Denning. Virtual Memory. *ACM Computing Surveys*, 2(3):153–189, September 1970.

[Den72] Peter J. Denning. On Modeling Program Behavior. In *Proceedings of Spring Joint Computer Conference*, pages 937–944. AFIPS Press, 1972.

[FOR*78] S. Fuller, J. Ousterhout, L. Raskin, P. Rubinfeld, P. Sindhu, and R. Swan. Multi-microprocessors: an overview and working example. *Proceeding of the IEEE*, 66(2):216–228, February 1978.

[FR86] R. Fitzgerald and R.F. Rashid. The Integration of Virtual Memory Management and Inter-process communication in Accent. *ACM Transactions on Computer Systems*, 4(2):147–177, May 1986.

[HL82] M. Herlihy and B. Liskov. A Value Transmission Method for Abstract Data Types. *ACM Transactions on Programming Languages and Systems*, 4(4):527–551, October 1982.

[JCD*79] A.K. Jones, R.J. Chansler, I.E. Durham, K. Schwans, and S. Vegdahl. StarOS, a Multiprocessor Operating System for the Support of Task Forces. In *Proceedings of the Seventh Symposium on Operating Systems Principles*, pages 117–127, 1979.

[LH86] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. In *Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing*, pages 229–239, August 1986.

[Li86] Kai Li. *Shared Virtual Memory on Loosely-coupled Multiprocessors*. PhD thesis, Yale University, October 1986. Tech Report YALEU-RR-492.

[Li88] Kai Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 94–101, August 1988.

[LLD*83] P.J. Leach, P.H. Levine, B.P. Douros, J.A. Hamilton, D.L. Nelson, and B.L. Stumpf. The Architecture of an Integrated Local Network. *IEEE Journal on Selected Areas in Communications*, 1983.

[MS87] P.R. McJones and G.F. Swart. Evolving the UNIX System Interface to Support Multi-threaded Programs. Tech Report 21, DEC Systems Research Center, September 1987.

[Nel81] Bruce J. Nelson. *Remote Procedure Call*. PhD thesis, Carnegie-Mellon University, May 1981.

[Nug88] Steven F. Nugent. *The iPSC/2 Direct-Connect Communications Technology*, pages 59–68. Intel Corporation, 1988.

[Pie88] Paul Pierce. *The NX/2 Operating System*, pages 51–57. Intel Corporation, 1988.

[PM83] M.L. Powell and B.P. Miller. Process Migration in DEMOS/MP. In *Proceedings of the ninth Symposium on Operating Systems Principles*, pages 110–119, 1983.

[RR81] R.F. Rashid and G.G. Robertson. Accent: A Communication Oriented Network Operating System Kernel. In *Proceedings of the Eighth Symposium on Operating Systems Principles*, pages 64–75, December 1981.

[RTY*87] R.F. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. Machine-independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architecture. In *Second International Conference on Architectural Support for Programming Lanugages and Operating Systems*, pages 31–41, October 1987.

[Sei85] Charles L. Seitz. The Cosmic Cube. *Communications of the ACM*, 28(1):22–33, January 1985.

[Spe82] Alfred Z. Spector. Performing Remote Operations Efficiently on a Local Computer Network. *Communications of the ACM*, 25(4):260–273, April 1982.