The Beehive Cluster System*

Aman Singla
Umakishore Ramachandran

College of Computing
Georgia Institute of Technology
Atlanta, Ga 30332-0280

1 Introduction

In this writeup, we present the system architecture of *Beehive*, a cluster system we are developing at Georgia Tech for supporting interactive applications and compute-intensive servers. The system provides a shared memory programming environment on a cluster of workstations interconnected by a high speed interconnect. The principal design features of Beehive include: a global address space across the cluster (Section 5), a configurable access granularity (Section 6), flexibility in supporting spatial and temporal notions of synchronization and consistency (Section 7), and multithreading (Section 4). The fundamental design principle is to use only commodity hardware and software components as the basis, and build the shared memory system architecture entirely in software. The mechanisms for shared memory parallel programming are made available to the application programmer via library calls. We consciously target application domains which are expected to be ideally suited for cluster parallel computing in designing the system architecture of Beehive. In particular, we base our design on our understanding of the requirements of interactive applications such as virtual environments, our work in the storage architecture of database servers [9, 10], as well as our experience in parallel computing for scientific domains [24, 30, 32]. Figure 1 pictorially depicts the current organization of the Beehive cluster. Each box of the cluster can be a uniprocessor or an SMP. We do not address heterogeneity in the processor architectures in our current design. The interconnect can be realized out of any commodity network switch so long as they have the right latency properties for shared memory style communication. The requirements from the operating system to support the Beehive system architecture are: a *network file system*, and the ability to *specify a virtual address range* during memory allocation – a feature which is easily implementable in most Unix operating systems (e.g. using the *mmap* system call). In addition to these two requirements, a *thread-aware* operating system would be a plus for supporting the multithreading features of Beehive.

The current implementation of Beehive uses Sun UltraSparc CPU boxes (Uniprocessors or SMPs) running Solaris operating system, interconnected by two different networks – 100 Mbit/Sec Fast Ethernet and Myricom's Myrinet switch. The following sections discuss the implementation and design aspects of the various system components.

*This work has been funded in part by NSF grants MIPS-9058430 and MIP-9630145 and NSF II grant CDA-9501637.

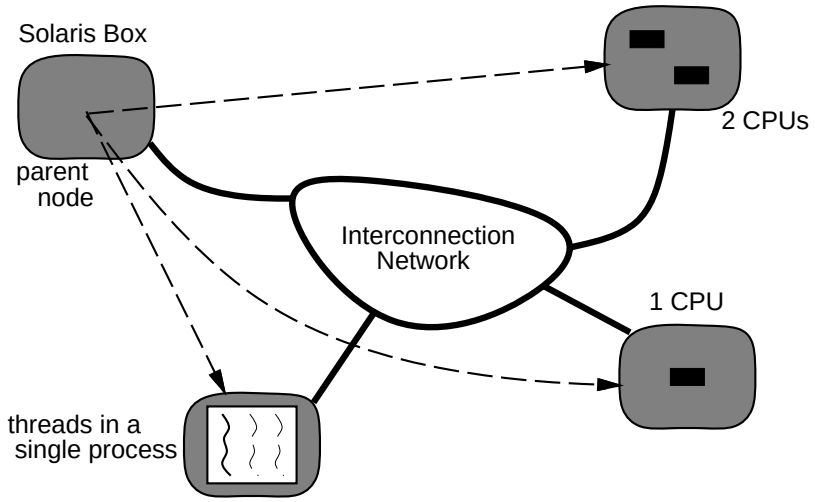

Figure 1: The Beehive Cluster

2 System Architecture

Parallel applications are developed on any node of the cluster using the Beehive API (Table 1), compiled, and linked to the Beehive libraries as shown in Figure 2. The resulting binary image can be started up on any node of the

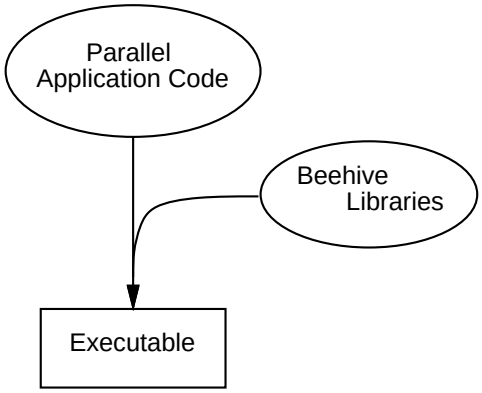

Figure 2: Beehive Applications

cluster, which is referred to as the `parent' node. Upon startup at the parent node, the executable reads a Beehive configuration file to identify the other nodes currently participating in the cluster. A Beehive process (a Unix process in the current implementation) is started up at each of the participating cluster nodes by the parent node to execute the same binary image. The reliance on the availability of a network file system is precisely for the purpose of not having to ship the executable to each node of the cluster. Furthermore, a global name space for the file system is usually assumed in programming parallel applications. The parent node sets up system level communication (see Section 3) with and among each of these remote Beehive processes. Every Beehive process creates several user level threads within its context, and associates a few of these threads exclusively for system level activities needed for performing the control, synchronization, memory allocation, and communication aspects of the Beehive API. User level thread is the vehicle for supporting multithreading at each node of the cluster and allows exploiting the SMP nature (if any) of each node of the cluster.

The current implementation of Beehive sits on top of Solaris – a thread-aware operating system for Sun SPARCsta-

Control	
bh_fork	fork a beehive thread to local/remote node
bh_join	join with a beehive thread
Global Memory	
bh_sh_malloc	allocate a piece of global memory
Communication	
sh_read	shared memory read
sh_write	shared memory write (invalidation based)
pset_write	explicit shared memory communication
sync_write	"
selective_write	"
Synchronization	
bh_tg_sync	temporal time group synchronization
bh_barrier	barrier
bh_lock	mutex lock
bh_unlock	mutex unlock

Table 1: Beehive API

tions [23]. Solaris allows the creation of multiple units of CPU scheduling called *lwp* (light-weight process) within the same address space. It also allows the creation of multiple user level threads within the same address space with a many-to-one mapping between the user-level threads and the lwp's. This is a particularly efficient way of using multiple CPUs on the same node for the same address space, a feature which suits the use of threads at the system level as well as the application level in Beehive.

Figure 3 shows the software system architecture of Beehive. The messaging layer provides an an "active message" [35] interface to the system components above it. This layer is invisible at the level of the application programs, and is used by the Beehive system components to accomplish the functionalities specified in the Beehive API. As is evident from the figure, these system components manage the control, synchronization, and shared memory style communication functions. We describe the messaging layer and these system components in more detail in the following sections.

3 Messaging Layer

To make cluster parallel computing a viable option for compute-intensive applications, it is imperative to have a low latency message communication mechanism. Fortunately, advances in networking technology have made low latency switches (e.g. Myricom Myrinet [2]) available for interconnecting PCs and workstations. Further, recent work in low latency messaging has resulted in the development of highly optimized communication kernels that exploit the low latency properties of such interconnects. *UNet* [34] running on top of ATM and fast Ethernet, and *Fast Messages* (FM) [21] on top of Myrinet are two such examples.

We have built the Beehive messaging layer in a flexible manner to integrate any such communication kernel into the system. The Beehive message interface to the system components above it consists of two calls: *msg_alloc* and

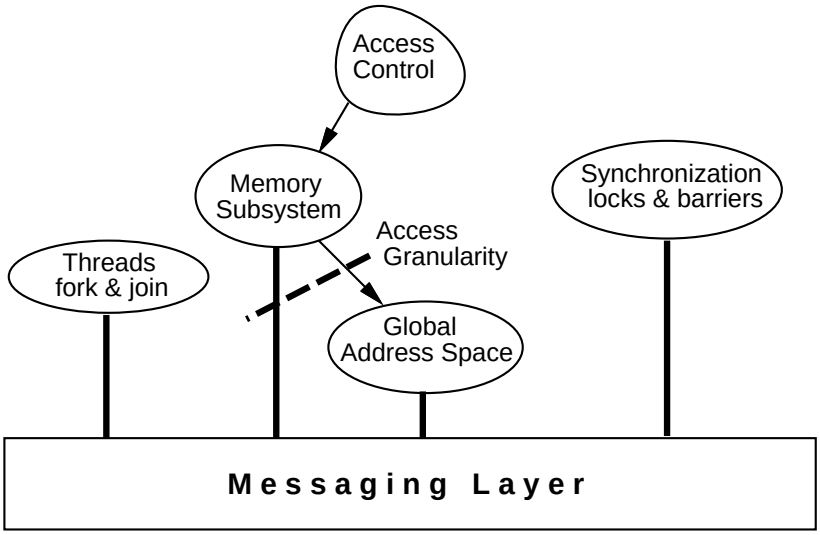

Figure 3: Beehive System Architecture

msg_send. The msg_alloc call (Figure 4) returns a message buffer, which can be filled up with the desired handler address and arguments. *Msg_send* is responsible for the message reaching the remote node. A *daemon* thread

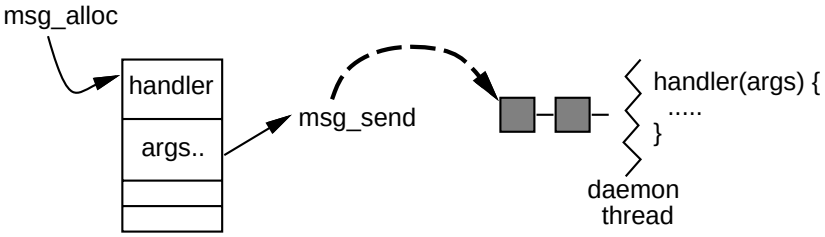

Figure 4: Active Message Interface

executes the handler with the arguments (both of which are specified in the message) on behalf of the Beehive system at the receiving end. Currently, we have two different implementations of the messaging layer. The first one sits on top of UDP datagram sockets using a 100Mbit/Sec Fast Ethernet, and implements a window-based reliable packet protocol since the UDP datagram service does not guarantee delivery of packets. The observed latency for this implementation is about 400 microseconds for a one-way null message. The second implementation sits on top of Fast Messages Release 2.0 [22] using Myrinet switches. The corresponding latency in this case is about 30 microseconds.

Consistent with the shared memory programming paradigm, the messaging layer is invisible to the programmer. However, the observed latency for internode communication can have a significant impact on the shared memory mechanisms we provide in Beehive. These are discussed in [31].

4 Threads Support

Multithreading is an important technique for hiding latency of expensive remote operations [27]. It is also useful to increase concurrency when a thread blocks for any reason – either for resources on a single node or at a synchronization point. With the recent proposals for multithreading single chip CPU architectures [33], and the trend

towards making workstation nodes out of SMP boxes it seems natural to support multithreading at the application programming level to make full use of the available hardware resources.

The Beehive API provides the *bh_fork* call to allow applications to create user level threads on local as well as remote nodes. Since Beehive provides a per-application shared address space across the cluster (see Section 5), all these threads belong to the same address space. The application program can either specify the node on which the thread should be created, or let the Beehive runtime system use the default policy for thread creation (round-robin among the nodes of the cluster). Currently, Beehive does not support thread migration. Thus the node-id is part of the thread-id. The *bh_join* call allows application level pair-wise join synchronization between threads belonging to the same application.

The Solaris user-level threads are used to implement this cluster-wide fork-join API of Beehive. A fork generally involves sending an active message (through the messaging layer) from the requesting node to execute a fork handler on the target node. A join is implemented similarly. The Beehive runtime system itself is multithreaded.

5 Global Address Space

Global address space is the simplest and most basic abstraction for state sharing in any distributed memory platform. The basic premise behind this abstraction is that with 64-bit processor architectures and 64-bit operating systems, there is sufficient amount of virtual address space so that it is reasonable to partition the total address space needed for an application among the nodes of the cluster. The current implementation is on a 32 bit Solaris operating system, but can be trivially extended to a 64 bit virtual address OS implementation as soon as one is available for the UltraSparcs.

As shown in Figure 5, the entire virtual address space on a single node is partitioned into two regions: private and shared. The private region is the same address range on all nodes and contains program code, static data, thread

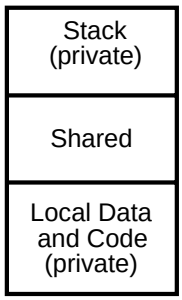

Figure 5: Partitioning the virtual address space

stacks and heap areas for the application. The state information internal to the Beehive system is also located in this area. As the name implies, this region is not shared among the nodes. The virtual memory map for the shared region is invariant for all the nodes which results in a global address space. Reference to a particular address in the shared region refers to the same piece of data across the cluster.

The global address space itself is cyclically partitioned among the nodes of the cluster to identify a home node for a particular range of addresses (Figure 6). The chunk size for this allocation can be determined at runtime to give the application control over the distribution of shared addresses. This division is done in such a way that a given virtual

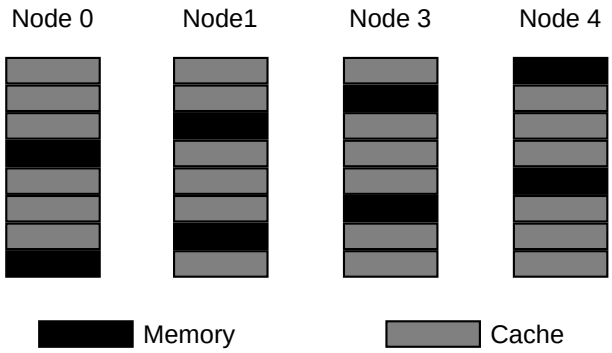

Figure 6: Cyclic partition of global address space

address from the shared area can uniquely identify the home node using a few bits for that address. The home node is responsible for maintaining the directory information for the global address space assigned to it, and doing the memory subsystem activity on its behalf. The backing store for a given virtual address on the home node is the primary memory copy, while the backing stores at the other nodes for this same address are "cache" copies. The backing store to a virtual address range is allocated at a non-home node only if that node accesses data in that range. Note that the virtual to physical memory mapping is internal to each node and is orthogonal to the realization of the global address space. The virtual memory page can use any mechanism - the traditional swap space, or network global memory [6].

This policy provides address equivalence for the global address space throughout the cluster - the same address refers to the same datum. This approach is similar to the one taken in Shasta DSM system [28]. The other approach taken by several DSM systems [3, 13, 14] is to have name equivalence. Our approach has the merit of saving the overhead of maintaining name tables required in supporting name equivalence, and making pointers valid uniformly across nodes. The only OS level support relied upon in achieving this partitioning of shared address space is simply being able to specify a virtual memory range during memory allocation, a feature which is easily implementable in most Unix operating systems using the *mmap* system call. Beehive API provides a *shared_malloc* call to the application program for global memory allocation.

6 Configurable Access Granularity

Different applications need different granularity of access to the shared address space varying from a few tens of bytes to a page size. One of the primary goals of Beehive is to decouple the unit of coherence maintenance and data transfer between nodes from the virtual memory dictated page size. The obvious benefit is that this would allow us to provide fine-grain sharing for applications that need it and avoid the pitfalls of false sharing. Beehive can be configured to use any given cache line size for different applications. This determines the coherence granularity at which the cache and memory directory information is maintained, and the communication granularity at which data transfers take place. Having an application specific granularity relieves the need and overhead of *twinning* and *diffing* present in most software DSM systems [3, 13, 14]. A configurable access granularity has also been implemented in the Shasta DSM system [28].

The more important aspect of access granularity is the access control mechanism. It is necessary to track potential violations of coherence and trigger the memory subsystem to take appropriate consistency actions if and when

required. Currently, we provide access to the shared memory region via explicit read/write primitives (see Section 7). Since we work in close cooperation with the application developers it is fairly straightforward for us to annotate the application code by hand. However, the Beehive memory subsystem can co-exist with any of the schemes for fine grain access control such as using dedicated hardware to snoop on the processor-memory bus [1, 4, 20], using the memory ECC bits to generate access violation traps [25], and using binary editing tools to explicitly check each load/store in the program [29, 28]. We are also exploring compiler-assistance for access control to arrive at a long term solution that is hardware platform independent.

7 Flexible Memory Subsystem

The Beehive system architecture basically allows using the local physical memory as a "cache" for portions of the global virtual address space that are accessed by a particular node. It is this cache which is managed by the Beehive memory subsystem to be described in this subsection. The management of the hardware caches in the processor itself are purely local to each node and does not concern this software architecture.

The primary contribution of this work is the design and implementation of a flexible software memory subsystem. Spatial notions of consistency and synchronization are well-known and are supported in Beehive. In [31], we have argued for the performance potential and programming ease of temporal notions of consistency and synchronization in the context of interactive applications. Further, the consistency and synchronization requirements associated with different data within an application and across applications could vary significantly. One static approach to deal with this situation is to provide a menu of coherence protocols and allow the applications to match the different consistency requirements for different data [3, 17]. A more dynamic approach is to track the sharing patterns in applications and adapt the coherence protocol to match this sharing pattern. It is well known that message passing programs, which have complete control over the management of and access to shared data, are able to deliver much better performance compared to shared memory programs. However this is at the cost of programming ease that comes with the shared memory paradigm. So the key here is to provide the flexibility in the memory system to derive the performance advantage of message passing without sacrificing the programming advantage of shared memory. This issue has been the focus of several recent research endeavors [7, 11, 16, 19, 26].

The Beehive memory system provides this flexibility along two orthogonal dimensions. Along one dimension we exploit knowledge of data sharing patterns in the applications to decide *whom* to send the data to (i.e the `future sharer set'). This builds on our earlier work [24] wherein we proposed a set of explicit communication mechanisms within the confines of coherent shared memory. Along the other dimension we exploit the opportunities that may exist in the applications for deciding *when* to send the data to the sharers. This is achieved via supporting different consistency levels based on application requirements.

7.1 Software Architecture

Figure 7 shows the software architecture of the memory subsystem. The memory subsystem supports access to shared memory via two calls: *sh_read* and *sh_write*, which are the simple reads and writes to shared memory. The base cache coherence protocol is a directory-based write-invalidate protocol. Combined with the explicit communication primitives to be described shortly, the architecture has to support both invalidation and update mechanisms. The memory access primitives follow a specific path through the various software components of the system de-

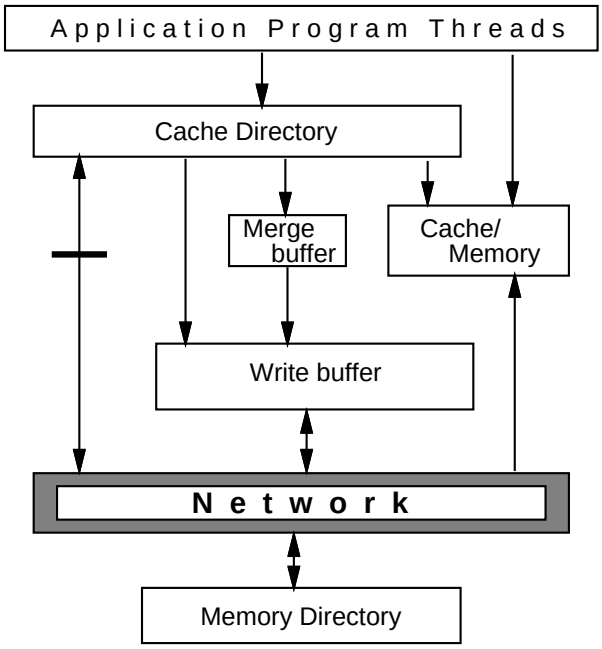

Figure 7: Software Architecture of the Memory Subsystem

pending on the type of request. The cache directory maintains the cache state for local cache lines, and the memory directory maintains the current sharer set in a `presence vector' using one bit per node. For writes to be non-blocking, we use a write buffer [8] to store all pending writes. The write buffer is actually implemented as a bit-vector for each cache line (per-byte dirty bits to indicate parts of the line that are dirty at this node). Data transfer between nodes are on a per-line basis which is the unit of coherence maintenance. Incoming remote writes are compared against this bit vector to selectively update the corresponding cache line. The actual data is written in place. This technique basically eliminates the need for twinning and diffing of shared pages [3, 13, 14], or doubling of writes either in software [15] or in hardware [12] done in some page-based DSM systems to avoid false sharing effects. A merge buffer [5] is used to smoothen the outgoing update traffic. These components are at a node level, with multiple application threads being associated with the same cache, directories, write, and merge buffers.

7.2 Explicit Communication

The idea behind explicit communication mechanisms is pretty straightforward. We want to push the data out upon a write to “future” readers so that the read-miss latency is avoided at the consumer end. The communication overhead in a simple-minded write-update protocol for cache coherence far outweighs the potential reduction in read-miss latency. Our solution is to selectively send updates on top of the base write-invalidate protocol. In our earlier work [24], we evaluated the efficacy of three primitives corresponding to three different situations that we encountered in scientific applications. The difference among the three primitives is simply the heuristic used to determine the “future” consumers. In **pset_write**, the set of consumers is statically determinable at compile time and is specified as a *processor mask* to the primitive. In **sync_write**, the next (if any) acquirer of a mutex lock is available from the lock structure. This information is used directly to push the data written during the critical section governed by this mutex lock to the prospective consumer node. The third primitive, **selective_write**, is the most general for use in situations where the communication pattern cannot be discerned *a priori*. In such situations, the following heuristic is used which seems to work quite well. Upon selective_write, push the data to the current sharers as given

by the `presence vector' for that memory location (available in the memory directory). This basically switches the coherence protocol to a write-update from a write-invalidate. However, if the sharer set changes again (i.e. a new reader joins the set) then invalidate the current sharer set and revert to the base write-invalidate protocol until the next selective_write.

7.3 Support for Different Consistency Levels

Along the other dimension, control over flushing of the write and merge buffers gives us opportunities to relax the consistency model and decide `when' to launch consistency actions. Supporting sequential consistency (SC) [18] and release consistency (RC) [8] are pretty straightforward. For SC the buffers are flushed on each write access, whereas for RC they are flushed at each release synchronization point.

Supporting the delta consistency mechanism developed in [31] requires additional machinery. The current implementation uses real time as the frame of reference for the consistency actions. Upon the first write to a particular shared memory location, the corresponding merge buffer entry is timestamped. Entries in the merge buffer are selectively flushed whenever `$delta - l$ time units have elapsed since this timestamp (where *delta* is application specified, and l is the expected latency on the network). As we said earlier, the mechanism for temporal synchronization (tg_sync) and delta consistency are inter-twined. In fact, the system gets control from the application to take consistency actions either at read/write to shared memory or at tg_sync points. Therefore, the consistency actions are launched either immediately upon a write (if *delta* is less than the time remaining to the next tg_sync) or at a subsequent tg_sync point commensurate with *delta*.

7.4 Support for Synchronization

The memory subsystem also provides the spatial synchronization primitives – mutex locks and barrier. The *sync_write* primitive which we described earlier uses this lock structure to obtain the identity of the next node (if any) waiting for the lock to determine whom to push the data to. The memory system also implements temporal synchronization using the real time clock.

The Beehive system architecture is designed with extensibility in mind. The merit of this organization lies in the ease with which new access primitives can be added to the memory subsystem without changing the underlying layers. Also new mechanisms can be introduced into the lower generic layers very modularly. Table 1 summarizes the control, synchronization, global memory allocation, and communication primitives available in Beehive.

References

[1] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the 21th Annual International Symposium on Computer Architecture*, pages 142–153, April 1994.

[2] N. J. Boden et al. Myrinet: A Gigabit-per-second Local Area Network. *IEEE Micro*, pages 29–36, February 1995. Myricom, Inc.

[3] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *The 13th ACM Symposium on Operating Systems Principles*, October 1991.

[4] D. R. Cheriton and K. J. Duda. Logged Virtual Memory. In *Fifteenth Symposium on Operating Systems Principles*, 1995.

[5] F. Dahlgren and P. Stenstrom. Using Write Caches to Improve Performance of Cache Coherence Protocols in Shared Memory Multiprocessors. Technical report, Department of Computer Engineering, Lund University, April 1993.

[6] M. J. Feeley, W. E. Morgan, F. H. Pighin, A. R. Karlin, H. M. Levey, and C. A. Thekkath. Implementing Global Memory Management in a Workstation Cluster. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, December 1995.

[7] M. I. Frank and M. K. Vernon. A Hybrid Shared Memory/Message Passing Parallel Machine. In *Proceedings of the 1993 International Conference on Parallel Processing*, 1993.

[8] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. L. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[9] V. Gottemukkala, U. Ramachandran, and E. Omiecinski. Relaxed Index Consistency for a Client-Server Database. In *International Conference in Data Engineering*, 1996.

[10] S. Gukal, U. Ramachandran, and E. Omiecinski. Transient Versioning for Consistency and Concurrency in Client-Server Systems. In *PDIS*, 1996.

[11] J. Heinlein, K. Gharachorloo, S. A. Dresser, and A. Gupta. Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.

[12] L. Iftode, J. P. Singh, and K. Li. Understanding Application Performance on Shared Virtual Memory Systems. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[13] P. Keleher, S. Dwarkadas, A.L. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the Winter 94 Usenix Conference*, pages 115–131, January 1994.

[14] P. Kohli, M. Ahamad, and K. Schwan. Indigo: User-level Support for Building Distributed Shared Abstractions. In *Fourth IEEE International Symposium on High-Performance Distributed Computing (HPDC-4)*, August 1995.

[15] L. I. Kontothanassis and M. L. Scott. Software Cache Coherence for Large Scale Multiprocessors. In *Proceedings of the First International Symposium on High Performance Computer Architecture*, pages 286–295, January 1995.

[16] D. Kranz, K. Johnson, A. Agarwal, J. Kubiatowicz, and B-H Lim. Integrating Message-Passing and Shared Memory: Early Experience. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 54–63, May 1993.

[17] J. Kuskin et al. The Stanford FLASH multiprocessor. In *Proceedings of the 21th Annual International Symposium on Computer Architecture*, pages 302–313, April 1994.

[18] L. Lamport. How to make a Multiprocessor Computer that Correctly executes Multiprocess Programs. *IEEE Transactions on Computer Systems*, C-28(9), 1979.

[19] J. Lee and U. Ramachandran. Architectural Primitives for a Scalable Shared Memory Multiprocessor. In *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 103–114, Hilton Head, South Carolina, July 1991.

[20] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, M. Parkin, B. Radke, and S. Vishin. The s3.mp Scalable Shared Memory Multiprocessor. In *International Conference on Parallel Processing*, pages Vol I:1 – 10, 1995.

[21] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of Supercomputing '95*, 1995.

[22] S. Pakin, M. Lauria, et al. Fast Messages (FM) 2.0 User Documentation.

[23] M. L. Powell, S. R. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks. SunOS Multi-thread Architecture. In *Proceedings of the 1991 Winter Usenix Conference*, 1991.

[24] U. Ramachandran, G. Shah, A. Sivasubramaniam, A. Singla, and I. Yanasak. Architectural Mechanisms for Explicit Communication in Shared Memory Multiprocessors. In *Proceedings of Supercomputing '95*, December 1995.

[25] S. K. Reinhardt et al. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *Proceedings of the ACM SIGMETRICS 1993 Conference on Measurement and Modeling of Computer Systems*, pages 48–60, Santa Clara, CA, May 1993.

[26] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User Level Shared Memory. In *Proceedings of the 21th Annual International Symposium on Computer Architecture*, pages 325–336, April 1994.

[27] R. H. Saavendra-Barrera, D. E. Culler, and T. von Eicken. Analysis of Multithreaded Architectures for Parallel Computing. In *ACM Symposium on Parallel Algorithms and Architecture*, July 1990.

[28] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[29] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Laurus, and D. A. Wood. Fine-grain Access Control for Distributed Shared Memory. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297–306, October 1994.

[30] G. Shah, A. Singla, and U. Ramachandran. The Quest for a Zero Overhead Shared Memory Parallel Machine. In *International Conference on Parallel Processing*, pages Vol I:194 – 201, 1995.

[31] A. Singla, U. Ramachandran, and J. Hodgins. Temporal Notions of Synchronization and Consistency in Beehive. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, 1997.

[32] A. Sivasubramaniam, A. Singla, U. Ramachandran, and H. Venkateswaran. A Simulation-based Scalability Study of Parallel Systems. *Journal of Parallel and Distributed Computing*, 22(3):411–426, September 1994.

[33] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting Choice: Instruction Fetch and Issue on a Implementable Simultaneous MultiThreading Processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[34] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[35] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.