JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING **40,** 131–137 (1997)
ARTICLE NO. PC961273

Low-Cost, High-Performance Barrier Synchronization on Networks of Workstations

DONALD JOHNSON, DAVID LILJA,* JOHN RIEDL, AND JAMES ANDERSON†

*Department of Computer Science, and *Department of Electrical Engineering, University of Minnesota, Minneapolis, Minnesota 55455; and †Department of Electrical and Computer Engineering, Purdue University, West Lafayette, Indiana 47907*

Circulating active barrier (CAB) is a new low-cost, high-performance hardware mechanism for synchronizing multiple processing elements (PEs) in networks of workstations at fine-grained programmed barriers. CAB is significantly less complex than other hardware barrier synchronization mechanisms with equivalent performance, using only a single conductor, such as a wire or copper run on a printed-circuit board, to circulate barrier packets between PEs. When a PE checks in at a barrier, the CAB hardware will decrement the count associated with that barrier in a bit-serial fashion as a barrier packet passes through, and then will monitor the packets until all PEs have checked in at the barrier. The ring has no clocked sequential logic in the serial loop. A cluster controller (CC) generates packets for active barriers, removes packets when no longer needed, and resets counters when all PEs have seen the zero-count. A hierarchy of PEs can be achieved by connecting the CCs in intercluster rings. When using conservative timing assumptions, the expected synchronization times with optimal clustering are shown to be under 1 μs for as many as 4096 PEs in multiprocessor workstations or 1024 single-processor workstations. The ideal number of clusters for a two-dimensional hierarchy of N PEs is shown to be $[N(D + G)/(I + G)]^{1/2}$, where G is the gate propagation delay, D is the inter-PE delay, and I is the intercluster transmission time. CAB allows rapid, contention-free check-in and proceed-from-barrier and is applicable to a wide variety of system architectures and topologies. © 1997 Academic Press

1. INTRODUCTION

A network of workstations (NOW) is an attractive platform for many parallel processing tasks, with networking technologies turning the raw computational power of a cluster of inexpensive components into a high-performance multiprocessing system [4, 7, 15, 22]. Specialized interconnections and adapters that allow for user-level access to shared data and synchronization are necessary for dramatic performance improvement in scalable parallel computers or NOWs [21]. All areas of fine-grained interaction need such improvements, but the focus of this paper is restricted to the area of barrier synchronization for fine-grained applications (those with fewer than approximately 1000 instructions between synchronizations).

Barriers are important synchronization operations in multiprocessor systems [5, 19]. When a processor executes a barrier instruction, it first checks in at the barrier to indicate to the other processors that it has arrived at the specified synchronization point. It then must wait for all other processors participating in the barrier to check in, after which all processors can proceed past the barrier to begin executing their next assigned tasks. Tasks may include parallel loop iterations [16], emulation of very long instruction words (VLIW) [14], or dataflow steps [11].

Current barrier implementations typically use software trees [18], software- or hardware-based counters [12, 13, 17], or hardware fan-in trees [3, 5, 6, 19, 20]. While the software trees are inexpensive, their synchronization delay is too long for fine-grained applications, especially on systems lacking a shared-bus. Counter-based methods introduce potentially high contention to access the shared counters, which can lead to unacceptably long, and unpredictable, synchronization delays. However, the best hardware solutions are not easily adapted to a wide range of existing systems, but depend on a particular topology, or even a particular implementation. The fan-in tree is an example of a class of hardware solutions with very good performance. The hardware required to implement the best techniques is often complex and expensive, requiring $O(N^2)$ wiring complexity [5, 20] for the fastest performance, a severe problem on networks of workstations. Slow and expensive associative memory may also be required [20]. Note that fine granularity precludes using barriers in a multiprogramming environment since a task-switch by one PE would delay any barrier(s) in which it is involved until that PE resumes execution. Extremely fast context-switching [1, 8] can be used to mitigate this problem.

The barrier mechanism proposed in this paper is inexpensive, using only simple bit-serial hardware in each PE with only a single-conductor serial ring between PEs. Since the ring itself has no clocked sequential logic, the only latency introduced by the barrier hardware is a single gate delay for each PE, plus the conductor's unavoidable time-of-flight delay between PEs. This configuration allows the performance of this new barrier mechanism to approach the performance of a fully connected hardware fan-in tree.

2. CIRCULATING ACTIVE BARRIER (CAB) SYNCHRONIZATION

The existing barrier synchronization mechanisms typically trade off cost and complexity with performance. An ideal barrier mechanism, however, would:

(1) allow for multiple barriers, each with potentially different sets of processors;

(2) have user-level access to avoid context-switching into the OS kernel;

(3) allow for rapid check-in with no contention in the typical case;

(4) have rapid execution resumption when all PEs have checked-in; and

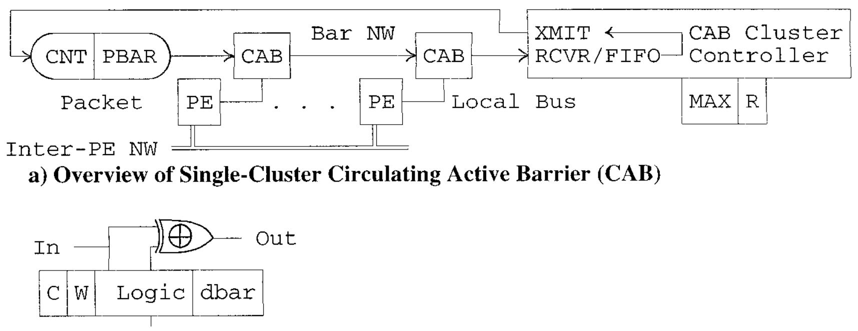
(5) be applicable to a wide variety of system architectures and topologies.

CAB meets these goals by using simple and inexpensive hardware with an integrated special-purpose bit-serial network between processors to take advantage of the simplicity of serial data operations while minimizing the latency such operations introduce. An added benefit of this simplicity is minimized cost of the replicated hardware at each PE, with somewhat more complexity in the shared cluster controller.

Barriers that are available for PE check in are classified as "active" and are circulated around a ring of CAB hardware modules, each one attached to a single PE. Each active barrier is in a packet consisting of a barrier identifier and a count of the remaining PEs to check-in at that barrier. When a PE reaches a barrier, it loads its CAB hardware with the barrier identifier that it is awaiting. If done in a blocking mode, the CAB hardware activates an appropriate local signal, such as I/O wait, to block its PE until all PEs for that barrier have checked in (uniprocessing is assumed for each PE). If done in a nonblocking mode, a fuzzy barrier [9] is created so that a PE can do other work after check-in before explicitly waiting for barrier completion. The CAB hardware will monitor the packets as they pass through it, detecting a match (using a bit-serial exclusive-OR) when the desired barrier reaches it. A bit-serial decrement of the count field of the packet is then performed.

The comparison and decrement operations within each CAB are done with only combinational logic in the loop itself. There are, of course, sequential components within the CAB hardware that must be able to control the loop data during the next clock cycle. As a result, each node in the ring adds only the delay of a single XOR gate to the ring latency, regardless of the clock speed. Following the decrement operation, the CAB hardware continues to monitor the packets as they pass through it until a barrier matches with a count of zero, at which time it unblocks its PE. Note that the last PE completing check-in is immediately unblocked. A cluster controller is responsible for placing active barriers on the ring, resetting counts when all PEs have proceeded from a barrier, and removing barrier packets when no longer needed.

Figure 1 shows an example implementation of the CAB with barrier packets inserted (with CNT = MAX) or deleted by the cluster controller, which would typically be attached to one of the loop node PEs. The CAB at each PE is able to detect a match of the desired barrier (dbar) and the packet's barrier (PBAR), and to decrement the packet's count (CNT) or detect a count of zero. Note that



a) Overview of Single-Cluster Circulating Active Barrier (CAB)

b) CAB per PE

PBAR:	Circulating active barrier identifier (cluster controller inserts/deletes packet)
CNT:	Number of PEs remaining to check-in at PBAR
dbar:	Desired barrier # PE is awaiting (if W is true)
W flag:	True when waiting for more PEs to check in at dbar
C flag:	True when waiting for PBAR=dbar so that CNT can be decremented
MAX:	Array of number of PEs, one for each active barrier
R flag:	Array of reset flags, one for each active barrier (true when zero is circulating)
Logic:	Includes clock generator, ID comparator, and decrement circuitry

FIG. 1. Barrier synchronization hardware.

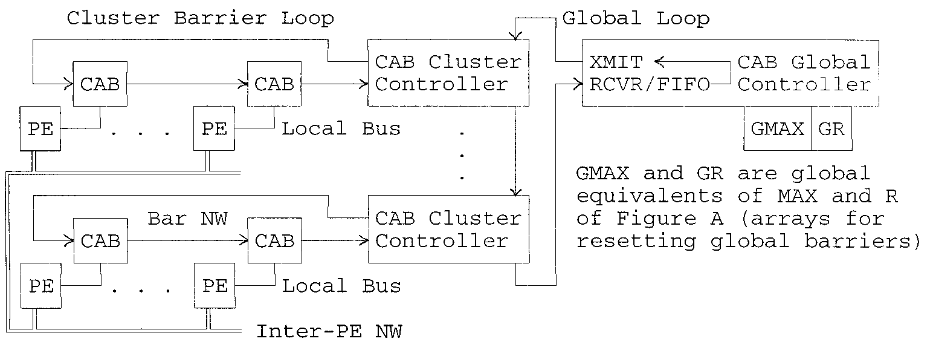


FIG. 2. Overview of a two-level synchronization barrier.

a zero count for a packet is the ‘‘all-checked-in’’ indicator. When the cluster controller detects that a counter has become zero, a flag, R_n, is set to indicate that barrier packet with a zero count is being circulated. When that zero again reaches the controller, the counter is reset to the MAX_n value adjusted for any PEs checked in at the new barrier. CAB modules starting with the final one to check in will see the resetting packet twice, and if any are already checked in for the next synchronization, CNT will become negative (2’s complement) so that no early check-ins are lost. Note that any number of barrier counters are possible, limited only by the number of PBAR bits, and that any barrier may be used by any PE. For example, this structure allows nested barriers, which could be the result of nested loops.

The CAB hardware must be accessible at the user-level to allow rapid access, but it must be protected from access by an unauthorized PE. For example, more than one set of PEs may have independent sets of barriers on a particular cluster. Since the barrier identifiers are physically available to any PE on the cluster, the CAB hardware must protect the integrity of the system by mapping logical barrier IDs into authorized physical IDs. On some systems, external register-mapping may be used for fastest performance [10], but most systems can at least accommodate memory-mapping. The available barriers must be mapped into hardware using the kernel mode execution by using physical barrier identifiers supplied by a barrier system call similar to that for acquiring and releasing UNIX ports. For performance considerations, however, their *use* must not involve context-switching into the kernel mode. The barrier server would also initiate generation and removal of circulating barriers through communication with the cluster controller. The PE that initializes the barrier is responsible for communicating with the cluster controller to insert a new barrier packet for circulation. Also, this PE must send the barrier number to all participating PEs to initialize their CAB modules using the general-purpose inter-PE communication network, which would also be used by error recovery protocols.

Although the proportionality constant is small because of the absence of sequential logic in the loop, the $O(N)$ loop time results in unacceptable latencies for a large number of PEs, making a hierarchy of loops desirable. In this paper we limit our discussion to two levels of hierarchy, though extensions to additional levels are expected to be straightforward. Barriers that involve only PEs within a cluster would be handled by the mechanism already described. If PEs accessing the same barrier reside in different clusters, an intercluster loop, similar to the intracluster loop, will be needed. The intercluster part of each CC shown in Fig. 2 will look similar to a cluster’s CAB module. A CC sets CNT to local PEs + 1 and detects when a global barrier’s CNT = 1 within the local cluster, and then will check in at the appropriate global barrier, whose count was originally the number of clusters required to check in. When a CC sees a global-loop barrier with a count of zero, it will finish global check-in by circulating the matching local barrier with a zero count. Checked-in global barriers within a cluster will be simultaneously checked by serial comparators in parallel. For a two-level hierarchy, two queued comparators are most likely sufficient since a particular cluster is unlikely to be involved in more than two global barriers, and global barriers would normally be involved only if the number of PEs for a barrier exceeds the number of PEs in a cluster. As on the local loop, there are no serial components in the global loop outside of the global controller to minimize the latency.

3. PERFORMANCE EVALUATION

If properly mapped [13], only Nodes/2 barriers must be accommodated in a loop, with each barrier capable of counting all nodes. Since the count requires lg Node bits ($\lg = \log_2$) and the barrier identifier requires lg(Nodes/2) bits, the optimized packet size for either a cluster or an intercluster ring which can accommodate all possible reachable barriers would be of bit length

$$2 \lg \text{Nodes} - 1. \tag{1}$$

The packet length would be greater than the optimum if a parity bit or additional barrier identifier bits were added, making the maximum latency between 4 and 17% longer for 1–2 added bits for closely spaced PEs. The impact of adding one or two bits per packet on a network

TABLE I
Parameters for Performance Equations

Parameter description	*Parameter description*	*Parameter description*
D Inter-PE propagation time	I Intercluster propagation time	X Total global CNT bits
G Gate-delay time	N Total PEs	Y Total global PBAR bits
P Period of Xmit clock	Z Total clusters	N/Z Total PEs per loop
C No. of CNT bits	B Total PBAR bits	lg N $\log_2 N$

between workstations would be minimal since conductor latency, not bit-time, is the major limiting time factor.

The various parameters used in this section are described in Table I. The time from the last check-in until all PEs are notified is the release latency produced by the CAB synchronization mechanism. For barriers totally within a single cluster, the lowest time (T_l) involves one traversal of the packet around the loop, with check-in occurring just prior to the desired packet arriving at the "last-in" node. This time includes the time the packet spends in the cluster controller, $(C + B + 1)P$, since the entire packet must be inside the controller before a decision can be made as to whether to recirculate or remove a barrier (which takes one clock cycle). The traversal time outside of the controller, $(D + G)N$, includes the sum of all gate delays and inter-PE propagation times. The total lowest time is then

$$T_l = (D + G)N + (C + B + 1)P,$$

or if optimized via Eq. (1) with $C = \text{Lg } N$ and $B = C - 1$, the resulting time within the controller is $P \lg N$, yielding

$$T_l = (D + G)N + 2P \lg N. \tag{2}$$

The expected latency waiting for the desired packet would be one-half the loop time. Once the desired packet is seen, it must make, at most, a complete loop traversal (with a CNT = 0) before all PEs will be unblocked. Assuming the number of active barriers is small enough so that the FIFO in the cluster controller is empty, which is the typical case if the number of PEs per barrier is high, the mean expected time is then

$$T_e = 1.5T_l,$$

or if optimized,

$$T_e = 1.5(D + G)N + 3P \lg N. \tag{3}$$

The normal longest time would involve a complete loop traversal before encountering the desired barrier, leading to double the lowest time. The very worst case would involve a situation with $N/2$ active barriers (two PEs per barrier), which is extremely unrealistic. In this hypothetical case, the time spent outside the controller is insignificant, since the limiting factor is transmitting the backed-up packets within the controller. Assuming two loop traversals for the packet, with the resetting zero packet bypassing the controller's FIFO, the worst-case time is

$$T_w = (C + B)PN,$$

or if optimized,

$$T_w = (2 \lg N - 1)PN. \tag{4}$$

When multiple clusters are involved, the minimum latency includes times spent in the local loop, $(D + G)N/Z$, in the local cluster controller, $(C + B + 2)P$, which includes one bit-time for mapping to the global loop, in the global loop, $(I + G)Z$, and in the global controller, $(X + Y + 1)P$. The minimum time is then

$$T_l = (D + G)N/Z + (C + B + 2)P \\ + (I + G)Z + (X + Y + 1)P,$$

or if optimized with $C = \lg N/Z$, $B = C - 1$, $X = \lg Z$, and $Y = X - 1$, the lowest time is:

$$T_l = (D + G)N/Z + 2P \lg N + (I + G)Z + P. \tag{5}$$

The expected time is $1.5T_l$, which if optimized is

$$T_e = 1.5(D + G)N/Z + 3P \lg N + 1.5(I + G)Z + 1.5P. \tag{6}$$

Similarly the worst-case time is

$$T_w = (C + B)(N/Z + Z), P,$$

or if optimized,

$$T_w = (2 \lg N - 1)(N/Z + Z)P. \tag{7}$$

Figure 3 shows the expected synchronization time (T_e) based on two models: (a) a network of closely spaced single-PE workstations (SP-WS), and (b) a network of closely spaced multi-PE workstations (MP-WS: one cluster per workstation). The following values are assumed:

(1) packets are of the optimum size, 2 lg Nodes − 1, as described above,

(2) the XOR gate of the CAB has a propagation delay of 1.5 ns,

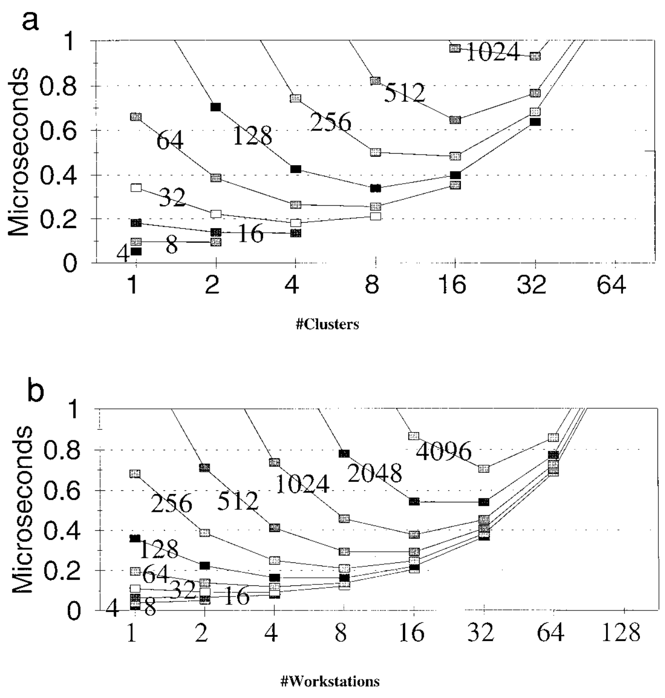


FIG. 3. Expected CAB synchronization time. (a) Network of single-PE workstations, and (b) network of multiprocessor workstations. Line labels are the total number of PEs involved in a barrier.

(3) the inter-PE time is 0.15 ns for MP-WS systems and 5 ns for SP-WS systems,

(4) the intercluster time is 10 ns for SP-WS and 5 ns for MP-WS,

(5) the serial clock period is 2 ns,

(6) only dedicated (single-task) PEs are involved in barriers, and

(7) the number of active barriers is small enough so that only one packet is in a cluster controller.

As can be seen from Fig. 3, the clustering becomes very important as the number of PEs increases because of the $O(N)$ loop-time outside the cluster controller. Each line represents a particular total number of PEs, so the number of PEs per cluster can be calculated by dividing by the number of clusters. Notice that most curves have an optimum clustering, producing a minimum synchronization time. This minimum depends on the type of system, but the slope is gentle enough so that a slightly "nonoptimum" choice would not be disastrous.

Rather than generating plots to observe the minimum synchronization time for a given configuration, it could prove useful to analytically determine the optimum clustering based on the number of PEs. Since

$$T_e = 1.5(D + G)N/Z + 3P \lg N + 1.5(I + G)Z + 1.5P$$

when using the optimized packet sizes, the optimum number of clusters occurs when T_e's derivative with respect to Z is zero:

$$I + G - (D + G)/Z^2 = 0,$$

or

$$Z = [N(D + G)/(I + G)]^{1/2}. \tag{8}$$

For the Fig. 3 examples, the optimum number of clusters is 0.75 $N^{1/2}$ (for SP-WS) or 0.50 $N^{1/2}$ (for MP-WS). For the SP-WS configuration, $D \approx 3G$ and I $\approx 7G$, resulting in a $(D + G)/(I + G)$ ratio near 0.5. For the MP-WS configuration, D has little influence, and $I \approx 3G$, making the ratio close to 0.25. Other configurations, with differing times, could be analyzed in a like manner.

In order to demonstrate ease of implementation and the validity of the timing analysis presented previously, a CAB module with a four-bit barrier number and four-bit counter was simulated [2]. The CAB module was implemented using LSI Logic's G10 CMOS ASIC library, which has a 0.29-μm effective transistor channel length and is optimized for 3.3-V supply operation. The Synopsys Design Analyzer was used to synthesize the VHDL code for a single CAB module. In addition to a bit-serial adder, five state machines were defined for the CAB functions: equal detect, zero detect, bit counter, check-in, and processor wait. The digital logic for the module, generated by the Design Analyzer tool for the G10 library, consisted of eight D flip-flops and 51 simple combinational gates. Once the digital logic was generated, LSI Logic tools were used to generate the floorplan for a CAB module. The chip area required for a single CAB module was found to be quite small: 115 × 115 μm. From the CAB layout, a transistor level HSPICE-compatible netlist with back-annotated parasitic layout capacitance and resistance was generated.

Meta-Software's HSPICE was used to perform a timing analysis of the CAB netlist with G10 transistor models supplied by LSI Logic. The propagation delay from the receive pin to the transmit pin through the single XOR gate was found to be 470 ps. A ring of CAB modules was also successfully simulated using a clock period of 2 ns.

4. SUMMARY

CAB is a fine-grained barrier synchronization mechanism that can be implemented with relatively inexpensive serial hardware while offering performance comparable to the highest performance known mechanisms. A contention-free ring with no clocked delays allows for multiple operations to be simultaneously performed. The mechanism allows for multiple and nested barriers and can be applied to both existing and new architectures.

As has been demonstrated, the expected synchronization times for the CAB mechanism can be under 1 μs for as many as 4096 PEs using multiprocessor workstations or 1024 single-processor workstations using conservative assumptions on parameters such as gate delay and electrical propagation time. As a result, fine-grained computation, such as VLIW emulation, becomes feasible using such networks of processors. CAB also provides a fast, contention-free mechanism for loop synchronization that will not adversely affect the performance of other inter-PE or PE-memory communications. When more than 16 PEs are involved in a barrier synchronization, implementing a hierarchical CAB system improves performance, with the optimum number of clusters for a two-dimensional hierarchy being $[N(D + G)/(I + G)]^{1/2}$, where N is the number of PEs, G is the gate delay, D is the inter-PE delay, and I is the intercluster time. The feasibility of the CAB system was demonstrated using detailed, logic-level simulations. A promising area for future work would be extending the mechanism beyond two levels of hierarchy.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under Grant MIP-9221900. A preliminary version of the work, including more operational details, was presented at the International Conference on Parallel Processing in August, 1995 [13]. Sincere thanks go to Darren Pearson and Paul Torgerson of the Minneapolis Circuits Technology Group of LSI Logic for their time and generous support.

REFERENCES

1. Alverson, R., Callahan, D., Cummings, D., Koblenz, B., Porterfield, A., and Smith, B. The Tera computer system. *International Conference on Supercomputing,* 1990, pp. 1–6.
2 Anderson, J. Simulation and analysis of barrier synchronization methods. University of Minnesota Tech. Rep. *TR HPPC-95-04,* University of Minnesota, 5/95.
3. Beckmann, C., and Polychronopoulos, C. Fast barrier synchronization hardware. *Proc. of Supercomputing '90,* Nov. 1990, pp. 180–189.
4. Bennett, J., Carter, J., and Zwaenepoel, W. Adaptive software cache management for distributed shared memory architectures. *Internation Symposium on Computer Architecture,* May 1990, pp. 125–134.
5. Cohen, W., Dietz, W., and Sponaugle, J. Dynamic barrier architecture for multi-mode fine-grain parallelism using conventional processors. *ICPP Proceedings,* 1994, Vol. 1, pp. 93–96.
6. Cray Research, *Cray T3D System Architecture Overview,* Oct. 1993, Chap. 3, pp. 24–27.
7. Delp, G., and Farber, D. Memnet: An experiment in high-speed memory-mapped local network interfaces. University of Delaware Tech. Rep. *Udel-EE-TR85-11-1R2,* University of Delaware, 1986.
8. Fan, X. Realization of multiprocessing on a RISC-like architecture. *Multiprocessing and Microprogramming,* June 1992, pp. 195–206.
9. Gupta, R. The fuzzy barrier: A mechanism for high speed synchronization of processors. *Third Architectural Support for Programming Languages and Operating Systems,* Apr. 1989, pp. 54–63.
10. Henry, D., and Joerg, C. A tightly-coupled processor–network interface. *Fifth Architectural Support for Programming Languages and Operating Systems,* Oct. 1992, pp. 111–122.
11. Iannucci, R. Toward a dataflow/Von Neumann hybrid architecture. *International Symposium on Computer Architecture,* 1988, pp. 131– 140.
12. Johnson, D., Lilja, D., and Riedl, J. A distributed hardware mechanism for process synchronization on shared-bus multiprocessors. *ICPP Proceedings,* 1994, Vol. 2, pp. 268–275.
13. Johnson, D., Lilja, D., and Riedl, J. A circulating active barrier synchronization mechanism. *ICPP Proceedings,* 1995, Vol. 1, pp. 202–209.
14. Lam, M. Software pipelining: An effective scheduling technique for VLIW machines. *SIGPLAN 88 Conference on Programming Language Design and Implementation,* June 1988, pp. 318–328.
15. Lenoski, D., Lauson, J., Gharachorloo, K., Weber, W., Gupta, A., and Hennessy, J. The directory-based cache coherence protocol for the DASH multiprocessor. *17th Annual International Symposium on Computer Architecture,* May 1990, pp. 148–159.
16. Lilja, D. Exploiting the parallelism available in loops. *Computer* **27** (Feb. 1994), 13–16.
17. Lubachevsky, B. Synchronization barrier and related tools for shared memory parallel programming. *ICPP Proceedings,* 1989, Vol. 2, pp. 175–179.
18. Mellor-Crummey, J., and Scott, M. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Systems,* **9** (Feb. 1991), 21–65.
19. O'Keefe, M., and Dietz, H. Hardware barrier synchronization: Static barrier MIMD (SBM). *ICPP Proceedings,* 1990, Vol. 1, pp. 35–42.
20. O'Keefe, M., and Dietz, H. Hardware barrier synchronization: Dynamic barrier MIMD (DBM). *ICPP Proceedings,* 1990, Vol. 1, pp. 43–46.
21. Snir, M. Scaleable parallel computer vs network of workstations, Keynote address at *1994 ICPP.*
22. Zhou, S., Stumm, M., and McInerney, T. Extending distributed shared memory to heterogeneous environments. *IEEE Trans. Computers* **39** (July 1990), 30–37.

DONALD E. JOHNSON earned a B.S. from University of Wisconsin-River Falls in 1966, a Ph.D. in chemistry from Michigan State University in 1970, and a Ph.D. in computer science from the University of Minnesota is expected in 1997. His research interests are in the area of improving performance for multiprocessor interactions, such as shared memory and synchronization. He worked as a research scientist in industry for 10 years, as an independent computer consultant for 12 years, and has taught computer science for 11 years. He is currently an assistant professor of computer science at the University of Minnesota. He is an ACM member.

DAVID J. LILJA received a Ph.D. and an M.S., both in electrical engineering, from the University of Illinois at Urbana–Champaign, and a B.S. in computer engineering from Iowa State University in Ames. He

is currently a faculty member in the Department of Electrical Engineering and the Director of Graduate Studies in Computer Engineering at the University of Minnesota in Minneapolis. Previously, he worked as a research assistant at the Center for Supercomputing Research and Development at the University of Illinois, and as a development engineer at Tandem Computers Incorporated in Cupertino, CA. His main research interests are in computer architecture, parallel processing, and high-performance computing. He is a senior member of the IEEE Computer Society, a member of the ACM, and is a registered professional engineer.

JOHN RIEDL has been a member of the faculty of the computer science department of the University of Minnesota since March 1990. His research interests include collaborative systems, distributed database systems, distributed operating systems, and multimedia. At the 1988 Data Engineering Conference, he and Dr. Bhargava received the Best Paper Award for their work on "A Model for Adaptable Systems for Transaction Processing." Riedl received a B.S. degree in mathematics from the University of Notre Dame, Notre Dame, IN in 1983 and the M.S. and Ph.D. degrees in computer science from Purdue University, West Lafayette, IN in 1985 and 1990, respectively.

JAMES R. ANDERSON received a bachelor's degree in electrical engineering from the University of Minnesota, Twin Cities, in 1995. He is currently a research assistant and MSEE candidate in computer engineering at Purdue University, West Lafayette. He has worked as an ASIC circuit designer for LSI Logic and is also a member of Tau Beta Pi and the IEEE Computer Society. His research interests include computer architecture, parallel processing, and low-power VLSI.

Received March 1, 1996; revised August 20, 1996; accepted September 10, 1996