Hardware Supported Synchronization Primitives for Clusters

Alberto F. De Souza[1], Sotério F. Souza[1], Cláudio Amorim[2], Priscila Lima[1] and Peter Rounce[3]

[1]Departamento de Informática, Universidade Federal do Espírito Santo, Vitória-ES, Brazil
[2]Programa de Engenharia de Sistemas e Computação – PESC/COPPE, Universidade Federal do Rio de Janeiro, Rio de Janeiro-RJ, Brazil
[3]Computer Science Department, University College London, London, United Kingdom

Abstract - *Parallel architectures with shared memory are well suited to many applications, provided that efficient shared memory access and process synchronization mechanisms are available. When the parallel machine is a cluster with physically distributed memory, software based synchronization mechanisms together with virtual memory infrastructure can implement Software Distributed Shared Memory (S-DSM), a shared memory abstraction on a distributed memory machine. However, the communication network overload from the emulation can limit the performance of such systems. This problem motivated our research, in which we developed a set of synchronization primitives for S-DSM on reconfigurable hardware. This hardware implements an auxiliary synchronization network, which works in parallel to the data communication network. Experiments evaluating our hardware implementation against a software one showed that our system increases the performance of these S-DSM primitives by a factor of 40 or more.*

Keywords: Clusters, Distributed Synchronization, Barriers, Locks.

1 Introduction

Parallel machines can have shared or distributed memory. Shared memory parallel machines are easier to program than distributed memory ones mainly because the data structures manipulated by parallel algorithms can be more easily accessed in the former. However, shared memory parallel machines are harder, and consequently costlier, to implement. But, it is possible to use software, the virtual memory infrastructure and the data communication network of distributed memory machines to emulate shared memory systems [1]. These shared memory systems are known as Software-Distributed Shared Memory systems (S-DSM). In S-DSM systems many processors may possess a copy of a piece of the shared memory (e.g. a virtual memory page). Consistency among the copies of these shared items is guaranteed only at synchronization points of the computation. Proper selection of synchronization points coupled with knowledge of the memory consistency model make programming S-DSM systems as easy as shared memory systems in most cases.

The implementation of efficient S-DSM systems demands fast synchronization mechanisms. However, in many of such systems, synchronization mechanisms exchange messages between processors through the conventional communication network in competition with other traffic. These messages also pass through the network protocol layers, further slowing synchro-nization. Observation of these deficiencies motivated us to investigate the hardware implementation of distributed synchronization mechanisms. Hence, we have developed a distributed synchronization system, using reconfi-gurable hardware, where the processors of a cluster use a separate auxiliary synchronization network (Figure 1).

To implement our distributed synchronization system we studied the synchronization mechanisms necessary for S-DSM, evaluating the importance and viability of a hardware implementation of a subset of these. Our evaluations led us to implement two synchronization mechanisms: barrier and lock. Both are synchronization primitives heavily used in parallel processing.

When associated with a group of processes, barrier has the property of not allowing any of them to proceed unless all members of the group have reached the barrier. Lock controls access to critical regions. Each lock has an associated token and, if a process needs to access the lock's critical region, it must first request and acquire the token, or lock. Access to a lock by processes is mutually exclusive, so a process requesting the lock must wait if the lock is already held by another process.

We have compared our hardware implemented barrier primitive with the same primitive implemented in software using MPI [2]. Our barrier was 40 times faster than the MPI_Barrier primitive. Our lock operated in a similar time to our barrier primitive.

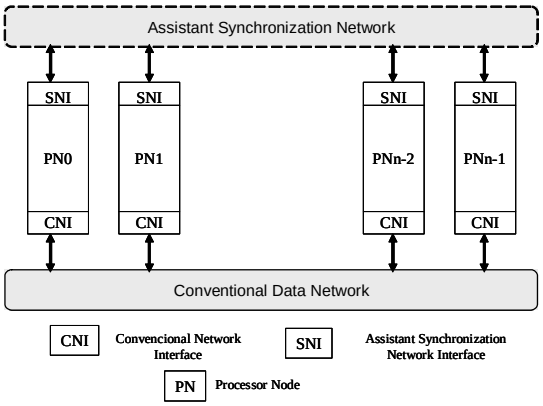

Figure 1: Distributed synchronization system

2 Related Work

Two works in the literature present results relevant to our work: the Dietz et al. TTL_PAPERS [3] and the Hayakawa and Sekiguchi Synchronization and Communication Controller (SCC) [4].

TTL_PAPERS [3] is an implementation in TTL hardware of Purdue's Adapter for Parallel Execution and Rapid Synchronization (PAPERS [5]). This hardware makes use of the parallel ports of a cluster's computers in conjunction with some simple external hardware for the implementation of a barrier. A single barrier is made available and all processors of the cluster are members of a single group, so that they synchronise at this one barrier. The hardware described does not implement locks.

The shorter time for barrier synchronization in TTL_PAPERS is 2.5µs for a 4 processor cluster, and it is estimated that this would not change with the number of PEs. However, the design of the external hardware changes and its complexity increases with PE number, and an extendable system might well have barrier synchronization times that increase with PE count. Our hardware obtained barrier synchronization times of 1.686µs on average, which would increase logarithmically with PE count. We cannot perfectly compare these times with TTL_PAPERS, as its experiments use processors with lower frequencies than those in ours.

The SCC [4] implements barriers for clusters using specialised PCI boards that are inter-connected in a one-dimensional daisy-chain by 40-way cables such that messages from one end to the other traverse all the nodes. The "sync-comm" network produced by the boards provides a dedicated synchronization network separate from the conventional network, as does our auxiliary network. As with the TTL_PAPERS, just a single barrier is available, but a sub-set of the processors can be selected to participate in the barrier, and again lock was not provided. Barrier synchronization times in SCC were 3.2µs and 6.2µs for two and four processor clusters respectively, whereas experiments on our hardware with 2-processor clusters gave results inferior than 2µs on average.

3 Hardware synchronization mechanism

Our auxiliary synchronization network consists of a hierarchical network (Figure 2) that operates through messages sent by processors via 2 network elements: Nodes at the bottom associated with the processors, and Synchronizers in the upper levels.

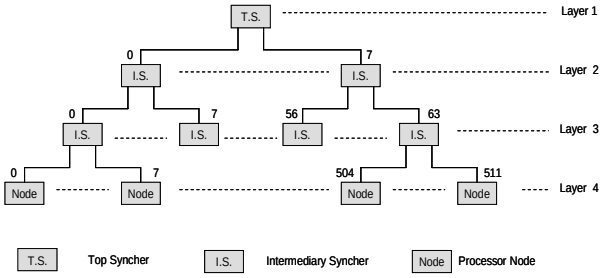

Figure 2: Topology of the auxiliary network

Our requirements for synchronization were for this auxiliary network to carry 4 types of messages: two for the barrier and two for the lock. We augmented these with two further messages: *reset* and *broadcast*. The *reset* message initializes the Global Clock, an extra facility provided by the synchronization network [6], while *broadcast* provides a general-purpose message distribution capability. The auxiliary network is designed on the premise that downward messages are broadcast to all lower level elements – the Intermediate Synchronizers (IS) retransmit all such messages to all lower level elements such that they are received synchronously. Barriers are implemented by masks distributed across the Synchronizers. For a particular barrier (there may be more than one), a synchronizer holds a mask, the bits of which indicate which of its sub-trees participate in the barrier: for the synchronizers in the level above the Nodes, the mask bits indicate which of its attached Nodes are in the barrier. Figure 3 illustrates the general format of the messages.

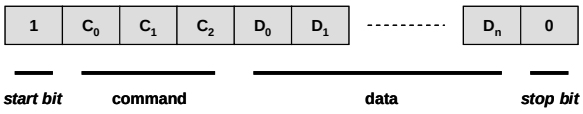

Figure 3: General format of a message

A message is detected through a start bit – a transition from inactivity (logic 0). After this bit, three command bits are sent containing the message code, which indicate the operations to be carried out. A number of data bits, varying from zero to 76, follow, and a message terminates with the control stop-bit, forcing the line to inactive. Of the 8 commands available, only 6 are used with the rest reserved for future use. The six commands, or message types, are detailed below:

reset. Initializes the Global Clock. This command is passed up in the hierarchy of Synchronizers until it reaches the top. Then, it goes down initializing all the Node Global

Clock counters by synchronously arriving at each of the Nodes [6]. A *reset* message has no data bits.

barrier. Creates a barrier. This message traverses the network to the top, configuring a barrier. All Nodes participating in the barrier send a barrier creation message, though not all reach the top. Synchronizers that have already come in contact with the barrier message do not propagate it up. Barrier mask information is propagated as data via these messages to the Synchronizers indicating who participates of the barrier.

barrier reaching. Informs the Synchronizer that a particular barrier has been reached. When all Nodes or Synchronizers connected to a particular Synchronizer have indicated that they have reached a barrier, this Synchronizer informs the one above it of this fact, unless it is the Top Synchronizer, when the barrier is considered to be completed. At this point, the Top Synchronizer sends a message of the same kind down the hierarchy to signal that the barrier has completed. Barrier masks in the Synchronizers have their bits cleared as this message propagate downward.

broadcast. Disseminates a message with a 76-bit data field throughout the network. The data content is user defined. A *broadcast* message first goes up the hierarchy and then goes down to reach all Nodes.

lock acquire. Requests a lock. A lock request traverses the network to the top. If the Top Synchronizer verifies that the lock is free, it sends a *lock acquire* message down the tree, giving all Nodes the ID of the lock's acquiring Node, otherwise the message is discarded.

lock release. Frees a lock. The message traverses the network to the top before being broadcast down, informing all Nodes that the lock has been freed.

3.1 The Top Synchronizer

The Top Synchronizer (TS), see Figure 4, is unique, occupying the root of the auxiliary network tree. It is responsible for the control of creation and liberation of barriers, and for the granting of all locks. The TS has eight incoming and outgoing serial ports for communication with the next lower level of the network. At the top right of Figure 4 is shown the 64 MHz Pulse Generator that synchronizes the auxiliary network and increments the Global Clock.

Receiver Down has an 8-entry store for incoming messages on its 8 incoming serial links. It signals the control core with the identity of the incoming port on which a message has been received. The control core reads the message via the internal bus, identifies the type and acts accordingly. Messages of type *reset* and *broadcast* are sent on all outgoing serial ports without any local action. Messages of type *barrier* cause the synchronizer to create the barrier. On receipt of the first message about the creation of a specific barrier, the barrier memory is updated; subsequent *barrier* messages for this same barrier are discarded. With messages of type *barrier reaching*, the TS records, in the Barriers Memory, the receipt of each message for a particular barrier, until it has received such messages from all sub-trees involved in the barrier, at which point it updates the Barriers Memory and broadcasts a *barrier reaching* message downwards. Messages of type *lock acquire* request the acquisition of a specific lock. If the lock is available, the Locks Memory is updated with the identifier of the Node requesting the lock and a *lock acquire* message is broadcast down the network, signalling the acquisition of the lock and the acquiring processor ID. If the lock is unavailable, the message is discarded. Unsatisfied requests cause requesting processors to wait until a *lock release* for this requested lock is received. After that, the lock request can be re-tried. Messages of type *lock release* update the Locks Memory and generate a *lock release* broadcast signalling the availability of the lock.

Transmitter Down has only one message store, since each message is always sent out on all 8 outgoing serial links to the lower hierarchical levels.

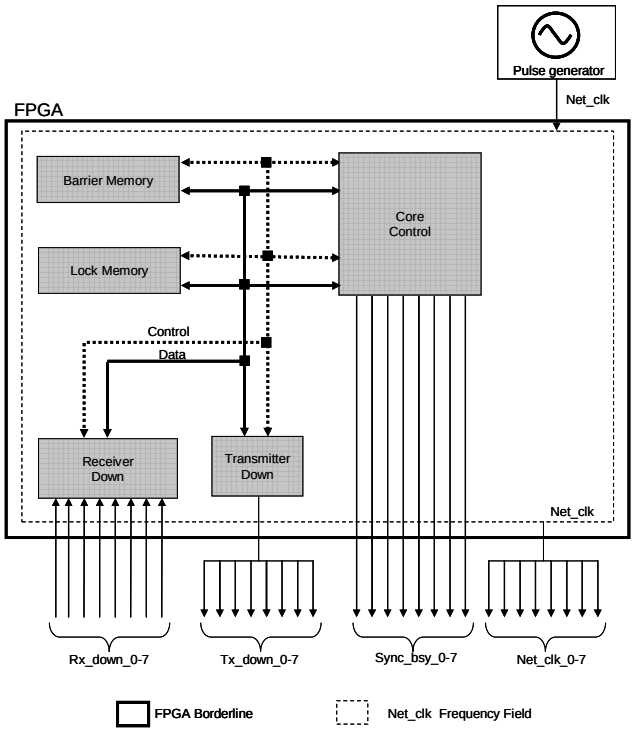

Figure 4: The Top Synchronizer

3.2 The Intermediate Synchronizer

The Intermediate Synchronizer (IS), which implements the interior nodes of the network tree, are shown in Figure 5. The Receiver Down and Transmitter Down are the same as in the TS: concentrating incoming messages from lower levels, and broadcasting outgoing messages on all links. Transmitter Up and Receiver Up units handle single message transfers with the immediately superior Synchronizer. Receiver Up makes received messages available over the internal data bus to other elements as soon as the data of the message is available. The control core executes according to the message's command. Downward messages are forwarded via Transmitter Down to all lower level elements immediately in

order to provide for synchronous receipt across the network in as few cycles as possible. The only internal processing is for downward messages of type ***barrier reaching***, when the IS barrier memory is updated, freeing the barrier locally.

For upward propagating messages of types *reset*, *broadcast*, *lock acquire* and *lock release*, the control core just commands their retransmission via Transmitter Up. Messages of type *barrier* require that the Barriers Memory is accessed to verify if the specified barrier has already been created: if so, the control core discards the message; if not, the barrier is created locally and the message is sent upwards, discarding the barrier mask relative to the current level. A barrier creation message has its size reduced by 8 bits each time it is forwarded. Messages of type *barrier reaching* update the Barriers Memory by clearing the bit in the barrier mask for the sub-tree from which the message arrived. If this is the last of the mask bits to be *zeroed*, the message is sent up the network; if not, the message is not forwarded, but discarded.

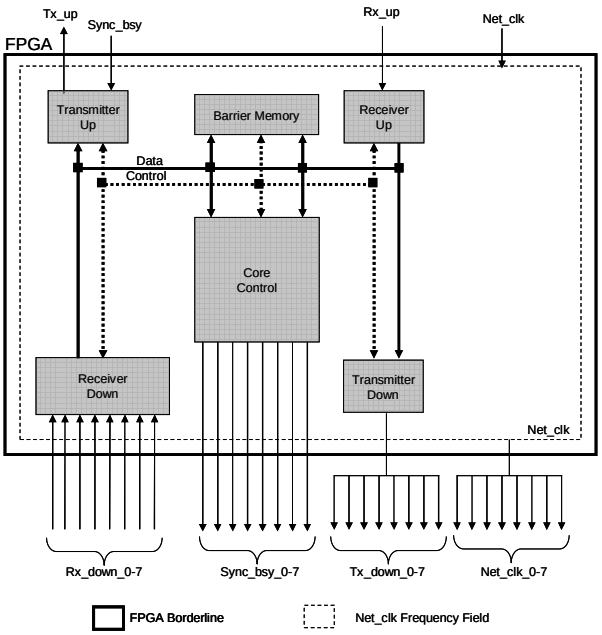

Figure 5: The Intermediate Synchronizer

3.3 The Node

The Node, structure shown in Figure 6, consists of a PCI board installed inside each computer to implement the lowest level of the network. Memory within the board is mapped to processor addresses over the PCI bus [7]. To send a message, the Node processor (NP) writes it to a specific address in the PCI bus interface. To discover if a barrier has been reached, the NP reads from a set of 256 addresses, each one corresponding to a barrier. Lock status (free / not free) is read from another set of 256 addresses, each one corresponding to a lock. Broadcast messages are read from a range of 1024 addresses that form a circular list, each address corresponding to a specific broadcast message.

Implementation of the Global Clock requires a 64-bit counter in each Node [6], which is incremented at each positive transition of the clock signal – Net_clk –, coming from the Top Synchronizer. The network design has been implemented to guarantee that all Nodes receive the Global Clock and any *reset* message at the same instant. Therefore, it can be assured that the counters of Nodes are initialized and/or incremented in a synchronous manner, holding the same counting (time) value at each clock cycle.

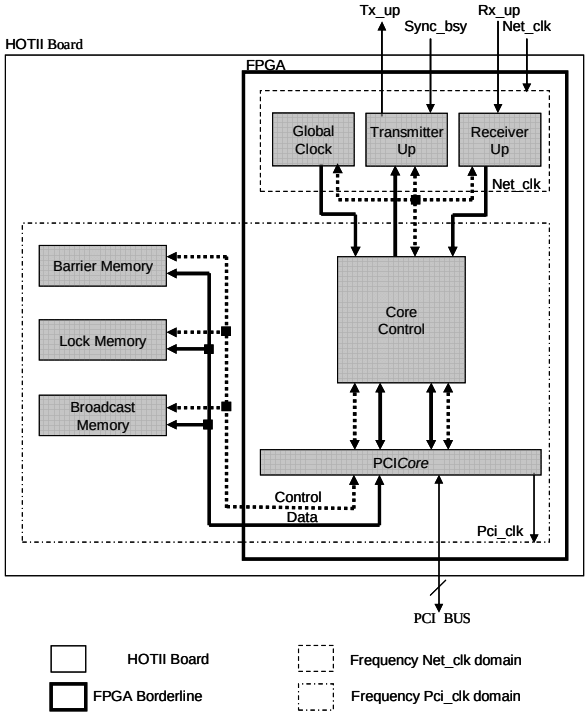

Figure 6: Blocks diagram of Node

Receiver Up and Transmitter Up are essentially the same as in the IS. The control core is the element responsible for treating the messages deriving from the PCI bus and Receiver Up. All messages, without exception, coming from the PCI bus are transmitted upwards. All messages received from the network via the Receiver Up require action from the control core.

Of the messages originated at the PCI bus, the *barrier* ones require that the Barriers Memory of the Node be updated, creating the barrier. When a downward *reset* message is received, the Global Clock counter is reinitialized. For downward messages, *barrier reaching and lock release*, the Barriers and Locks Memory are updated respectively. Downward *broadcast* messages are stored in the Broadcast Memory, along with the lower-order 48 bits of the Global Clock, which record their reception time. Downward *lock acquire* messages update the Locks Memory with the identifier of the acquiring Node.

4 Methods

To perform validation and evaluation tests of the system developed, we implemented the hardware in H.O.T. II FPGA development boards [8]. Such a board consists of the basic

operational circuitry, memory banks and a model XC4062XLA–HQ240 FPGA, made by Xilinx [9]. The FPGAs development tool chosen was also from Xilinx. The experimental system, shown in Figure 7, has a 2-level hierarchy with just 2 Nodes and a Top Synchronizer, but no Intermediate Synchronizers. Each element is a microcomputer with attached H.O.T. II board, programmed to be either a Node or a TS. Category 5, Shielded Twisted Pair (STP) cables of 5m length with 4-pairs are used to connect the elements.

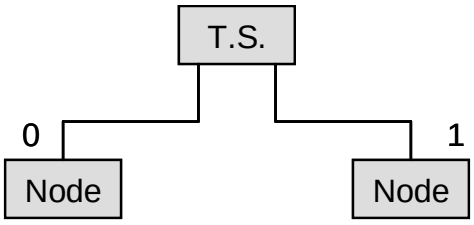

Figure 7: Diagram of the environment for experiments

The machine hosting the TS was used solely to load the FPGA with the logic for the TS and to provide power to the H.O.T. II board. The machines hosting the Nodes, besides the H.O.T. II boards, had 100Base-T Ethernet network interfaces to inter-connect them via a switch to create our experimental cluster. Communication with the H.O.T. II board and uploading of the Node hardware program into the FPGA was thorough a Linux driver module developed by the researcher teams of DI/UFES and COPPE/UFRJ. The module is loaded at operating system initialization. This module issues the mapping of the PCI bus to memory positions, enabling communication with the FPGA and other board components, such as the memories and ports.

5 Test programs

To validate the hardware and to evaluate the auxiliary network, we implemented a series of small C programs. These programs were not intended to perform any productive computation but were designed to exercise key characteristics of the synchronization network.

We developed a program that executes a parallel computation that uses synchronization via barriers, using the Message Passing Interface (MPI) mechanism as software support [2]. Figure 8 contains the key elements of that program, which implements a barrier in a conventional way by software. The bolded line of code implements the barrier.

The code of Figure 8 was then slightly modified to create the code of Figure 9, where 256 barriers are implemented by hardware (see bolded lines in Figure 9). This code uses software modules that rely on the auxiliary network to provide synchronization support.

Validation and evaluation of the hardware lock required the development of test programs in a similar fashion to that of the barrier implementation. Figure 10 shows the code that contains the call to a hardware implemented lock.

```
unsigned int j = atoi(argv[1]) ;
MPI_Init(&argc, &argv) ;
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank) ;
MPI_Comm_size(MPI_COMM_WORLD, &num_procs) ;
MPI_Barrier(MPI_COMM_WORLD) ;
unsigned long start_all = SyncGetTsc() ;
for (k=0;k<j;k++) {
  for (b=0;b<256;b++) {
    start = SyncGetTsc() ;
    MPI_Barrier(MPI_COMM_WORLD) ;
    stop = SyncGetTsc() ;
    … /* Computation of the average barrier time */
  }
}
unsigned long end_all = SyncGetTsc() ;
MPI_Finalize() ;
```

Figure 8: Barrier implemented in software

```
for (k=0;k<j;k++) {
  for (barrier=0; barrier <256;b++) {
    SyncCreateBarrier(barrier, mask);
    start = SyncGetTsc() ;
    SyncReachBarrier(barrier) ;
    SyncVerrifyBarrier(barrier) ;
    stop = SyncGetTsc() ;
      … /* Computation of the average barrier time */
  }
}
```

Figure 9: Barrier implemented in hardware

The test programs developed were instrumented so as to enable us to measure lock and barrier times. The code instrumentation instruction used was *read-time stamp counter – rdtsc* [10] – via the call *SyncGetTsc().* The programs were compiled by MPI compiler (mpicc), and run on the two processors configured as Nodes.

```
    for (k=0;k<j;k++) {
      start = SyncGetTsc() ;
      do {
          SyncGetLock(lock) ;
          l = SyncVerifyLock(lock) ;
       } while (l != proc);
       stop = SyncGetTsc() ;
       SyncFreeLock(lock) ;
       … /* Computation of the average lock time */
    }
```

Figure 10: Lock implemented in hardware

Evaluation was based on the average of the barriers and locks times, on the total execution times of the test programs, and on the speedup obtained by the hardware implementation of barriers in comparison to their software implementation. We obtained the duration of a barrier from two timestamp readings via *SycGetTsc* calls placed in defined points of the test program. The duration of lock is obtained in similar manner, according to Figure 10. Likewise, total execution times of a program were acquired. Having gathered all this information, these times are computed from the timestamps by equation 4.1:

$$time_{total} = (t_{final} - t_{initial}) / f_p \qquad (4.1)$$

where $\boldsymbol{time_{total}}$ is the required time, $\boldsymbol{t_{initial}}$ and $\boldsymbol{t_{final}}$ are the first and second timestamps read by *SyncGetTsc* calls, and $\boldsymbol{f_p}$ is the frequency of the Global Clock.

The speedup indicates in which proportion a system is faster than another. In this work, we computed the speedup according to equation 4.2:

$$speedup = t_{soft} / t_{hard} \qquad (4.2)$$

where $\boldsymbol{t_{soft}}$ is the time taken by the software implementation to execute a lock or barrier and $\boldsymbol{t_{hard}}$ is the time taken by the hardware one.

6 Experiments

We performed a series of tests, results presented in Table 1, on the program that implements barriers in software, and another set, results presented in Table 2, on the program that implements barriers in hardware. Each series consisted of running a program six times, exponentially varying in powers of ten the number of barriers created. We measured the barriers time and the total execution time of each program running each test.

Table 1: Barrier implementation in software.

Number of barriers (un)	Average (µs)	Total execution time of each program (s)
256	106.064	0.041130420
2,560	95.284	0.294931485
25,600	87.773	2.678029310
256,000	88.905	27.129206158
2,560,000	89.008	271.460952398
25,600,000	87.074	2,658.359010029

Table 2: Barrier implementation in hardware.

Number of barriers (un)	Average (µs)	Total execution time of each program (s)
256	1.713	0.001969581
2,560	1.906	0.019094939
25,600	1.656	0.191182427
256,000	2.059	1.942289606
2,560,000	1.763	19.861682971
25,600,000	1.852	199.708281373

As tables 1 and 2 show, the mean times for hardware implemented barriers are smaller than for software implemented ones. It can also be seen that the total execution times of the corresponding programs indicate that those with hardware barrier implementation performed considerably better (as the frequency of the real-time clock of the Node processors is 1.532939GHz, the precision on the time is sub-nanosecond).

The chart of Figure 11 displays the speedup between the two kinds of barrier implementation. The horizontal dimension represents the number of barriers in the execution of a test program. The vertical dimension is computed by the division of the *Average* column of Table 1 (mean time of barrier in software) by the corresponding column of Table 2 (mean time of barrier in hardware). These values show that a barrier that has been implemented in hardware is, on average, around 50 times faster than one that has been implemented in software.

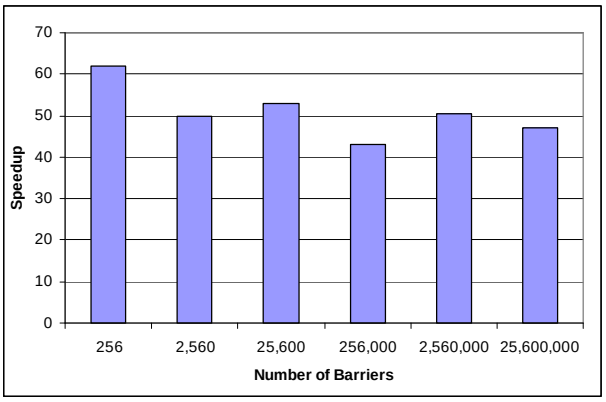

Figure 11: Speedup for hardware barriers

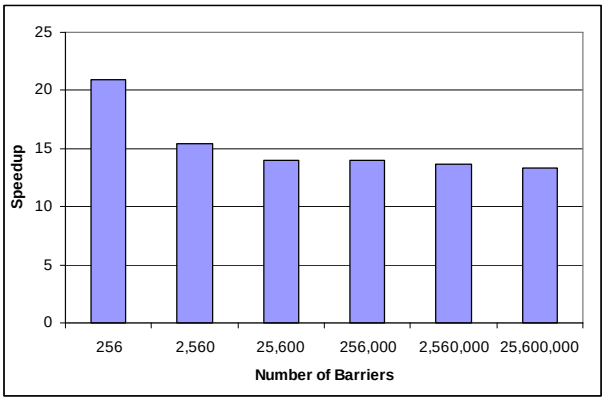

Figure 12: Speedup of total execution time

Figure 12 presents the process execution time speedup for the hardware implementation compared to the software one. As the barrier count reaches the value of 256,000, the hardware-based program has an execution time 13 times smaller than that of the software-based one, while for lower barrier counts, around hundreds of barriers, this ratio is around 20.

Table 3: Lock implementation in hardware.

Number of locks (un)	Average (µs)	Total execution time of each program (s)
256	1.147	0.002349573
2,560	1.163	0.023554924
25,600	1.169	0.237558289
256,000	1.169	2.380268197
2,560,000	1.165	23.941582051

Lock performance tests were performed in a similar way to the barrier tests with the program of Figure 10 executed six times, exponentially varying the numbers of locks in powers of 10. Each run measured the time of each lock and the total execution time of the program. Table 3 shows the results. The mean times of hardware implemented locks are very close to those for barriers. This is expected, as message processing and transmission through the auxiliary network is similar for both barriers and locks.

7 Conclusions

Our experiments validate the proposed synchronization design which we believe provides an improved solution over similar hardware solutions in the literature for the following reasons. The design has better features: two synchronization primitives – barrier and lock; a message transmission infra-structure; and a Global Clock. Moreover, when these mechanisms are used with other network resources, our system becomes a unique tool for the debugging of parallel programs [11]. The tree-structured network architecture makes the design readily extendable with a logarithmic increase in layers with processing nodes and with synchronization overheads scaling at a similar logarithmic rate. The performance is similar if not yet provably better than other designs, while we consider the implementation costs to be low.

The results obtained from our experiments confirm that barriers implemented in hardware perform 40 times faster than barriers implemented in software with MPI. The lock times that were measured are consistent with the barrier times, inducing the belief that the performance of the hardware implementation of this primitive would also be much superior to an implementation in software.

As future work, we intend to perform experiments for the performance evaluation our auxiliary network as the synchronization infrastructure needed for a distributed shared memory environment. Cilk [12] is a distributed shared-memory environment that we envisage as adequate to evaluate barriers and locks in this context.

8 References

[1] C. Amza et al. TreadMarks: Shared Memory Computing on Networks of Workstations. IEEE Computer, February 1996, p. 18-28.

[2] MPI: A Message-Passing Interface Standard. Massage Passing Interface Forum, V1.1, June 1995. http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html.

[3] H. G. Dietz, R. Hoare, and T. Mattox. A fine-grain parallel architecture based on barrier synchronization. International Conference on Parallel Processing, 1996, pp. 247-250.

[4] K. Hayakawa, S. Sekiguchi. Design and Implementation of a Synchronization and Communication Controller for Cluster Computing Systems. 4º Int. Conference on High Performance Computing in Ásia-Pacific Region, Vol. 1, May 2000, pp. 76-81.

[5] T. Muhammad. Hardware barrier synchronization for a cluster of personal computers. M.Sc. Dissertation. Purdue University, May 1995.

[6] C. L. Amorim, A. F. De Souza. Distributed global clock for clusters of computers. United States Patent No. 20060212738, 21/09/2006 .

[7] PCISIG. PCI Local Bus Specification – Revision 2.2. December 1998.

[8] Virtual Computer Corporation. H.O.T. II – Hardware API Guide. Version 2.3. Virtual Computer Corporation, 1999a.

[9] Xilinx. XC4000XLA/XV Filed Programmable Gate Arrays. Version 1.3, Xilinx Product Specification, October 1999.

[10] Intel. Pentium II Processor: Using the RDTSC Instruction for Performance Monitoring. Application Notes. Intel, January 1998.

[11] W. Meira Jr., T. LeBlanc, A. Poulos. Waiting time analysis and performance visualization in carnival. ACM SIGMETRICS Symposium on Parallel and Distributed Tools, May 1996.

[12] R. M. da Silva, L. Whately, Lauro, M. C. S. de Castro, C. Bentes, C. L. Amorim. Runtime Support for Running Applications with Dynamic and Asynchronous Task Parallelism in Software DSM Systems. Proceedings of the 17th Simposium on Computer Architecture and High Performance Computing. Los Alamitos CA/USA : IEEE Computer Society, 2006. v. 1. p. 1-10