Fast Synchronization of Large Multiprocessors Using Wired-NOR Barriers and Counting Semaphores [1]

Kai Hwang and Shisheng Shang
Department of Electrical Engineering–Systems
University of Southern California
Los Angeles, CA 90089–0781

Abstract: A new wired-NOR snoopy mechanism is developed, which can be used along with counting semaphores, for fast barrier synchronization in a large shared-memory multiprocessor. Through critical timing analysis, we prove that 525 processors can be synchronized in 540 ns using the hardware barrier. The barrier wires result in a factor of $O(10^3)$ reduction in synchronization time, as compared with software barrier implementation. The wired-NOR synchronization is capable of supporting *Fork* and *Join* constructs at run time without prior knowledge of the synchronization pattern. The scheme supports also divided and partially-ordered synchronization patterns, if the barrier patterns are known at post compile time. In general, the more snoopy lines used, the higher degree of multiprogramming it can support. Counting semaphores are coupled with limited barrier wires to further increase the application potential. Performance results obtained from simulation experiments verified the effectiveness of the hardware operations. This snoopy synchronization mechanism supports massive parallelism in very large, multiprogrammed, shared-memory multiprocessors.

Index Term: Barrier synchronization, counting semaphores, multiprogramming, operating systems, parallel processing, shared-memory multiprocessors.

---

[1]Manuscript submitted to *IEEE Transactions on Parallel and Distributed Systems*, April 1991. This research was supported by NSF Grant No. 89-04172. For all future correspondence, contact Prof. Kai Hwang via Email kaihwang@panda.usc.edu, or FAX (213) 740-4449, or mail to Dept. of EE-Systems, USC, Los Angeles, CA 90089-0781, USA.

Fast Synchronization of Large Multiprocessors Using Wired-NOR Barriers and Counting Semaphores[1]

Kai Hwang and Shisheng Shang

CENG Technical Report: 91-06

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, CA 90089-2562 (213)740-4470
kaihwang@panda.usc.edu; sshang@panda.usc.edu

April 1, 1991

[1]This research was supported by NSF Grant 89-04172 to the University of Southern California.

1 Introduction

Partitioning and replication of programs are two approaches to parallel programming on an MIMD computer system. Program partitioning demands multiple processes to use the same data sets in the shared memory. Program replication is often done in a multicomputer using distributed memory. To support massive parallelism, a shared-memory multiprocessor demands a fast hardware mechanism for interprocessor synchronization. Reducing synchronization overhead will greatly enhance the performance of a multiprocessor [11]. Most barrier synchronization mechanisms are realized in software [1, 3, 4, 9, 10, 22]. Some systems use combining networks [8, 13, 21]. Some other systems implement barriers in cache coherence hardware protocols [7, 9]. Recently, O'Keefe and Dietz [16, 17], proposed the use of centralized hardware and associative memory in synchronizing MIMD multiprocessors.

The software approach requires intolerable time to synchronize a large multiprocessor, but it is more flexible to use. Intuitively, the worst time to cross a software barrier increases quadratically with respect to the number of processes to be synchronized. O'Keefe and Dietz's method works only in situations, in which the barrier patterns are predictable at compile time. The barrier synchronization scheme proposed in [6] applies only in a nonpreemptive, multiprogrammed environment. In both approaches, the number of synchronizable processes is upper bounded by the number of processors allocated. This inevitably limits the degree of multiprogramming in a multiprocessor system.

Removing the above operational constraints, we introduce a new hardware approach to achieving fast barrier synchronization in a large system with $O(10^2)$ to $O(10^3)$ processors. The idea of using wired-NOR was inspired by the pulldown bus concept [14]. We have presented the original idea in a conference paper [12]. This journal version is significantly extended from the conference presentation. We consider a multiprocessor system using a modified *run-to-completion* scheduling policy which allows preemption. This is extended from the work of Zahorjan and McCann [24], which is nonpreemptive. When there are two or more processes created at the same processor, the processes are scheduled locally in a round-robin fashion. In other words, process preemption is allowed, but process migration is still

prohibited. We first describe the wired-NOR design of snoopy lines, which take advantage of the broadcast capability of a pull-down bus. The scalability issues are discussed with a critical timing analysis.

Then we demonstrate how to use the proposed hardware to accomplish fast barrier synchronization. A combined hardware and software method is shown to achieve wired barrier synchronization assisted by counting semaphores in the shared memory. The wired-NOR hardware is efficient to synchronize any subset of concurrent processes, provided the barrier patterns are predictable at compile time. We reveal the effect of line width on the degree of multiprogramming. Finally, three synthesized workloads are used to simulate the performance of the wired-NOR barrier synchronization in a multiprogrammed multiprocessor. New illustrative examples, pseudo codes for parallel primitives, support of dynamic synchronization, timing analysis between hardware and software synchronizations, and simulated workload studies are all new results extended from the ICPP paper.

2 Wired-NOR Synchronization Hardware

We propose to use a small number of m snoopy lines across a large number of n processors, as shown in Fig. 1. We use a wired-NOR, pull-down design. The broadcast capability of this design reduces the hardware complexity significantly as compared with existing designs [5]. Each processor uses a *control vector* $X = (X_0, X_1, \cdots, X_{m-1})$ and a *monitor vector* $Y = (Y_0, Y_1, \cdots, Y_{m-1})$ for distributed synchronization control. The X and Y vectors are mapped into the shared memory. Thus, they are accessible by all processors. Using distributed control, this scheme has the advantage of reduced interprocessor traffic for synchronization purpose.

Each control vector X represents the location of a *set/reset circuit*, which affects voltage levels on m snoopy lines. Each X_i bit is the input to an NPN bipolar transistor. The monitor vector Y represents the location of a *probing circuit* for sensoring the voltages on all lines; i.e. the Y_i bit checks the output of the transistor tied to the i^{th} line. The vectors X and Y are program accessible from each processor. The control vector X can be read and

written, but the monitor vector Y is read only.

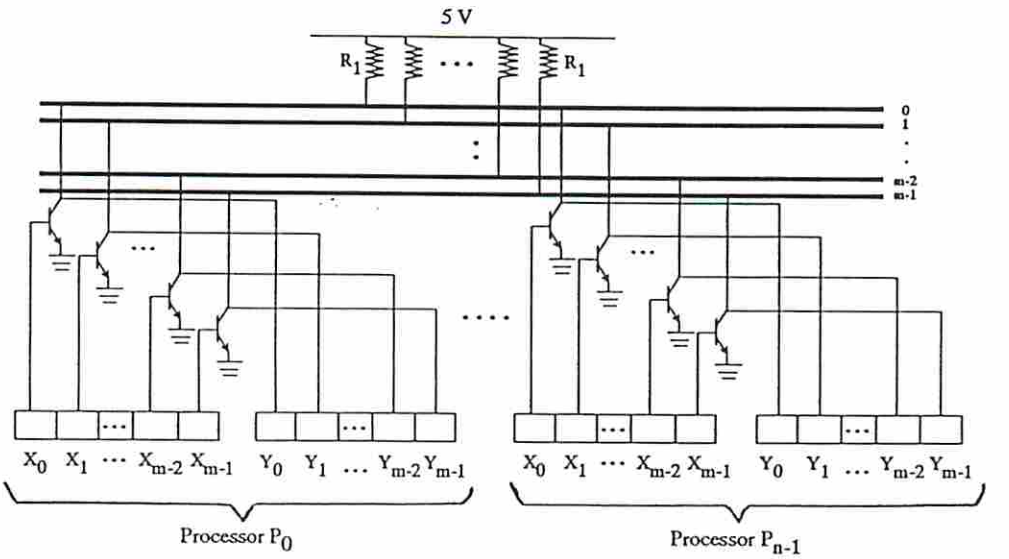


Figure 1: The hardware design of a concurrency control bus with m snoopy lines for an n-processor system. (P_i: processor i, X_i: the i^{th} bit in control vector X, and Y_i: the i^{th} bit in monitor vector Y)

Each line is wired-NOR to n bipolar transistors, which are tied to n probing circuits. For example, when any X_i bit is high, the transistor is closed and it pulls down the current. The voltage level of line i ($0 \leq i \leq m-1$) becomes low. When all X_i are low, all transistors connected to line i are open. Thus, the voltage level of line i becomes high. We will describe in Section 4 how to use these pull-down lines to accomplish fast barrier synchronization.

Next, we want to decide the number, m, of lines required, given the following system parameters in a multiprogrammed multiprocessor:

- k The degree of multiprogramming supported.
- ℓ_i The number of processes created in program i.
- P_i The number of processors allocated to program i.
- b_i The number of active synchronization points demanded in program i.
- S_i The number of snoopy lines required for program i.

Note that $\ell_i \geq P_i$ for $i = 1, 2, \cdots, k$, and $\sum_{j=1}^{k} P_j \leq n$, where n is the total number of processors in the system. An active synchronization point is initialized but has not reached the rendezvous point of a program. The value of S_i depends on the value of b_i and how many lines are allocated to each synchronization point, when ℓ_i processes are concurrently executed on P_i processors. Based on a line allocation policy, we estimate that

$$S_i = b_i \lceil \ell_i / P_i \rceil + 1 \tag{1}$$

Let S be the total number of required snoopy lines. Obviously, we have $S = \sum_{i=1}^{k} S_i$ as a lower bound on m. Thus, the degree of multiprogramming is limited by the number of snoopy lines that a system has and by characteristics of the application programs run on the system. For an example, an 8-processor system supports the interleaved execution of $k = 4$ programs. Assume $\ell_1 = 8$, $\ell_2 = 16$, $\ell_3 = 16$, $\ell_4 = 4$; $P_1 = 1$, $P_2 = 2$, $P_3 = 4$, $P_4 = 1$; $b_1 = 1$, $b_2 = 2$, $b_3 = 4$, and $b_4 = 1$. Thus, $S_1 = 9$, $S_2 = 17$, $S_3 = 17$, and $S_4 = 5$. Therefore, we need to use $m = 9 + 17 + 17 + 5 = 48$ snoopy lines. In general, the more lines the system has, the more synchronization points it can support among multiple programs. The wired-NOR lines can be built into a dedicated concurrency control bus on the backplane of a multiprocessor cluster. For a large system, the control lines may have to run across multiple backplanes using high-speed cables.

3 Scalability and Timing Analysis

Two hardware design issues are addressed first: (1) how many monitory transistors (gates) can be connected to a wire? and (2) how fast the wire will respond to a synchronization request? To answer the first question, we have to consider the DC characteristics of the circuit at logic 0 or 1 levels separately. Using the TTL family, the equivalent circuit of a snoopy line at logic 0 is shown in Fig. 2(a), where Q_1 is an NPN transistor, Q_2 and R_3 form the input stage of a probing circuit, R_1 is a pullup resistor, and R_2 corresponds to the output stage of a control circuit.

In the worst case, there is only one transistor pulling down the snoopy line. The total

pulldown current should not exceed the maximum current allowed in an NPN transistor. The pulldown current includes the currents from the pullup circuit and from n probing circuits. The Q_1 transistor must operate in saturation to lock the line in a logic 0 level. In this case, Q_2 is also saturated.

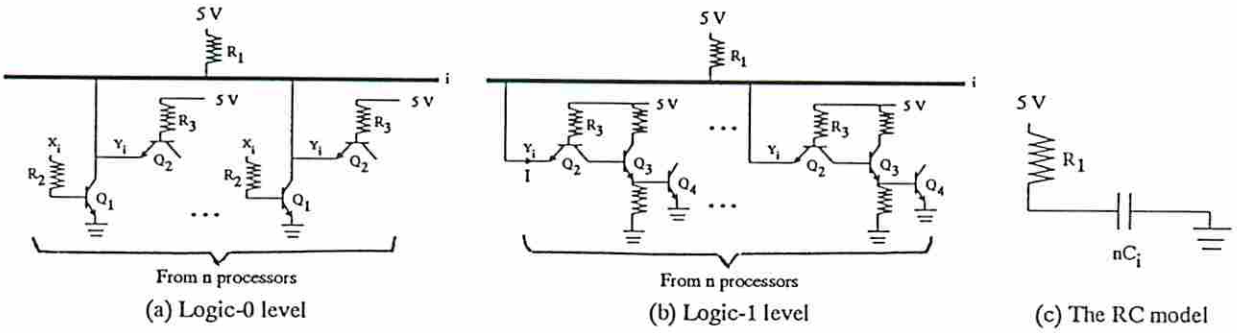


Figure 2: The equivalent circuit of a snoopy line i. ($0 \leq i \leq m-1$, C_i: input capacitance of a Q_2)

The *delay time* t_d of a snoopy line consists of the transistor *switching time* (t_s) and the *rise time* (t_r). The t_s equals the sum of the delay time and rise time of an NPN transistor. The rise time is modeled as an RC circuit, shown in Fig. 2(c). A DC analysis suggests $t_r = 2.2RC = 2.2nR_1C_i$. Therefore, we have:

$$t_d = t_s + 2.2nR_1C_i \tag{2}$$

Typical parameters used in the wired-NOR design are shown in Table 1, based on TTL devices [23]. From Eq. 2, the delay time is written as $t_d = 35 + 55nR_1$, where R_1 is expressed in kΩ.

Table 1: Typical values of electrical parameters used in the synchronization bus design with TTL transistors.

$I_{C_1(max)}$	800 mA	h_{FEI}	0.5
$V_{BE(sat)}$	0.8 V	R_2	100 Ω
$V_{BC(inv)}$	0.7 V	R_3	4 kΩ
$V_{CE(sat)}$	0.2 V	t_s	35 ns
$V_{EC(inv)}$	0.3 V	C_i	25 pF
h_{FE}	100		

The delay time is thus upper bounded by $35 + 55 \times 9.19 = 540$ ns. To study the scalability of the proposed scheme, we have to determine the maximum number of processors, that can be connected to each line and its delay time. Table 2 lists the size, n, of a multiprocessor system supported by different values of R_1. At the best, 525 processors can be attached to a single wire when $R_1 = 0.0175$ kΩ. There is a tradeoff between the delay time and the power consumption. The shorter the delay time (for smaller R_1), the more the power it will consume.

Table 2: The size n of a multiprocessor by choosing different values of R_1 and the corresponding synchronization delay t_d.

R_1 (kΩ)	n	t_d (ns)
0.01	320	211
0.0175	525	540
0.02	459	540
0.03	306	540
0.04	229	539
0.05	183	538
0.10	91	536
0.50	18	530
1.00	9	530

The wired design using standard TTL logic can be further scaled up to support large systems with more than 500 processors. The delay time and the number of processors for different values of R_1 are illustrated in Fig. 3. For a small shared-memory multiprocessor with less than 30 processors, the wired synchronization can be done in 52 ns, which is faster than any existing hardware scheme implemented. Using higher power bipolar transistors such as those specified in [15], one can further reduce the delay time to be able to synchronize more than one thousand of processors in a single system.

We compare the performance of wired barrier synchronization with the *test-and-set* software implementation in a shared memory. In a software synchronization, each processor synchronizes by acquiring a lock, updating a barrier counter, releasing a lock, and reading the barrier counter until the barrier is crossed. Let t_{lock} be the time to acquire and release

a lock, and t_{mem} be the access time to read a barrier counter in shared memory. Consider p processors reaching the barrier at the same time. The time to cross the barrier is thus estimated as $T_{SW} = pt_{lock} + pt_{mem} + \frac{p(p-1)}{2}t_{mem} = pt_{lock} + \frac{p(p+1)}{2}t_{mem}$. The first term corresponds to the time for p processors to enter the critical section and modify the counter, the second term is the time for p processors to read a counter, and the last term represents the time wasted in spinning by all processors.

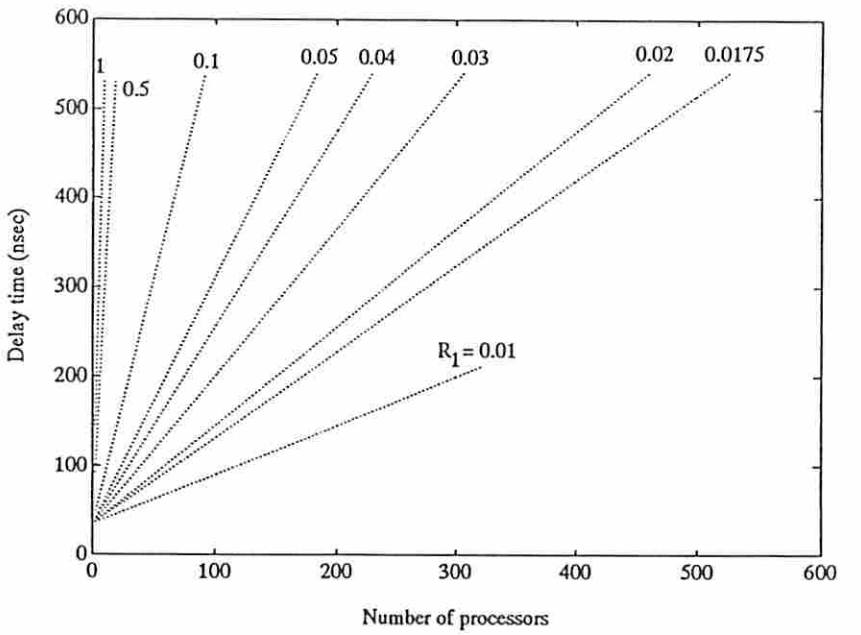
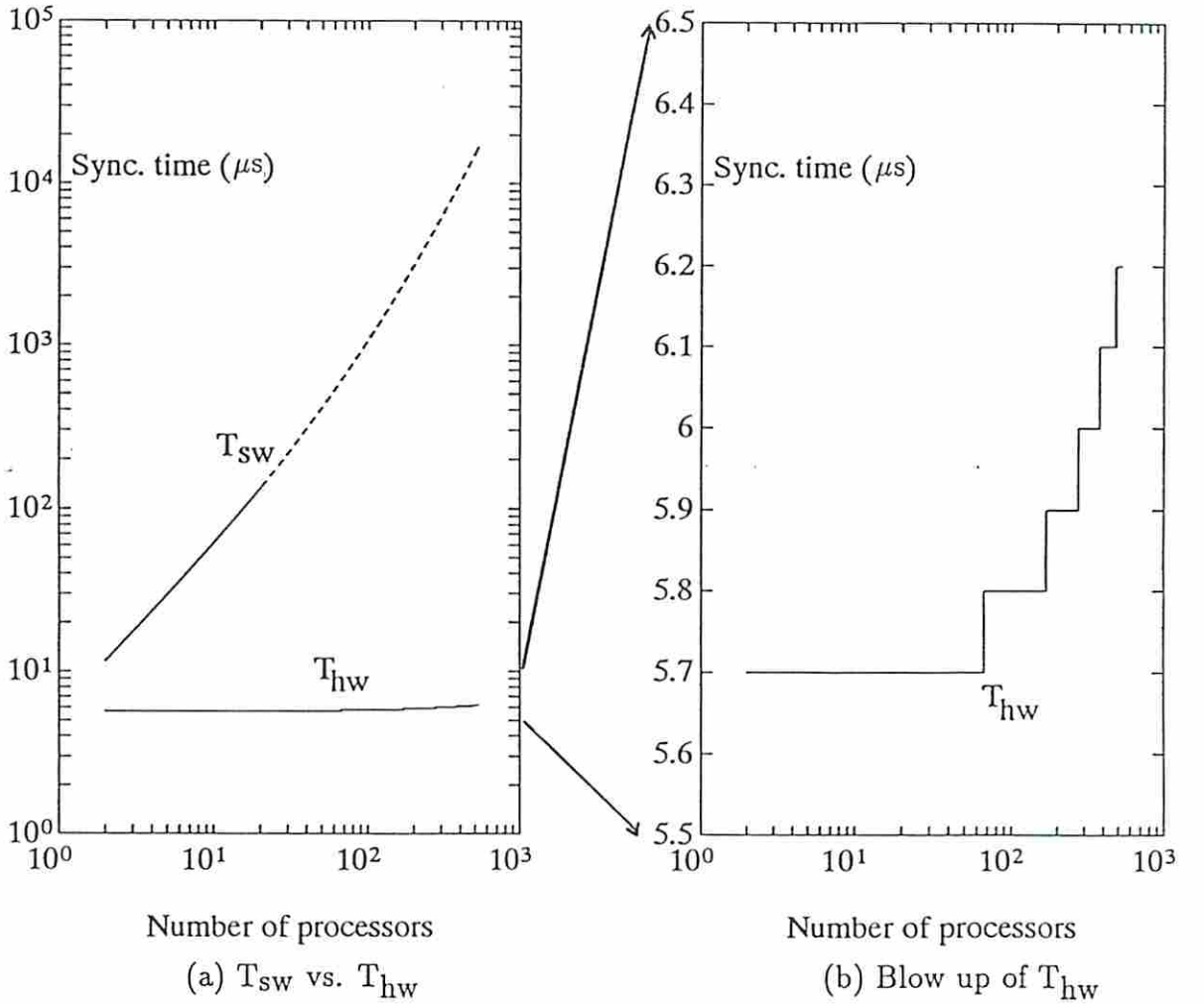


Figure 3: Delay times vs. the number of processors for different values of R_1 (in kΩ).

The values of t_{lock} and t_{mem} for a 20-processor Sequent Symmetry multiprocessor are 5.6 μs and 100 ns [2, 20], respectively. Thus, the software synchronization requires a time T_{SW} of 134 μs. For the wired-NOR barrier, it takes 5.6 μs to reset the control vector, and one memory cycle, i.e. about 100 ns, to read the monitor vector X. As a result, the time T_{HW} for a processor to complete a barrier synchronization is about 5.7 μs. After the control vectors are reset, the hardware responds within 46 ns, which is faster than a memory access time. The above analysis concludes that the wired-NOR hardware is about 30 times faster than that of the *test-and-set* software synchronization, for a small multiprocessor system of 20 processors.

The time difference will be even greater, as the number of processors increases. Figure 4(a) shows the two plot of the times, T_{sw} and T_{hw}, as the number of p processors ranging from 2 to 525. The dashed lines beyond 20 processors are the estimated sync. time for the *test-and-set* software approach. For a large multiprocessor, say $p > 500$, the wired synchronization is at least one thousand time faster than the software approach. Figure 4(b) illustrates the blow up of the T_{hw}. There are small jumps at $p = 67, 171, 275, 379$, and 483 processors within 0.5 μs in the curve. This is due to the delay time of the wired-NOR hardware exceeding multiple of 100 ns, which requires an additional memory cycle, i.e. 0.1μs, for a processor to recognize the completion of the barrier synchronization.



(a) T_{sw} vs. T_{hw} (b) Blow up of T_{hw}

Figure 4: Synchronization time using the wired-NOR barrier, as compared with the *test-and-set* software implementation. ($R_1 = 17.5\Omega$ for the hardware)

4 Barrier Synchronization Operations

We show next how to perform the wired barrier synchronization using the wired-NOR lines. We first specify below the syntax of a *fork* construct, supported by the wired-NOR lines:

fork(func[(arg1, arg2,$\cdots$)], nproc);
void (*func)();
unsigned int nproc;

sync();

init_barrier(bp, func, nproc);
barrier bp;
void (*func)();
unsigned int nproc;

wait_barrier(bp);
barrier bp;

Multiple child processes are created by a *fork* primitive. The number of processes (*nproc*) created is determined statically at compile time or dynamically at run time. The code need not be duplicated for each child process, instead it is shared by all child processes. In fact, only the *process control block* (PCB) and local variables are needed in each child process. Child processes can be treated as *threads* or *light-weight processes*. The *sync* primitive causes the parent process to spin, until all child processes finish their tasks. The parent process can perform other computations between the *fork* and the *sync* primitives, concurrently with the execution of child processes.

Figure 5(a) shows the flow graph for the *fork* and *sync* parallel primitives, when ℓ child processes are forked out concurrently with the parent process. The branch f_i is a task executed by a child process i, and T is the task executed by the parent process. Besides the *fork* and *sync* primitives, we use an *init_barrier*, residing in the parent process, to initialize a barrier as a rendezvous point of *nproc* child processes. We define a special data type, called *barrier*, to specify different barrier points. A *wait_barrier* is used to delay the child process to a busy wait state, until all child processes have reached the same barrier. At that moment,

all child processes stop spinning. Results are not defined, if the barrier is not initialized in a parent process.

A barrier can be used many times by the child processes. Figure 5(b) illustrates the flow graph for using the *init_barrier* and *wait_barrier* along with the use of *fork* and *sync*, when ℓ child processes are created by a parent process. The f'_i, f_i" are the subtasks executed by each child process i, and the T is the task executed by the parent process. The barrier point *bp* in the graph can be used many times by the child processes. The *init_barrier* and *wait_barrier* support flexible synchronization patterns among child processes.

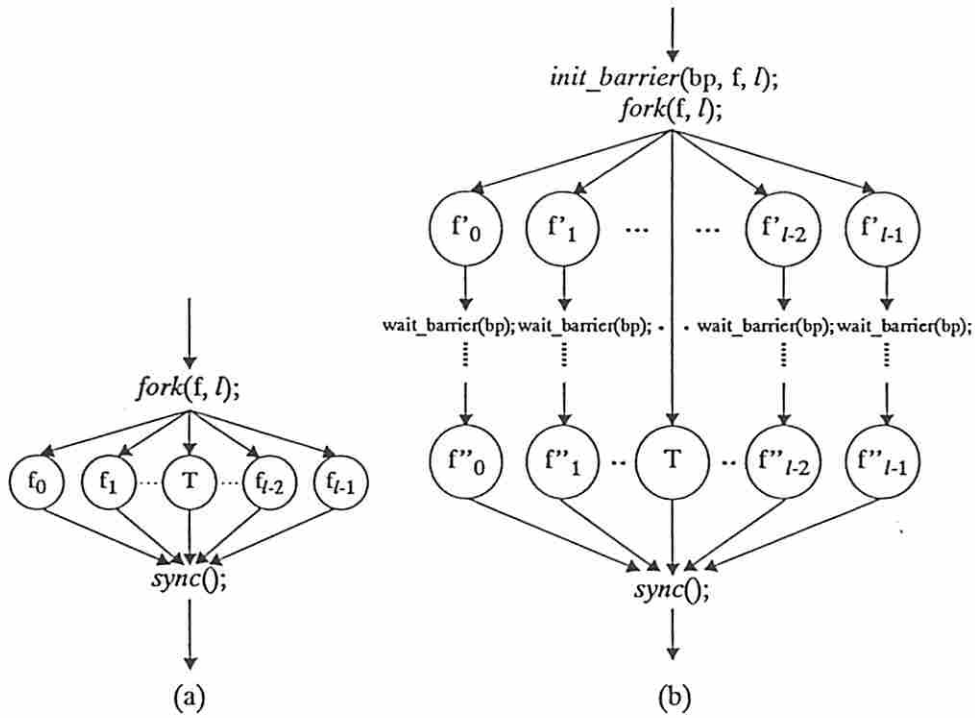
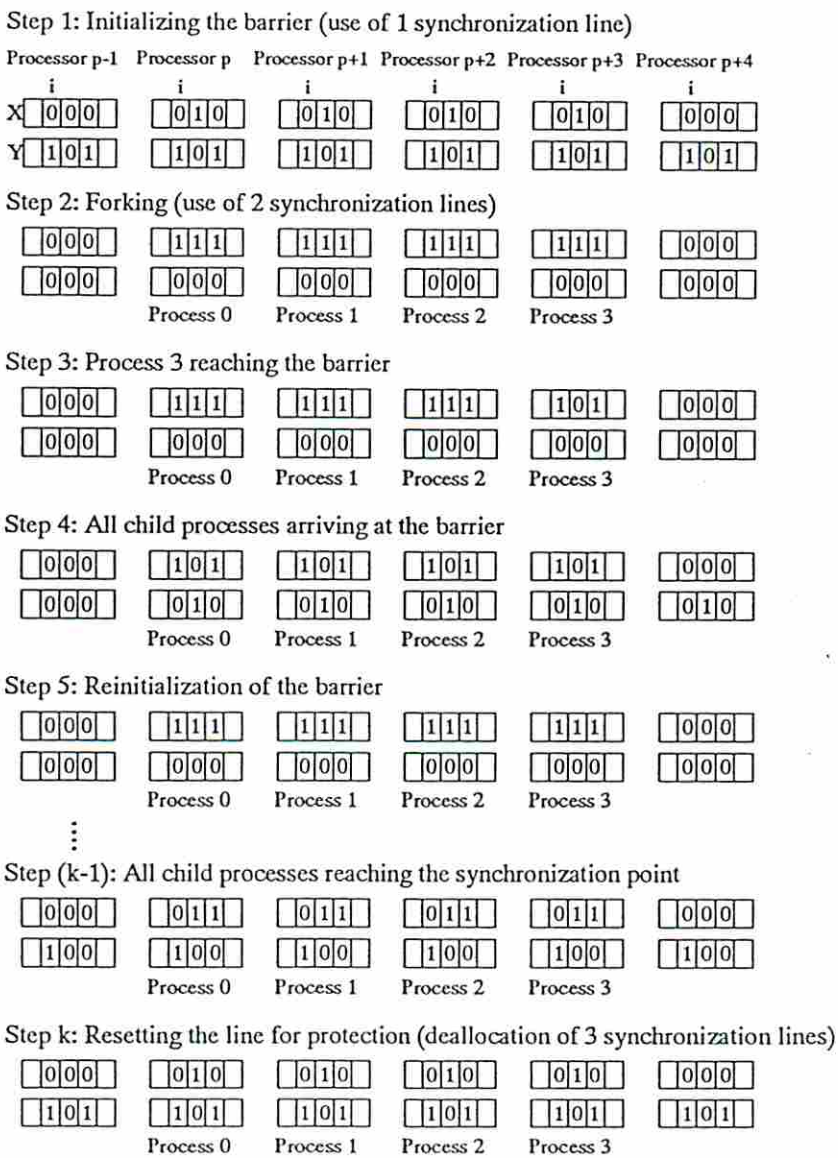


Figure 5: (a) The flow graph to fork and synchronize ℓ child processes, and (b) the flow graph using *init_barrier* and *wait_barrier* to fork ℓ child processes.

The *fork* and *sync* primitives support data partitioning. By passing different values, the *fork* and *sync* support function partitioning as well. The functions of *init_barrier* or *wait_barrier* are similar to those of *fork* and *sync*, except an additional reinitialization step is required. We show in the following example the steps of initializing the *barrier*, and waiting the *barrier* with the forking and synchronization of four processes on four processors.

Example 1: Multiple synchronizations of four processes on four processors using repeated barrier initializations.

The $(i+1)$th snoopy line is allocated to the *barrier*. Each X_{i+1} bit is set to 1 in processor p through $p+3$. Step 2 forks out 4 processes. Besides duplicating 4 copies of the PCB and some local data, the *fork* assigns two snoopy lines, say i and $i+2$ $(0 \leq i, j \leq m-1)$, dispatches 4 processes to 4 processors, and sets the X_i and X_j bits to 1 without changing the remaining bits.



The initialization of the *barrier* is done in Step 1. Suppose process 3 reaches the barrier first in Step 3. It resets the X_{i+1} bit, then starts to read Y_{i+1} bit. Process 3 does not

stop reading the bit, until the (i+1)th snoopy line becomes high, i.e. $Y_{i+1} = 1$. No network traffic is generated by a process, while it is busy waiting. When all processes arrive at the *barrier*, the X_{i+1} bits are reset, and the Y_{i+1} bits become high in Step 4.

The next step is for reinitialization of the *barrier*. The X_{i+1} bit is set to 1 by each individual process. As a result, the *barrier* can be used many times. The Steps 3 through 5 may be repeated as many times as the *barrier* is used. When all child processes reach the synchronization point (see Step S_{k-1}), they can terminated. But, before its termination, each of the child process must reset the X_{i+2}, as indicated in Step S_k. When the *sync* in the parent process detects that line $i+2$ is high, it returns 3 allocated lines to the operating system. The parent process then continues to execute the program behind the *sync* primitive.

¶

Since vector X is accessible by all processors, atomic operation should be used to modify its content. The use of one additional line for synchronization is especially important, when the system is required to support multiprogramming. If only one line is used, the parent process cannot tell whether all child processes catch the voltage change on the line. Therefore, the allocated line can not be released to the operating system. Only when all child processes are complete, the line becomes available. In this case, the operating system has to monitor the termination of all child processes or delay the deallocation until completion. The first approach imposes a high overhead on the operating system, while the second results in a poor utilization of snoopy lines.

Polychronopoulos [18] assumed that all processes execute at the same speed and no reuse of the lines. This assumption will limit the application potential. In another paper [6], the problem was partially solved by adding a latch for each processor to hold the synchronization information, until read by the processor. The problem is still hinged on the assumption that process preemption is not allowed. Our method alleviates this problem by using a dedicated line to make sure the barrier line will be pulled down instantly without delay, once all processes reaching the synchronization point. When the parent process detects the fact that this line becomes high (i.e. all child processes have acknowledged), it releases

the allocated lines and return them back to the kernel for other programs to use. As a result, the wired-NOR lines can be better utilized by many processes.

When the number of processes is much greater than the number of available processors, more snoopy lines are demanded. Besides one line used for protection, each child process is allocated with an exclusive snoopy line. However, child processes among different processors can share the same set of lines. That is, when ℓ child processes with b active synchronization points are executed on P processors ($\ell \geq P$), $b\lceil \ell/P \rceil + 1$ snoopy lines are needed.

5 Coupling Barriers with Semaphores

One can combine the use of wired-NOR snoopy lines with the use of counting semaphores from the shared memory of a multiprocessor system. The purpose is to handle cases limited by insufficient number of snoopy lines. Now, we use two sets of primitives in using of hardware barrier and software semaphores simultaneously:

m_fork(func[(arg1, arg2, $\cdots$)], nproc);
void (*func)();
unsigned int nproc;

m_sync();

m_init_barrier(bp, func, nproc);
barrier bp;
void (*func)();
unsigned int nproc;

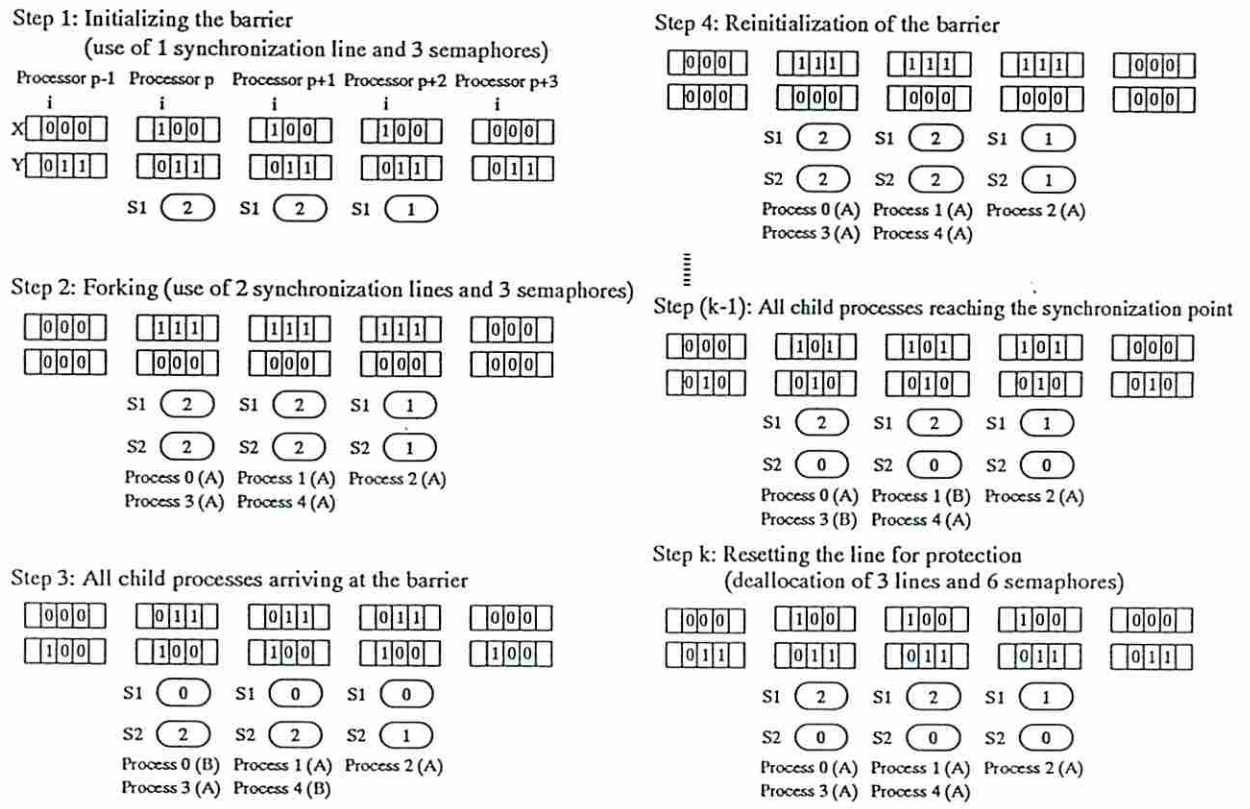
m_wait_barrier(bp);
barrier bp;

The system calls employ one software semaphore among all child processes assigned to the same processor. The use of semaphores implies better support for medium to large-grain parallelism. We assume that the operating system provides kernels, such as *wait* and *signal*, to facilitate the suspension and resumption of processes using these semaphores. The

m_init_barrier and *m_wait_barrier* operate similarly as *m_fork* and *m_sync*, except an additional reinitialization step on semaphores is required for each barrier. The next example shows the steps of using *m_init_barrier* and *m_wait_barrier* in synchronization of five child processes on three processors, using this combined approach. The letters (A) and (B) distinguish *active process* from *blocked process.*

Example 2: Multiple synchronizations of five processes on three processors using three wired-NOR lines and six semaphores.

The initialization of the *barrier* is performed in Step 1. The number of processes assigned to a processor is determined. The ith snoopy line is allocated for the *barrier.* In addition, 3 semaphores (S1) are demanded for each of the allocated processors. The initial value of each S1 is equal to the number of processes to be executed on the corresponding processor.



Step 2 is for process forking. The allocation and initialization of 3 semaphores (S2) are performed in the *m_fork* system call. The values of semaphores S2 are the same as their

counterparts initialized in Step 1, which is equal to the number of child processes accessing it. Only two snoopy lines ($i+1$ and $i+2$) are allocated to each occurrence of *m_fork*: one of the line is used for probing, and the other signals the completion of synchronization. The semaphores are distributed to distinct cache blocks to avoid unnecessary cache invalidations due to false sharing.

When reaching the barrier point, all processes, except the last one in each processor, are blocked by the *wait* kernel. Only active processes consume the CPU time in this case. When all child processes reach the *barrier* (see Step 3), the values of the semaphores S1 all become 0, all X_i bits are reset by the active child processes, and the snoopy line i becomes high again.

Step 4 reinitializes the *barrier*. Each active child process sets X_i bit to 1 again, reloads the value of the S1 to its initial value. The blocked process in each processor are activated by the *signal* kernel. The Steps 3 and 4 may be repeated as many times as the *barrier* is used. When all child processes arrive at the same synchronization point, the values of S2 becomes all 0. The active child processes reset X_{i+1} bits, and the Y_{i+1} bits become high (see Step S_{k-1}). In Step S_k, each active child process resets bit X_{i+2}, and wakes up those blocked processes. When the *m_sync* in the parent process detects that the Y_{i+1} bit becomes high, it releases 3 lines to the operating system and destroys 6 semaphores used.

¶

The atomic operation on the semaphores is implied. Since all processes occupy the same address space, the overheads incurred in *wait* and *signal* kernels are roughly equal to the time to move a PCB between the ready queue and the block queue, which is very short. The context-switching overhead among child processes is estimated as the time to store or to reload the content of all registers in a CPU. When ℓ child processes with b active synchronization points are executed on P processors ($\ell > P$), $b+1$ snoopy lines and bP counting semaphores are needed.

The pseudo codes for these primitives are given below. We use a special data type, called *semaphore*, to support the counting semaphores. The operation *wait* on a semaphore

c, denoted as *wait(c)*, causes an executing process to be blocked (suspended) and placed on a queue for semaphore *c*. The operation *signal* on a semaphore *c*, denoted as *signal(c)*, causes all processes waiting in the queue to be activated (resumed) and ready for execution. The implementation of the *wait* and the *signal* kernels may be accomplished by either using the hardware supporting atomic operators, such as *test-and-set*, *compare-and-swap*, *fetch-and-add* if available, or simply embedded in the cache coherence protocol.

Figure 6 shows the pseudo codes for *m_fork* and *m_sync*. Several additional variables (mask, counter[ ]) are needed by the semaphores. The *mask* variable is exclusively used by the *m_sync* for testing if all child processes have reached the barrier. The size of array *counter* is determined during the forking time and is equal to the number of processors that are allocated to this job. Each counter is a semaphore to be used by a *group* of child processes.

We assume that the compiler would reserve five memory words for the forked processes. The first two words contain the addresses of the synchronization mechanism. The third word stores the reset and probing bit pattern, the fourth word has the bit pattern for protection, and the last word stores the counter index that specifies which counting semaphore the forked process will use.

During the joining stage, a process subtracts 1 from the counting semaphore (the index specified in Loc[4]). The *update-add* operator is employed to ensure that the content of the counter is correct even when multiple processes are accessing it. If the semaphore is greater than 0, the process is temporarily suspended. If the process is the last one to access the semaphore (counter is 0), it resets the synchronization mechanism, spins until the line (indicated in Loc[2]) becomes high, wakes up the suspended processes, then resets the protection line.

Figure 7 shows the pseudo codes for *m_init_barrier* and *m_wait_barrier*. Similar to the previous discussion, each *bp* stores a probing bit pattern. However, only one snoopy line is allocated for each *bp*. Several semaphore variables (b_counter[ ]) are needed. The size of the array is equivalent to the number of processors allocated and is determined at

```
A([arg1, arg2, ···])
{       int Loc[5];
        ......
        ......
        m_join();
}
m_fork(A[(arg1, arg2, ···)],k)
void (*A)();
unsigned int k;
{       shared int mask;
        unsigned int ℓ;
        duplicate k copies of process control block for code A;
        allocate processors P_0, P_1, ···, P_{n-1} processors;    /* k≥n */
        create shared semaphore counter[p];
        allocate snoopy line c_0 and c_1;
                    c_1        m-c_1-1
        mask = 00···00 1 000···000;
        for ℓ = 0, n-1 {
                                                  c_0        m-c_0-1
            update-∨(Pℓ.X, mask ∨ 00···00 1 000···000);
            counter[ℓ]=0;
        }
        for ℓ = 0, k-1 {
            assign process ℓ to processor P_j, j = ℓ mod n, for execution;
            ℓ.Loc[0] = &P_j.X;
            ℓ.Loc[1] = &P_j.Y;
                        c_0        m-c_0-1
            ℓ.Loc[2] = 0···0 1 000···000;
            ℓ.Loc[3] = mask;
            ℓ.Loc[4] = j;
            counter[j] = counter[j] +1;
        }
        start the forked processes;
}
m_sync()
{       while (Pq.Y ∧ mask) ⊕ mask ≠ 0;      /* q: processor that execute this code */
        deallocate the snoopy lines;
        destroy all generated semaphores;
}
m_join()
{       if (update-add(counter[Loc[4]],-1) > 0)
            wait(counter[Loc[4]]);
        else {
            update-∧(*Loc[0], ~ Loc[2]);
            while (*Loc[1] ∧ Loc[2]) ⊕ Loc[2] ≠ 0;
            signal(counter[Loc[4]]);
            update-∧(*Loc[0], ~ Loc[3]);
        }
}
```

Figure 6: The pseudo codes for *m_fork* and *m_sync*. (*: indirect addressing, ∨: bitwise or, &: address operator, ∧: bitwise and, ~: bitwise complement, ⊕: bitwise exclusive-or)

```
A([arg1, arg2, ···])
{       ......
        m_wait_barrier(bp);
        ......
        m_join();
}
m_init_barrier(bp, A, k)
barrier bp;
void (*A)();
unsigned int k;
{       shared unsigned int c_counter[n];
        unsigned int ℓ;
        allocate snoopy line d0;
        forked processes to be executed on processors P0, P1, ···, Pp−1;    /* k≥n */
        create shared semaphore b_counter[n];
                  d0        m−d0−1
        bp = 0···0 1 00···00;
        for ℓ = 0, n − 1 {
            update-∨(Pℓ.X, bp);
            b_counter[ℓ]=0;
        }
        for ℓ = 0, k − 1 {
            j = ℓ mod n;
            b_counter[j] = b_counter[j] +1;
        }
        for ℓ = 0, n − 1
            c_counter[ℓ] = b_counter[ℓ];
}
m_wait_barrier(bp)
barrier bp;
{       if (update-add(b_counter[Loc[4]], −1) > 0)
            wait(b_counter[Loc[4]]);
        else {
            update-∧(*Loc[0], ∼ bp);
            while (*Loc[1] ∧ bp) ⊕ bp ≠ 0;
            b_counter[Loc[4]]= c_counter[Loc[4]];
            update-∨(*Loc[0], bp);
            signal(b_counter[Loc[4]]);
        }
}
```

Figure 7: Pseudo codes for *m_init_barrier* and *m_wait_barrier*. (*: indirect addressing, ∨: bitwise or, &: address operator, ∧: bitwise and, ∼: bitwise complement, ⊕: bitwise exclusive-or)

run time. Additional variables (c_counter[ ]) are demanded, and each of which is exclusively used by one processor for reinitialization purpose in the *m_wait_barrier*. Because the use of the *m_wait_barrier* would incorporate with the use of the *m_fork*, the data initialized by the latter can be used by the former. We make use of such property in the *m_wait_barrier* (Loc[4]), when it tries to access a designated semaphore.

6 Partially-Ordered Barrier Synchronizations

We show next how to use the wire-NOR hardware to support a partially-ordered set of synchronization points involving different processor subsets. This means we can also support *dynamic barriers* as described in [16]. In this case, the barrier patterns are predicted at compile time and process preemption is not allowed.

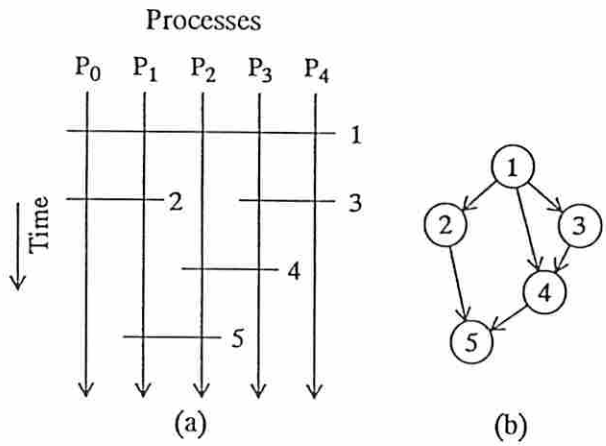
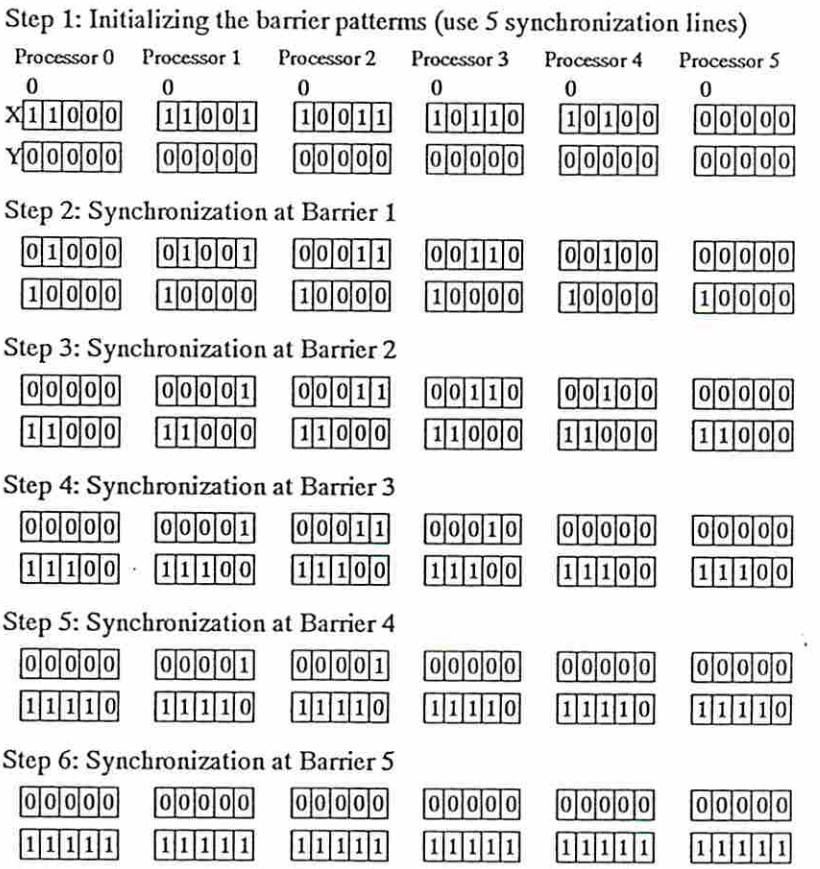


Figure 8: (a) Five synchronization points among five concurrent processes, and (b) the corresponding precedence graph showing the partial orders among them.

We use the same notation used by O'Keefe and Dietz to represent the barrier synchronization of five concurrent processes. As shown in Fig. 8(a), the vertical lines represent the execution path of concurrent processes, while the horizontal lines represent the barriers across the processes. The barrier is a point where every participating process must wait until the others arrive at the same point. The execution order of barriers is represented by the precedence graph (Fig. 8(b)). The following example shows the steps of using wired-NOR hardware to accomplish a sequence of five synchronizations in Fig. 8.

Example 3: Dynamic barrier synchronizations of five processes following a partially ordered precedence graph.

Since there are 5 barriers among the concurrent processes, we use 5 snoopy lines 0 through 4, representing the 5 barriers. Step 1 initializes the barrier patterns. The corresponding bits for the barriers in each process are all set to 1. For example, for Process 0, bits X_0, and X_1 are set to 1, for Process 1, bits X_0, X_1, and X_4 are set to 1, etc. After the barrier patterns are loaded, the processes start to execute.



When a process arrives at barrier 1, it resets bit X_0, then starts to read bit Y_0. The process does not stop reading bit Y_0, until it becomes 1. When all 5 processes reach the barrier, the snoopy line 0 becomes high (see Step 2). Steps 3 and 4 illustrates the synchronization at barriers 2 and 3. We have assumed that Step 3 occurs before Step 4, but the synchronization of Processes 0 and 1 is, in fact, independent of the synchronization of Processes 3 and 4. Our synchronization hardware can support the two synchronizations of barriers 2 and 3 simultaneously.

Steps 5 and 6 show the synchronization at barriers 4 and 5. Since the fourth snoopy line does not become high until Process 2 resets bit X_3 (barrier 4), then bit X_4 (barrier 5). Step 6 will never occur before Step 5. The order in the precedence graph is preserved implicitly by the hardware.

¶

We have shown that the wired-NOR hardware supports concurrent barriers. Any partially ordered set of barriers can be supported. From the above example, one may argue that it needs a lot of lines for an application with hundreds of barriers. Our solution is to reuse the snoopy lines. After a barrier is crossed, the corresponding line is released to another barrier. The allocation and initialization of snoopy lines may be done by a source-to-source precompiler, which, consulting with the precedence graph, inserts suitable library routines in the processes to detect synchronization pattern. For an n-processor system, there can be at most $n/2$ active barriers coexisting simultaneously. This means that $n/2$ wires are sufficient to synchronize n processors.

7 Effects on Multiprogramming Degree

We want to determine the relationship between the number of the snoopy lines used and the degree of multiprogramming, that can be supported by a shared-memory multiprocessor. A single user program may not be able to keep all processors busy. Multiprogramming is an attempt to increase the system utilization and thus throughput by keeping all processors busy performing useful computations all the time.

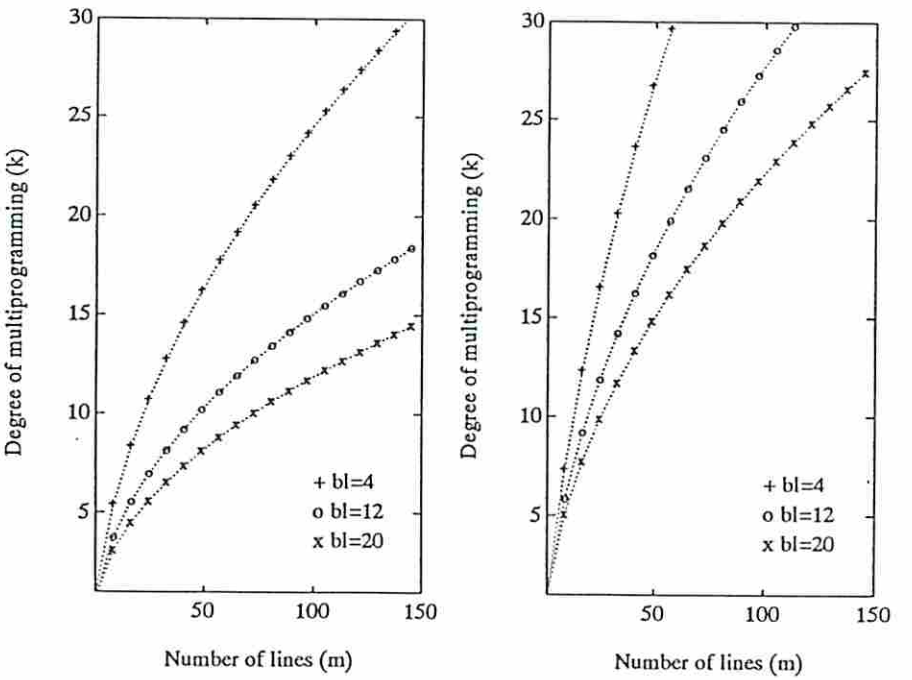
For simplicity, assume $b_i = b$ and $\ell_i = \ell$, where $i = 1, 2, \cdots, k$, as defined in Section 2. Consider a machine with n processors and m snoopy lines. Assume that $P_i = \min(n/k, \ell)$ processors are allocated to each program i. By Eq. 1, the required number S of snoopy lines is approximated by:

$$\begin{aligned} S &\approx \sum_{i=1}^{k} (b\lceil \ell \cdot k/n \rceil + 1) \\ &\approx b\ell k^2/n + k \end{aligned} \tag{3}$$

Since $S \leq m$, we obtain the degree of multiprogramming as follows:

$$k \leq \frac{-n + \sqrt{n^2 + 4b\ell mn}}{2b\ell} \tag{4}$$

The number of required synchronization points, $b\ell$, depends on the parallelism profiles in user programs. The profiles can be used to approximate the workload. For fixed values of $b\ell$ and n, the maximum degree of multiprogramming increases with the order of $\sqrt{m}$. Figure 9 illustrates the effects on multiprogramming degree by the number of snoopy lines used. Since the number of available lines is often lower than the multiprogramming degree desired, further increase in multiprogramming degree is discouraged.



(a) A 32-processor system (b) A 128-processor system

Figure 9: The multiprogramming degree (k) supported versus the number (m) of snoopy lines used under a workload ($b\ell$) varying from 4 to 20 synchronization points in two multiprocessor systems.

Next, we analyze the line allocation policy presented in Section 5, where semaphores are jointly used. Let S_i' be the new number of lines demanded by program i, and S' be the total number of lines required with multiprogramming degree k. Then we have

$$S_i' = b + 1 \tag{5}$$

And $S' = \sum_{i=1}^{k} S'_i = k(b+1)$. Since $S' \leq m$, the degree of multiprogramming is upper bounded by:

$$k \leq \frac{m}{b+1} \tag{6}$$

The multiprogramming degree increases with the number of sync lines used, and inversely with the number of sync points demanded in a program.

For a fixed m, using semaphores can support a higher degree of multiprogramming. The above analysis is based on static snapshot of the multiprogramming workload. By analyzing the workload in user programs, a computer architect can usually determine the appropriate number of snoopy lines, i.e. m, needed in a shared-memory multiprocessor. However, the situation becomes more complicated, where the multiprogramming degree varies from time to time. Synchronization lines are shared resources to be distributed among competing programs. If the multiprogramming degree is not well-controlled at run time, deadlock among them may occur.

To alleviate this problem, we use two approaches: one allows *preemption of snoopy lines from other programs to prevent deadlock.* Another declares *the maximum number of snoopy lines needed prior to execution.* The operating system must determine that the allocation of these lines will always result in a safe system without deadlock. If so, the request is granted. Otherwise, a program must wait, until other programs release enough snoopy lines and restores the system to a safe state.

The first approach is more flexible than the second one. But it inevitably imposes additional overhead in saving the status of synchronization, when the lines are preempted. Thus, it may not be suitable for applications with fine-grain parallelism. The necessity of using the maximum number of required sync lines in the second approach will increase the burden of user programmers. If the number is smaller than actually needed, it may result in a system deadlock. On the other hand, if the number is larger, the program may block other programs from entering the system. These approaches provide useful information for a kernel programmer to modify the multiprocessor OS to take full advantage of the snoopy lines provided.

8 Performance of Simulated Workloads

We evaluate the effectiveness of the wired-NOR synchronization hardware in a modified *run-to-completion* multiprogramming environment via simulation experiments. Three *fork-join* type workloads are used. These workloads represent some parallelism profiles in synthesized programs. These simulation results were obtained from a SUN Sparc workstation. The simulator was written in CSIM Version 13, which is a process-oriented multiprocessor simulation kernel [19]. The time quantum for round-robin is set to 0.1 time unit. The corresponding context-switching overhead is set to be 4% of the time quantum. In an actual system, there will be some variations on computation time for each process due to different cache hit ratios, different processor speeds, etc. To model this situation, we draw a random number uniformly from 0.95 to 1.05 as the simulated computation time for each process.

A. Workload 1: $\nabla FJ(\ell,t)$*, where* $\ell \geq 2$.

Besides the parent process, this workload creates ℓ child processes. Then, the parent process and all child processes must be synchronized at certain barrier. After the synchronization, the number of child processes is decremented by 1. Repeat the above steps until the number of child process equals 2. We use the "∇" symbol to designate the decreasing pattern of parallelism in the workload.

Assume that each child process demands the same computation time t, which equals one step in the parent process between any two adjacent synchronization points. As a result, the workload requires (ℓ-1)t time units. This workload is produced by $\ell - 1$ pairs of *m_fork* and *m_sync* primitives. Figure 10(a) shows the parallelism for ∇FJ(8,1.0), which demands 7.0 units of computation time.

B. Workload 2: $\Diamond FJ(\ell,t)$*, where* $\ell \geq 2$.

This workload is similar to the previous one, except the parallelism profile increases from 2 to ℓ, then decreases to 2. The "$\Diamond$" shape specifies the variation of parallelism with respect to time. The workload demands $2\ell - 3$ units of computation time. The example of workload 2, $\Diamond$FJ(10,1.0) in Fig. 10(b), which needs 17 time units to complete.

C. Workload 3: □FJ(ℓ,f,t), where $\ell \geq 2$ and $f \geq 2$.

Workload 3 has a fixed number of ℓ child processes during its entire life time. Different from the previous 2 workloads, the parent process does not synchronize with the child processes until the last synchronization point. The f represents the total times of synchronization required by the child processes (including the time of synchronization with the parent process). The "□" shape indicates that the parallelism profile does not change with respect to time.

This workload is generated by a pair of *m_fork* and *m_sync* and a pair of *init_barrier* and *wait_barrier* primitives. As a matter of fact, the child processes synchronize at the barrier point $f-1$ times. The total execution time is ft time units. Figure 10 shows the parallelism of an example □FJ(15,5,1.0), which needs 5.0 time units for execution.

¶

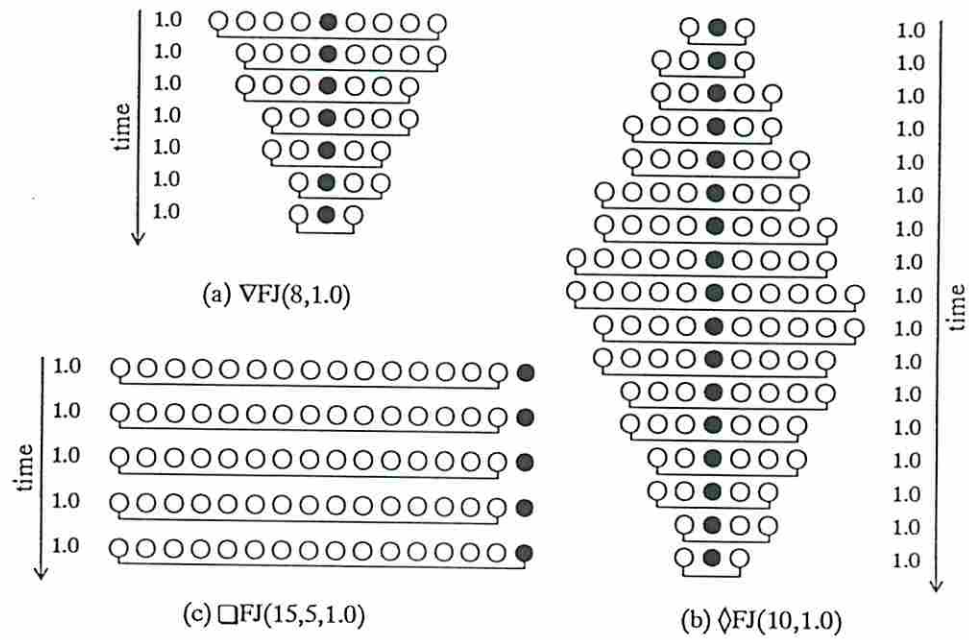


Figure 10: The parallelism profiles of three synthesized workloads used in simulation experiments. (o: a child process, •: one execution step in the parent process, —: a synchronization point, and 1.0: the computation time)

We now examine the response time versus the multiprogramming degree using the three workloads defined in Fig. 10. A similar study for three workloads mixed with equal

probability is also carried out. When a program is complete, the workload type for the next program is determined randomly. The system we have simulated is a 32-processor shared-memory multiprocessor. To simplify the simulation, we assume sufficient snoopy lines, so that preemption of lines among programs will never occur.

When only one type of workload exists in the system, the number of processors allocated to a program is equal to $\min(P/k, \ell)$, where P is the total number of processors in the multiprocessor, k is the multiprogramming degree, and ℓ is the maximum number of processes created (including the parent process). For mixed types of workload, the number of processors allocated to the new incoming program is $\min(A, \ell + 1)$, where A is the total number of free processors in the multiprocessor.

Usually, each parent process uses a dedicated processor for its execution. In some cases, when there is only one processor allocated to a program, the parent process shares the processor with its child processes. If the multiprogramming degree is larger than the number of processors in a multiprocessor, a processor may be shared by several programs. Since this is not our major concern, we thus neglect this extreme case in our simulation studies.

Figure 11 shows the *average response time* (ART) versus the *multiprogramming degree* (MD) for a 32-processor multiprocessor. The ART increases rapidly until the MD equals 16, then it increases slowly. When the MD is greater than 16, a program is executed on 1 or 2 processors. If there are 2 processors, a processor is dedicated to the main program, and the other executes all child processes. In this case, the ART of a program will not differ much, whether it is executed on 1 or 2 processors. This is why the response time increases gradually when the MD is greater than 16.

Comparing the ART of the mixed workload with those without mixing, there are no significant differences. This is because that the processor allocation policy we adopted is heavily dependent on the MD, but insensitive to the workload in the system. As a result, examining the ART of a system to a single workload, one can estimate the ART of the system with a variety of workload types.

The above simulation results reveal the dynamic behavior of a multiprogrammed

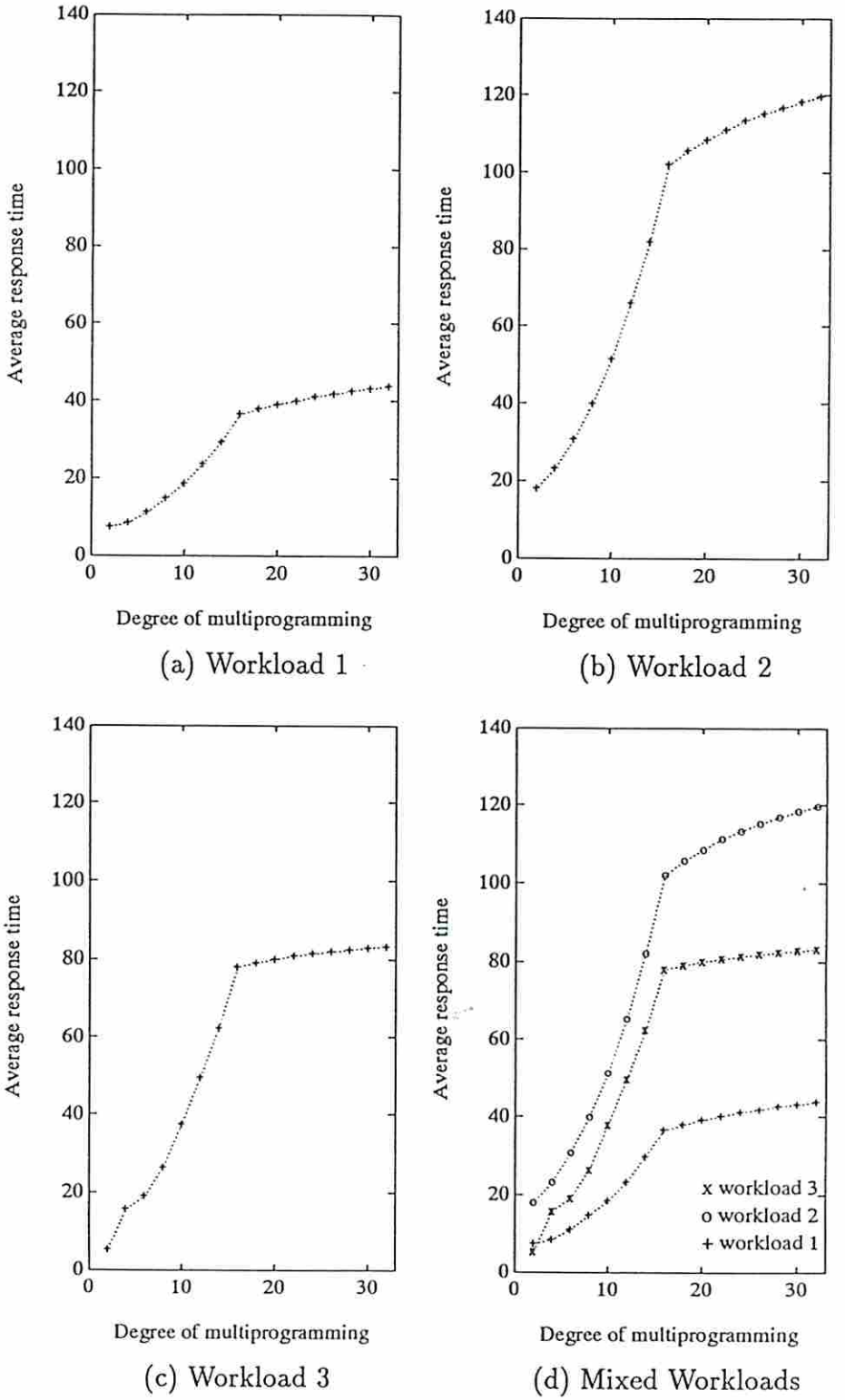


Figure 11: Average response time of a multiprogrammed system with 32 processors.

multiprocessor using the wired-NOR barrier synchronization hardware. By observing the average response time, a computer architect can select a reasonable multiprogramming degree for a target machine, which in turn will determine the appropriate number of snoopy lines used in a multiprocessor design.

9 Conclusions

We have developed a new wired-NOR snoopy mechanism for fast barrier synchronization in a shared-memory multiprocessor. The distributed control for barrier synchronization reduces the network traffic, while the processors are waiting for synchronization. The wired-NOR lines can synchronize 500 processors in about 500 ns. We have defined two sets of *fork* and *join* primitives, which facilitate data and program partitionings commonly demanded in parallel programming. One set supports fine-grain parallelism, while the other supports medium to large-grain parallelism. Incorporating the snoopy lines with a multithreaded OS kernel, pseudo codes of these primitives are given. The snoopy lines are capable of supporting these parallel constructs.

The relationship between the number of snoopy lines and the degree of multiprogramming has been revealed. This provides some guideline for a computer architect to design large multiprocessors that can be fast synchronized. In general, the more snoopy lines the system uses, the higher the degree of multiprogramming it can support. To alleviate the problem of a system deadlock, we have suggested two approaches: *preemption of snoopy lines*, and *declaration of the maximum number of snoopy lines prior to program execution.* These two approaches provides general guidelines for a kernel programmer to modify the UNIX for multiprocessor synchronization.

With three synthesized workloads, we evaluate the wired-NOR synchronization performance through simulation experiments. The simulation results reflected the dynamic behavior of a system. By observing the average response time versus the degree of multiprogramming, one can set a reasonable multiprogramming level for a given multiprocessor. The number of snoopy lines can thus be determined. The proposed hardware is scalable only

for the purpose of synchronization. Towards the design of a fully scalable multiprocessor, one must solve the scalability problems on system interconnects, latency tolerance, cache coherence, etc. which are beyond the scope of this paper.

References

[1] A. Agarwal and M. Cherian. Adaptive Backoff Synchronization Techniques. In *Proc. of The 16th Annual International Symposium on Computer Architecture*, pages 396–406, 1989.

[2] T.E. Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transaction on Parallel and Distributed Systems*, 1(1):6–16, Jan 1990.

[3] N.S. Arenstorf and H.F. Jordan. Comparing Barrier Algorithms. *Parallel Computing*, 12:157–170, 1989.

[4] T.S. Axelrod. Effects of Synchronization Barriers on Multiprocessor Performance. *Parallel Computing*, 3:129–140, 1986.

[5] B. Beck, B. Kasten, and S. Thakkar. VLSI Assist For a Multiprocessor. In *Proc. of The Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 10–20, Oct 1987.

[6] C.J. Beckmann and C.D. Polychronopoulos. Fast Barrier Synchronization Hardware. In *Proc. of 1990 IEEE Supercomputing*, pages 180–189, Nov 1990.

[7] J.R. Goodman, M.K. Vernon, and P.J. Woest. Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors. In *Proc. of 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 64–75, Apr 1989.

[8] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer — Designing an MIMD Shared Memory Parallel Computer. *IEEE Transaction on Computers*, C-32(2):175–189, Feb 1983.

[9] G. Graunke and S. Thakkar. Synchronization Algorithms for Shared-Memory Multiprocessors. *Computer*, pages 60–69, Jun 1990.

[10] R. Gupta. The Fuzzy Barrier: A Mechanism for High Speed Synchronization of Processors. In *Proc. of 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 54–63, Apr 1989.

[11] K. Hwang and D. DeGroot (editors). *Parallel Processing for Supercomputers & Artificial Intelligence.* McGraw-Hill, Inc., 1989.

[12] K. Hwang and S. Shang. Wired-NOR Barrier Synchronization for Designing Large Shared-Memory Multiprocessors. accepted to appear *International Conference on Parallel Processing,* St. Charles, IL, Aug 13-15, 1991.

[13] G.J. Lipovski and P. Vaughan. A Fetch-And-Op Implementation for Parallel Computers. In *Proc. of The 15th Annual International Symposium on Computer Architecture*, pages 384–392, 1988.

[14] J. Millman. *Microelectronics (2nd ed.).* McGraw-Hill, Inc., New York, 1987.

[15] Motorola Inc., Phoenix, AZ. *Bipolar Power Transistor Data, 6ed.*, 1989.

[16] M.T. O'Keefe and H.G. Dietz. Hardware Barrier Synchronization: Dynamic Barrier MIMD (DBM). In *Proc. of 1990 International Conference on Parallel Processing*, pages I43–I46, 1990.

[17] M.T. O'Keefe and H.G. Dietz. Hardware Barrier Synchronization: Static Barrier MIMD (SBM). In *Proc. of 1990 International Conference on Parallel Processing*, pages I35–I42, 1990.

[18] C.D. Polychronopoulos. Compiler Optimizations for Enhancing Parallelism and Their Impact on Architecture Design. *IEEE Transaction on Computers*, C-37(8):991–1004, Aug 1988.

[19] H.D. Schwetman. *CSIM Reference Manual (Revision 13).* Microelectronics and Computer Technology Corporation, Austin, Texas, Jan 1989.

[20] Sequent Computer Systems, Inc., Beaverton, Oregon. *Symmetry Technical Summary (Rev. 1.1)*, Dec 1987.

[21] A. Seznec and Y. Jégou. Synchronizing Processors Through Memory Requests in a Tightly Coupled Multiprocessor. In *Proc. of The 15th Annual International Symposium on Computer Architecture*, pages 393–400, 1988.

[22] H.M. Su and P.C. Yew. On Data Synchronization for Multiprocessors. In *Proc. of The 16th Annual International Symposium on Computer Architecture*, pages 416–423, 1989.

[23] Texas Instruments, Inc., Dallas, Texas. *The TTL Logic Data Book*, Mar 1988.

[24] J. Zahorjan and C. McCann. Processor Scheduling in Shared Memory Multiprocessors. In *Proc. of ACM SIGMETRICS 1990 Conference on Measurement and Modeling of Computer Systems*, pages 214–225, May 1990.