Purdue's Adapter for Parallel Execution and Rapid Synchronization: The TTL_PAPERS Design

H. G. Dietz, T. M. Chung, T. Mattox, and T. Muhammad

School of Electrical Engineering
Purdue University
West Lafayette, IN 47907-1285
hankd@ecn.purdue.edu

January 1995

Abstract

Tightly-coupled parallel machines are being replaced with clusters of workstations connected by communication networks yielding relatively long latencies, but ever-higher bandwidth (e.g., Ethernet, FDDI, HiPPI, ATM). However, it is very difficult to make parallel programs based on fine-grain aggregate operations execute efficiently using a network that is optimized for point-to-point block transfers.

TTL_PAPERS augments a cluster of PCs or workstations with the minimum hardware needed to provide very low latency barrier synchronization and aggregate communication. For example, UNIX user processes within a TTL_PAPERS cluster can perform a barrier synchronization in 2.5 microseconds, including all software overhead. This is four orders of magnitude faster than using UNIX socket connections over an Ethernet.

This paper presents the TTL_PAPERS principles of operation, implementation issues, low-level software interface, and measured performance of the basic operations.

This work has been supported in part by ONR Grant No. N0001-91-J-4013 and NSF Grant No. CDA-9015696.

Keywords

Parallel architecture, Fine-grain parallelism, Workstation clustering, Barrier synchronization, Aggregate communication.

1. Introduction

TTL_PAPERS is the TTL implementation of Purdue's Adapter for Parallel Execution and Rapid Synchronization. Although it provides sufficient support for barrier synchronization and aggregate communication to allow a cluster to efficiently execute fine-grain MIMD, SIMD, or even VLIW code, a four-processor TTL_PAPERS unit uses just eight standard TTL chips. At first, it is very difficult to accept that such inexpensive and simple hardware can implement very effective synchronization and communication operations for parallel processing. This is because the traditional views of parallel and distributed processing rest on a set of basic assumptions that are incompatible with the concept:

- Conventional wisdom suggests that the operating system should manage synchronization and communication, but even a simple context switch to an interrupt handler takes more time than TTL_PAPERS takes to complete a typical synchronization or communication. All interactions with the TTL_PAPERS hardware are I/O port accesses made directly from the user program; there are no OS modifications required and no OS call overhead is incurred.
- Communication operations are characterized primarily by latency (total time to transmit one object) and bandwidth (the maximum number of objects transmitted per unit time). The hardware and software complexity of most interaction methods results in high latency; high latency makes high bandwidth and large parallel grain size necessary. In contrast, TTL_PAPERS is very simple and yields a correspondingly low latency. Providing low latency allows TTL_PAPERS to work well with relatively fine-grain parallelism, but it also means that relatively low bandwidth can suffice.
- A typical parallel computer is constructed by giving each processor a method of independently performing synchronization and communication operations with other processors; in contrast, TTL_PAPERS interactions between processors are performed as aggregate operations based on the global state of the parallel computation, much as in a SIMD machine. This SIMD-like model for interactions results in much simpler hardware and a substantial reduction in software overhead for most parallel programs (as was observed in the PASM prototype [SiN87]). For example, message headers and dynamically-allocated message buffers are not needed for typical TTL_PAPERS communications. It is also remarkably inexpensive for the TTL_PAPERS hardware to test global state conditions such as `any` or `all`.

Thus, TTL_PAPERS does not perform any magic; it simply is based on a parallel computation model that naturally yields simpler hardware and lower latency. TTL_PAPERS is not really a network at all, but a special-purpose parallel machine designed specifically to provide low-latency synchronization and communication, taking full advantage of the "loosely synchronous" execution models associated with fine-grain to moderate-grain parallelism.

1.1. Synchronization

The only synchronization mechanism implemented by TTL_PAPERS is a special type of fine-grain barrier synchronization that facilitates compile-time scheduling of parallel operations [DiO92] [DiC94].

Hardware barrier synchronization was first proposed in a paper by Harry Jordon [Jor78], and has since become a popular mechanism for coordination of MIMD[1] parallel processes. A barrier synchronization is accomplished by processors executing a `wait` operation that does not terminate until sometime after all PEs have signaled that they are `waiting`. However, while building the 16 processor PASM (PArtitionable Simd Mimd) prototype in 1987 [SiN87], we realized that the hardware enabling a collection of conventional processors to execute both MIMD and instruction-level SIMD[2] programs was actually an extended type of barrier synchronization mechanism. Generalizing this barrier synchronization mechanism resulted in several new classes of barrier synchronization architectures, as reported in [OKD90] [OKD90a]. Unlike other PAPERS implementations, the TTL_PAPERS barrier mechanism provides only a subset of these features; however, these features are sufficient to enable conventional processors to efficiently implement fine-grain MIMD, SIMD, and VLIW[3] execution [CoD94a].

The TTL_PAPERS barrier mechanism also provides an effective target for compile-time instruction-level code scheduling [DiO92].

1.2. Communication

In addition to high-performance barrier synchronization, TTL_PAPERS provides low-latency communication. This mechanism is not equivalent to a shared memory nor a conventional message-passing system, but has several advantages for loosely synchronous communication. Basically, it trades asynchrony for lower latency and more powerful static routing abilities.

As a side-effect of a barrier synchronization, each PE can place information into the TTL_PAPERS unit and get information from whichever processor, or group of processors, is selected. Thus, the sender only outputs data; it is up to the receiver to know where to look for the data that the sender has made available to it. In TTL_PAPERS, we further simplify the hardware by extending this concept so that all participating processors must agree on and contribute to the choice of what data each processor will have available. Compared to conventional networks, this method allows less autonomy for each processor, but yields simpler hardware, better performance, and more powerful access to global state.

---

[1] MIMD refers to Multiple Instruction stream, Multiple Data stream; i.e., each processor independently executes it own program, synchronizing and communicating with other processors whenever the parallel algorithm requires.

[2] SIMD refers to Single Instruction stream, Multiple Data stream; i.e., a single processor with multiple function units organized so that the same operation can be performed simultaneously on multiple data values.

[3] VLIW refers to Very Long Instruction Word; i.e., a generalization of SIMD that allows each function unit to perform a potentially different operation on its own data.

The most basic TTL_PAPERS communication operation is a multi-broadcast that sends to each processor a bit mask containing one bit from each processor. This is a very powerful mechanism for examining the global state of the parallel machine. An obvious application is for processors to "vote" on what they would like to do next. For example, it can be used to determine which processors are ready for the next phase of a computation or which processors would like to access a shared resource (e.g., to implement a balanced, conflict-free, schedule for access to an Ethernet). However, because any communication pattern is a subset of multi-broadcast, it can also be used to implement general communication "routing."

In addition to one-bit multi-broadcast, some versions of PAPERS provide the ability to get four bits of data from any processor in a single operation. Using a unidirectional Centronics printer port, four bits is the theoretical maximum number of data bits that can be obtained by one port input operation. TTL_PAPERS implements four bit sends, and even more functionality, using simple NAND logic to combine signals from the processors. Operations directly supported include:

- Since TTL_PAPERS data is literally a four-bit wide NAND of data sent across all processors, computing a four-bit global NAND takes only one operation. Likewise, NAND can implement AND and OR functions by simply complementing the output or by complementing the inputs.
- To accomplish a four-bit broadcast from a single processor, all the other processors simply output 0xf — four 1 bits. Again, the result will be the complement of the data sent.
- To accomplish a one-bit multi-broadcast, each processor sends four bits of data such that processor *i*'s bit *i* is the data it wishes to send, and the other bits are all 1 values. For example, bit 2 will be 1 for processors 0, 1, and 3; thus, NANDing these signals together will result in a signal that is the complement of bit 2 from processor 2.

In summary, TTL_PAPERS provides almost no facility for autonomous communications, but does provide a very rich collection of aggregate communication operations based on global state.

1.3. Interrupts

To facilitate some level of asynchronous operation, TTL_PAPERS provides a separate interrupt broadcast facility so that any processor can signal the others. Such an interrupt does not really generate a hardware interrupt on each processor, rather, it sets a flag that each processor can read at an appropriate time. Although such a check can be made at any time, the two most obvious times are:

- When a barrier `wait` has taken an unexpectedly long time. This would indicate that one or more processors might have generated an interrupt, freezing the barrier state so that other processors would know to check for an interrupt.
- When the OS is invoked to perform a scheduling operation. Thus, gang scheduling and other OS functions can be implemented by having the OS on each processor use the interrupt facility to coordinate across processors.

The TTL_PAPERS interrupt facility provides all the necessary logic for generating an interrupt and providing a special “interrupt acknowledge barrier.”

2. PC Hardware

Although TTL_PAPERS provides very low latency synchronization and communication, it is interfaced to PCs using only a standard parallel printer port and is implemented with a minimal amount of external hardware. This section details the PC hardware involved in use of TTL_PAPERS.

Throughout the following description, we will distinguish between stand-alone PCs and PCs used as processors within a parallel machine by referring to the later as “PEs” — processing elements. The design presented here directly supports 4 PEs (PE0, PE1, PE2, and PE3), and we also detail how the design can be scaled to 8, 16, or 32 PEs. In fact, no significant changes are needed to scale the design to thousands of processors (the hardware structure that needs to scale is significantly simpler than the barrier AND tree implemented in the Cray T3D [Cra93]).

2.1. PE Hardware Interface

No changes are required to make standard PC hardware into a TTL_PAPERS PE. All that is needed is a standard parallel printer port and an appropriate cable. Although some of the PCs on the market provide extended-functionality parallel ports that allow 8-bit bidirectional data connections, many PCs provide only an 8-bit data output connection. Like the original PAPERS, TTL_PAPERS can be used with any PC; it uses only the functions supported by a standard unidirectional parallel port.

But if there is no parallel input port, how does TTL_PAPERS get data into the PC? The answer lies in the fact that the 8-bit data output port is accompanied by 5 bits of status input and 4 bits of open-collector input/output (which are sometimes implemented as output only) on two other ports associated with the 8-bit data output port. The way we use these lines, there are actually 12 bits of data output and 5 bits of data input.

All versions of both TTL_PAPERS and PAPERS use all 5 available input lines. However, the various versions differ in how many and which of output signals are used. Because the open-collector lines are generally not as well driven as the 8-bit data output lines and require access to a different port address, we generally use the open-collector lines only for signals that are modal, i.e., that change relatively infrequently and can have large settling times. The version of TTL_PAPERS discussed here uses 11 of the 12 available output lines, but only the 8-bit data output port is written in the process of normal interactions with the TTL_PAPERS hardware; the 3 other bits are used only for interrupt handling. Table 1 summarizes the signal assignments for the PC parallel port as it is used for the TTL_PAPERS interface.

Table 1: Parallel Signal Assignments

Std. Name	Use In TTL_PAPERS	I/O Address, Mask
Strobe	P_NAK, not interrupt ack	`P_PORTBASE`+2, 0x01
D0	P_D0, bit 0 of output nybble	`P_PORTBASE`+0, 0x01
D1	P_D1, bit 1 of output nybble	`P_PORTBASE`+0, 0x02
D2	P_D2, bit 2 of output nybble	`P_PORTBASE`+0, 0x04
D3	P_D3, bit 3 of output nybble	`P_PORTBASE`+0, 0x08
D4	P_LG, light LED Green	`P_PORTBASE`+0, 0x10
D5	P_LR, light LED Red	`P_PORTBASE`+0, 0x20
D6	P_S0, strobe P_RDY to 0	`P_PORTBASE`+0, 0x40
D7	P_S1, strobe P_RDY to 1	`P_PORTBASE`+0, 0x80
Ack	P_I3, bit 3 of input nybble	`P_PORTBASE`+1, 0x40
Busy	P_RDY, toggling ready signal	`P_PORTBASE`+1, 0x80
PE	P_I2, bit 2 of input nybble	`P_PORTBASE`+1, 0x20
SlctIn	P_I1, bit 1 of input nybble	`P_PORTBASE`+1, 0x10
AutoFD	P_IRQ, interrupt request	`P_PORTBASE`+2, 0x02
Error	P_I0, bit 0 of input nybble	`P_PORTBASE`+1, 0x08
Init	P_SEL, int/ready select	`P_PORTBASE`+2, 0x04
Slct	*reserved*	`P_PORTBASE`+2, 0x08

2.2. PE Port Bit Assignments

Although the parallel port hardware is not altered to work with TTL_PAPERS, the parallel port lines are not used as they would be for driving a Centronics-compatible printer. Thus, it is necessary to replace the standard parallel port driver software with a driver designed to interact with TTL_PAPERS. Toward this end, it is critical to understand which port addresses, and bits within the port registers, correspond to each TTL_PAPERS signal.

There are three port registers associated with a PC parallel port. These registers have I/O addresses corresponding to the port base address (henceforth, called `P_PORTBASE`) plus 0, 1, or 2. Typically, `P_PORTBASE` will be one of 0x378, 0x278, or 0x3bc, corresponding to MS-DOS printer names `LPT1:`, `LPT2:`, and `LPT3:`. As a general rule, most PCs use 0x378 for the built-in port, however, IBM PCs generally use 0x3bc. Workstations based on processors other than the 386, 486, or Pentium, e.g., DEC Alphas, also generally use 0x3bc; however, most of these processors map port I/O registers into memory addresses, so the port operations must be replaced with accesses to the memory locations that correspond to the specified port register I/O addresses. For example, the IBM PowerPC specification places I/O address 0x3bc at physical memory location 0x800003bc.

The bits within the register P_PORTBASE+0 are used to send TTL_PAPERS both strobe and data values, as well as controlling a bi-color LED (red/green Light Emitting Diode) on the PAPERS front panel. If a PC wants to mark itself as not participating in a group of barrier synchronizations, it should simply output 0xcf; this corresponds to setting P_S1, P_S0, P_D3, P_D2, P_D1, and P_D0 all equal to 1. Notice that, if a PC is not connected to one of the TTL_PAPERS cables, the TTL inputs will all float high, causing the missing PC to be harmlessly ignored in operations performed by the PCs still connected. In contrast, setting both P_S1 and P_S0 to 0 will ensure that all barrier operations halt. In normal operation, each TTL_PAPERS operation is triggered by toggling the P_S1 and P_S0 lines between (P_S1=1, P_S0=0) and (P_S1=0, P_S0=1); this can be done by simply exclusive-oring the previous output byte with (P_S1 | P_S0).

A PC sends data to TTL_PAPERS by setting the output nybble bits appropriately and toggling the strobe lines as described above. Performing this operation as two steps, first changing data bits then changing the strobe bits, can be done to increase the reliability of transmission. Data lines should be given time to settle before a new ready signal is derived from the new strobe signals. Changing strobes and data simultaneously results in a race condition in which the data bits have only about 20 nanoseconds "head start" — a small enough margin for many systems to perform unreliably.

The P_LR and P_LG lines are simply used to control a bi-color LED to indicate the status of the PC relative to the currently executing parallel program. When TTL_PAPERS is not in use, both bits should be set to 0, yielding a dark LED. When a parallel program is running, the LED should be lighted green, which is accomplished by making P_LG=1 and P_LR=0. When a PC is waiting for a barrier, it should make its LED red by setting P_LG=0 and P_LR=1. It is also possible to generate an orange status light by setting both P_LG and P_LR to 1, however, this setting is used only rarely (as a "special" status indication).

The second port register, P_PORTBASE + 1, is used to receive data from TTL_PAPERS. To enhance the portability of TTL_PAPERS to somewhat non-standard parallel printer ports, only these five bits are used as input: four bits of data and one bit to act as a ready line. Because 0x40 is the only bit that can be enabled to generate a true interrupt to the PC, earlier versions of PAPERS made P_RDY use 0x40 so that the PAPERS unit could generate a true hardware interrupt when a barrier synchronization had completed. However, this led to an inconvenient order for the bits of the input nybble, and we never found a good use for the true hardware interrupt (because interrupt handlers have too much latency), so the current arrangement makes P_RDY use 0x80. The new arrangement is superior not only because it keeps the bits of the input nybble contiguous, but also because the inversion of P_RDY is harmless, whereas the inversion of an input data line would require extra hardware or software to flip the inverted data bit (e.g., exclusive or with 0x80).

Note also that P_RDY is a toggling ready signal. The original PAPERS unit used software to "reset" the ready signal after each barrier synchronization had been achieved, thus requiring four port operations for each PAPERS synchronization. By simply toggling P_RDY when each new barrier is achieved, TTL_PAPERS can perform barrier synchronizations using just two port operations.

The third port register, `P_PORTBASE` + 2, serves only one purpose for TTL_PAPERS: parallel interrupt support. Actually, for the reasons described earlier, TTL_PAPERS never generates a "real" interrupt to a PC. However, parallel interrupts provide a mechanism for managing the use of the TTL_PAPERS unit in a more sophisticated way, for example: providing a better TTL_PAPERS "check-in" procedure, facilitating abnormal termination of parallel programs, implementing a user-level parallel file system, and even gang scheduling and parallel timesharing of the TTL_PAPERS unit.

To cause a parallel interrupt, a PC simply sets P_IRQ to 1. However, other processors will not notice that a parallel interrupt is pending unless they explicitly check. This is done by changing P_SEL to 1, which causes the normal P_RDY (described above) to be replaced by an interrupt ready flag... until P_SEL is again set to 0. Thus, any PC can check for an interrupt at any time without interfering with the operation of other PCs; for example, while delayed waiting for a barrier, it is essentially harmless to check for an interrupt. To encourage PCs to check for an interrupt, the interrupting PC can set its P_S1 and P_S0 bits to 0 (see above), forcing barriers to be delayed. When all PCs set their P_NAK to 0, this simply acts to perform a special interrupt barrier. The extended interrupt functionality is implemented by sending an interrupt code as a side-effect of this special barrier.

3. TTL_PAPERS Hardware

Thus far, this document has focused on the way in which PC hardware interacts with TTL_PAPERS. In this section, we briefly describe the hardware that implements TTL_PAPERS itself. The TTL_PAPERS design has been carefully minimized to use just 8 standard TTL 74LS-series parts and to fit on a single-layer circuit board (shown in Figure 1). The result is a remarkably simple design that is inexpensive to build, yet fast to operate. The logic design for TTL_PAPERS is logically (and physically) divided into three subsystems: the barrier and interrupt mechanism, the aggregate communication logic, and the LED display control. The complete circuit diagram is given in Figure 2. This section briefly explains how the required functionality is implemented by the board's logic.

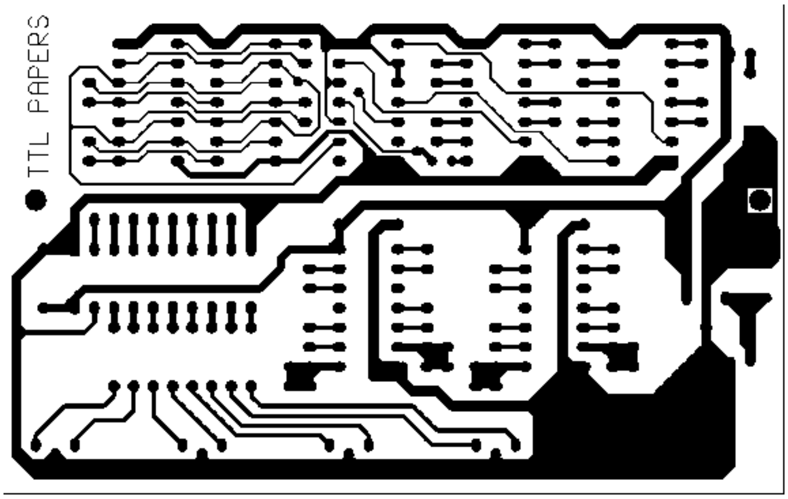

Figure 1: Circuit Board Traces, 1X Plot

3.1. Barrier/Interrupt Hardware

Although the DBM (Dynamic Barrier MIMD) architecture presented in [CoD94] [CoD94b] is far superior to the original DBM design as presented in [OKD90a], neither one is simple enough to be efficiently implemented without using programmable logic devices. Thus, TTL_PAPERS uses a variation on the SBM (Static Barrier MIMD) design of [OKD90]. The primary difference between the previously published SBM and the TTL_PAPERS mechanism is that there are two barrier trees rather than one. The reason is simply that the published SBM silently assumed that the barrier hardware would be reset between barriers, essentially by an "anti barrier." In contrast, the use of two trees allows the hardware for one tree to be reset as a side-effect of the other tree being used, halving the number of operations needed per barrier. Both of these two trees are trivially implemented using a 74LS20, a dual 4-input NAND. The result is latched by setting or resetting a 1-bit register implemented using 1/2 of a 74LS74.

The interrupt logic is remarkably similar to the barrier logic, also using a 74LS20 and 1/2 of a 74LS74. However, there is a difference in the connection between these chips. Interrupt requests are generated by any PE, and acknowledged by all PEs, while barrier synchronizations are always implemented by all PEs. Thus, to invert the sense of the interrupt request output from the NAND, the interrupt logic uses the simple trick of clocking the 1-bit register rather than setting it asynchronously. This trick saves a chip and actually yields a slightly more robust interrupt mechanism, because the interrupt is triggered by an edge rather than by a level.

Finally, because there are not enough input bits for each PE, the above two latched values must be independently selectable for each PE. The obvious way to select between values is using a multiplexor; however, there is no standard TTL 74LS-series part that implements selection between two individual bits. Instead, we construct the multiplexor using two quad driver chips which have outputs that can be independently set to a high-impedance state. The 74LS125 and 74LS126 differ only in the sense of their enable lines; thus, using the same select line to control

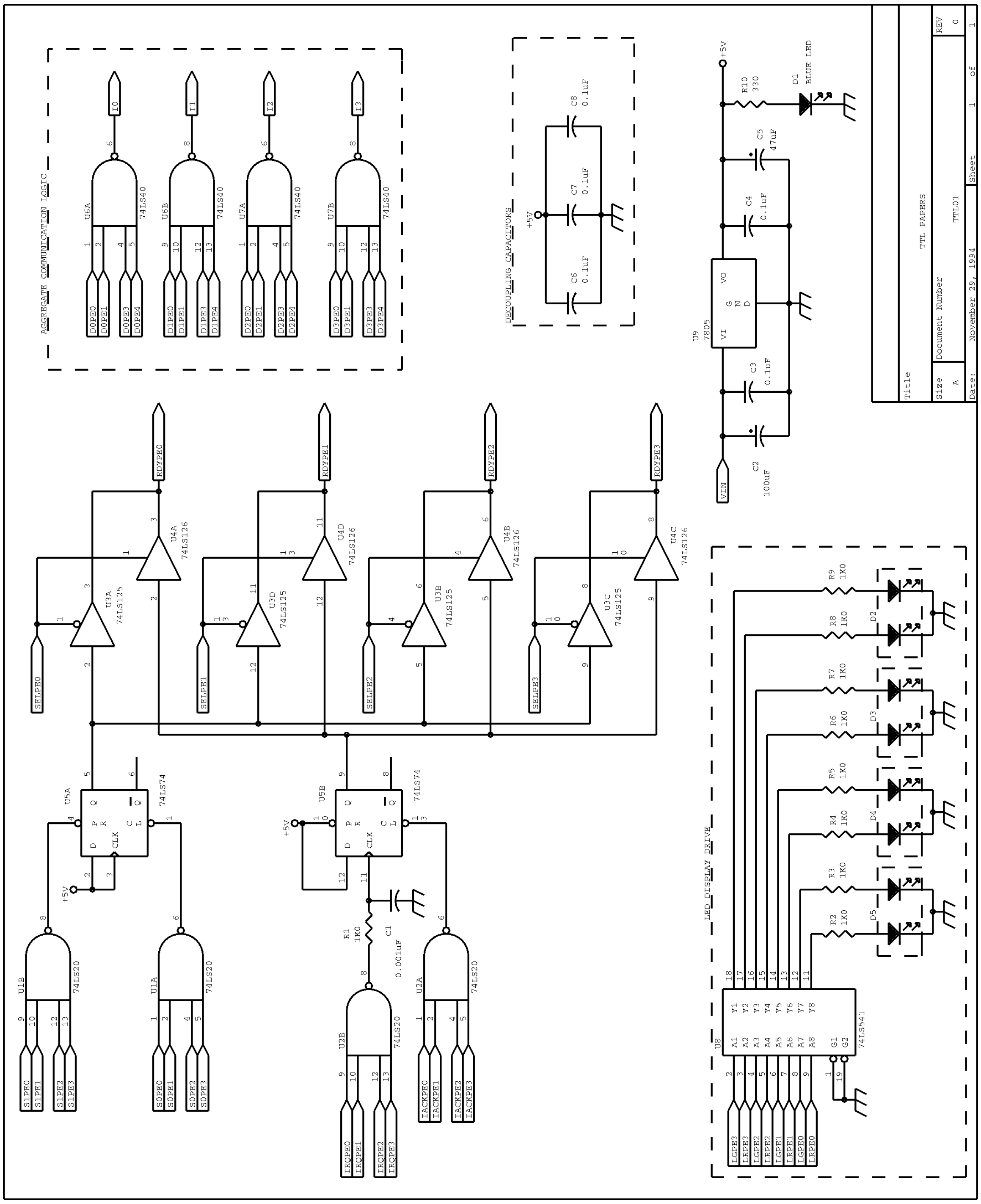

Figure 2: TTL_PAPERS Circuit Diagram

one output on each chip makes it possible to simply wire both outputs together. Only the selected signal will be passed; the other line's driver will be in the high impedance state.

Notice also that this logic trivially can be scaled to larger machines. Every 4 PEs will require a 74LS125 and a 74LS126. The 74LS74 remains unchanged, although a driver chip may be needed to increase fan-out of the outputs for more than 8 processors. The 74LS20 NAND gates simply get replaced by larger NAND trees. For example, for 8 processors, each 1/2 of a 74LS20 is replaced by a 74LS30 (8 input NAND). For 16 processors, chip count is minimized by implementing each NAND tree using a 74LS134 (12-input NAND) and 1/2 of a 74LS20, the outputs of which are combined using 1/4 of a 74LS32 (quad 2-input OR). A somewhat neater 16 processor board layout results from using two 74LS30 and 1/4 of a 74LS32 for each of the NAND trees, and a 32 processor system can easily be constructed using four 74LS30 and 3/4 of a 74LS32 for each NAND tree.

Although the circuitry scales to very large configurations, for a cluster containing more than 32 machines, the circuit complexity is high enough to warrant using a more sophisticated design based on programmable logic components.

3.2. Aggregate Communication Hardware

Although other versions of PAPERS provide internal data latching and fancy communication primitives, TTL_PAPERS simply offers NANDing of the 4 data bits across the processors.

In general, there is nothing tricky about building these NAND trees; they look exactly as described above. However, each NAND tree has an output that must drive one line to each of the processors, and the use of relatively long cables (up to about 10 feet each) requires a pretty good TTL driver. For the case of 4 processors, we have experimentally confirmed that the 74LS40 (dual 4-input NAND buffer) provides sufficient drive to directly power all 4 lines, although the signal transitions become somewhat slow (e.g., about 300 nanoseconds) as the cables get long. In a larger system, we recommend constructing the NAND tree as described above, and then using 74LS244 or 74LS541 octal drivers to increase the drive ability of the NAND outputs. With a typical cable, each driver within a 74LS244 or 74LS541 can drive up to about 4 lines.

3.3. LED Display Hardware

In the PAPERS prototypes, we have experimented with a variety of different status displays. However, by far the easiest display to understand was one using just a single bi-color LED for each processor. The color code is very intuitive: green means running, red means waiting (for more than two “clock” cycles), and black means not in use. The problem is that there is no trivial way to derive these color choices directly from the barrier logic, thus, the LEDs are explicitly set under software control. There is also a power light on the TTL_PAPERS unit, which is a blue LED to avoid confusion with the rest of the status display.

4. PAPERS Software

Although TTL_PAPERS will be supported by a variety of software tools including public domain compilers for parallel dialects of both C and Fortran [DiO92] [CoD94a], in this document we restrict our discussion to the most basic hardware-level interface. The code given is written in C (the ANSI C-based dialect accepted by GCC) and is intended to be run under a UNIX-derived operating system. However, this interface software can be adapted to most existing (sequential) language compilers and interpreters under nearly any operating system.

The following sections discuss the operating system interface, TTL_PAPERS port access, the basic barrier interface, and how NANDing is used to implement data communication.

4.1. Operating System Interface

Although it would certainly be possible to implement the TTL_PAPERS software interface as part of an operating system's kernel, typical latency for a minimal system call is between 5 and 50 times the typical latency for the TTL_PAPERS hardware port operations — layering overhead of a system call for each TTL_PAPERS operation would destroy the low latency performance. Thus, the primary purpose of the OS interface is to obtain direct user-level access to the ports that connect to TTL_PAPERS.

On older architectures, such as the 8080 and Z80, direct user access to I/O was accomplished by simply using the port I/O instructions or by accessing the memory locations that corresponded to the desired memory-mapped I/O device registers. Things are now a bit more complex. Using a PC based on the 386, 486, or Pentium processor, port I/O instructions are now protected operations. Similarly, the DEC Alpha, IBM PowerPC, and Sun SPARC use memory mapped I/O, but physical addresses corresponding to I/O ports generally are not mapped into the virtual address space of a user process. None of these problems is fatal, but it can take quite a while to figure out how to get things working right.

Thus, although the parallel printer port can be directly accessed under most operating systems, here we focus on how to gain direct user process access to I/O ports on a 386, 486, or Pentium-based personal computer running either generic UNIX or Linux.

4.1.1. Generic UNIX

In general, UNIX allows user processes to have direct access to all I/O devices. However, only processes that have a sufficiently high I/O priority level can make such accesses. Further, only a privileged process can increase its I/O priority level — by calling `iopl()`. The following C code suffices:

```
if (iopl(3)) {
    /* iopl failed, implying we were not priv */
    exit(1);
}
```

However, this call grants the user program access to *all* I/O, including a multitude of unrelated ports.

In fact, this call allows the process to execute instructions enabling and disabling interrupts. By disabling interrupts, it is possible to ensure that all processors involved in a barrier synchronization act precisely in unison; thus, the average number of port operations (barrier synchronizations) needed to accomplish some TTL_PAPERS operations can be reduced. However, background scheduling of DMA devices (e.g., disks) and other interference makes it hard to be sure that a UNIX system will provide precise timing constraints even when interrupts are disabled, so we do not advocate disabling interrupts.

Even so, performance of the barrier hardware can be safely improved by causing UNIX to give priority to a process that is waiting for a barrier synchronization. This improves performance because if any one PE is off running a process that has nothing to do with the synchronization, then all PEs trying to synchronize with that PE will be delayed. The priority of a privileged UNIX process can increased by a call to `nice()` with a negative argument between -20 and -1.

4.1.2. Linux

Although Linux supports the UNIX interface described in the previous section, it also provides a more secure way to obtain access to the I/O devices. The `ioperm()` function allows a privileged process to obtain access to only the specified port or ports. The C code:

```
if (ioperm(P_PORTBASE, 3, 1)) {
    /* like iopl, failure implies we were not priv */
    exit(1);
}
```

Obtains access for 3 ports starting at a base port address of `P_PORTBASE`. Better still, if the operating system is managing all parallel program interrupts, only the first two ports need to be accessible:

```
if (ioperm(P_PORTBASE, 2, 1)) {
    /* like iopl, failure implies we were not priv */
    exit(1);
}
```

Because the 386/486/Pentium hardware checks port permissions, this security does not destroy port I/O performance; however, checking the permission bits does add some overhead. For a typical PC parallel printer port, the additional overhead is just a few percent, and is probably worthwhile for user programs.

4.2. Port Access

Although Linux and most versions of UNIX provide routines for port access, these routines often provide a built-in delay loop to ensure that port states do not change faster than the external device can examine the state. Consequently, the TTL_PAPERS support code uses its own direct assembly language I/O calls. The code is:

```
inline unsigned int
inb(unsigned short port)
{
    unsigned char _v;
__asm__ __volatile__ ("inb %w1,%b0"
        :"=a" (_v)
        :"d" (port), "0" (0));
    return(_v);
}

inline void
outb(unsigned char value,
unsigned short port)
{
__asm__ __volatile__ ("outb %b0,%w1"
        : /* no outputs */
        :"a" (value), "d" (port));
}
```

The basic TTL_PAPERS interface is thus defined in terms of the above port operations on any of three parallel printer port interface registers. The following definitions assume that `P_PORTBASE` has already been set to the base I/O address for the port, which is generally 0x378 on PC clones and 0x3bc for IBM ValuePoint PCs.

```
/*      To output to P_PORTBASE */
#define P_OUT(x) \
        outb(((unsigned char)(x)), \
             ((unsigned short) P_PORTBASE))

/*      To input from P_PORTBASE+1 */
#define P_IN() \
        inb((unsigned short) (P_PORTBASE + 1))

/*      To output to P_PORTBASE+2 */
#define P_MODE(x) \
        outb(((unsigned char)(x)), \
             ((unsigned short) (P_PORTBASE + 2)))
```

It is important to note that, even though the TTL_PAPERS hardware interrupt mechanism does not have to be used, it is necessary that any processor connected to TTL_PAPERS set the mode port so that the `P_RDY` bit, and not `P_INT`, is visible. This can be done by executing:

```
P_MODE(P_NAK);
```

As part of the TTL_PAPERS initialization code.

4.3. Barrier Interface

Logically, each barrier synchronization consists of two operations:

[1] signaling that we are at a barrier and

[2] waiting to be signaled that the barrier synchronization has completed.

Although the TTL_PAPERS library generally combines these operations, here we discuss them as two separate chunks of code. The code for signaling that we are at a barrier is simply:

```
P_OUT(last_out ^= (P_S0 | P_S1));
```

This code just flips the sense of both strobe bits. Because `last_out` is initialized to have only one of the strobe bits high, this has the effect of alternating between `P_S0` and `P_S1`. Nothing else is changed, including the output data bits.

The code that actually waits for the barrier synchronization to complete is larger than one might expect, even though typically only a few instructions are executed. The reason that there is so much code has to do with three features of the TTL_PAPERS interface. The first complication is that the ready signal toggles, rather than always going to the same value. The second complication is that the status LEDs are software controlled. The third complication is that we want to check for an interrupt whenever a barrier is excessively delayed. The resulting code is something like:

```
/* Which condition am I waiting for? */
if (last_out & P_S0) {
    /* Waiting for P_RDY */
    if ((!(P_IN() & P_RDY)) && (!(P_IN() & P_RDY))) {
        /* Polled twice, make LED red */
        P_OUT(last_out ^= (P_LG | P_LR));
        /* Continue waiting */
        while (!(P_IN() & P_RDY)) CHECKINT;
        /* Ok, LED green again */
        P_OUT(last_out ^= (P_LG | P_LR));
    }
  } else {
    /* Waiting for not P_RDY */
    if ((P_IN() & P_RDY) && (P_IN() & P_RDY)) {
        /* Polled twice, make LED red */
        P_OUT(last_out ^= (P_LG | P_LR));
        /* Continue waiting */
```

```
        while (P_IN() & P_RDY) CHECKINT;
        /* Ok, LED green again */
        P_OUT(last_out ^= (P_LG | P_LR));
    }
}
```

In the initial version of the support library, `CHECKINT` is nothing — TTL_PAPERS interrupts are not used. However, we can easily check for an interrupt by defining `CHECKINT` as:

```
{
    /* Make P_INT status visible */
    P_MODE(P_SEL | P_NAK);
    /* Check for interrupt */
    if (P_IN() & P_INT) {
        /* Process the interrupt.... */
    } else {
        /* Restore P_RDY */
        P_MODE(P_NAK);
    }
}
```

4.4. NAND Data Communication

Although the TTL_PAPERS library provides a rich array of aggregate communication operations, all that the hardware really does is a simple 4-bit NAND as a side-effect of a barrier synchronization. However, this operation typically requires 5 port accesses rather than just the 2 port accesses used to implement a barrier synchronization without data communication. The extra port operations are required because:

- When a PE changes one or more of its output data bits, the TTL_PAPERS hardware NANDs in the new data, immediately changing the input data bits for all PEs. Thus, if one PE gets far enough ahead of another PE, it could change the data before the slower PE has been able to read the previous NAND result. If interrupts are disabled and an initial ordinary barrier synchronization is performed, then a block of data can be transmitted safely using static timing analysis alone to ensure that this race condition does not occur. Unfortunately, it isn't practical to disable interrupts under UNIX, so the safe solution is to follow each data transmitting barrier with an ordinary barrier that ensures all PEs have read the data before any PE can change the data. This adds 2 port operations.
- The TTL_PAPERS hardware is fed both data and strobe signals on the same port. Thus, data and strobe signals change simultaneously. As they race through the cables to the TTL_PAPERS logic and back out the cables, the TTL_PAPERS logic should give the NAND data at least a 20 nanosecond lead over the toggling of the `P_RDY` bit, but is 20

nanoseconds enough to ensure that the NAND data doesn't lose the race? Well, it depends on the implementation of the PC port hardware and the electrical properties of the cables... which is another way of saying that the race isn't acceptable. An additional port operation is used to ensure that the data wins the race.

There are two possible ways to insert the extra operation. One way is to double the output operation, so that we first change the data and then toggle the strobe. This has the advantage that only PEs that are actually changing their data would need to insert the extra operation. If only early PEs change their data, we don't see any delay; however, the asymmetry of the PE operations can actually result in more delay than if the extra operation was always inserted. The other alternative is to resample the NAND data value after `P_RDY` has signaled it is present. This is somewhat more reliable than the doubling of the output operation, but the delay is always present for any PE that reads the data.

In any case, the result is that a barrier synchronization accompanied by an aggregate communication will take 5 port operations. For example, to perform a reliable NAND with our contribution being the 4-bit value *x*, we could first:

```
last_out = ((last_out & 0xf0) | x);
```

This sets the new data bits so that a barrier synchronization will transmit them along with the strobe. The next step is to perform a barrier synchronization, precisely as described in section 4.3. Having completed the barrier, we know that the NAND data should now be valid, so we resample it:

```
nand_result = P_IN();
```

Finally, a second barrier synchronization is performed to ensure that no PE changes its output data until after everyone has read the current NAND data.

Notice that *nand_result* is actually an 8-bit value with the NAND data embedded within it; thus, to extract the 4-bit result we simply use:

```
((nand_result >> 3) & 0x0f)
```

All the TTL_PAPERS library communication operations are built upon this communication mechanism. For example, ANY, ALL, and voting operations require just one such operation, or 5 port accesses. Larger operations, such as an 8-bit global OR, 8-bit global AND, or broadcast require 2 of the above transmission sequences, or 10 port accesses.

5. Performance

The performance of the basic TTL_PAPERS operations is summarized in the table below. These numbers were obtained using 486DX33-based IBM ValuePoint PCs running Linux version 1.1.75. Machines with faster printer ports obtain correspondingly higher performance; in fact, performance can be increased by nearly an order of magnitude without any other changes.

TTL_PAPERS Operation	**Latency for 4 PEs (microseconds)**	**Latency for *n* PEs (microseconds)**
Partition	0.1	0.1
Barrier Synchronization	2.5	2.5
ANY Test	6.3	6.3
ALL Test	6.3	6.3
4-bit Global OR	6.3	6.3
1-bit Multibroadcast	6.3	6.3 * ceil(*n* / 4)

TTL_PAPERS Operation	**Bandwidth for 4 PEs (bytes/s per PE)**	**Bandwidth for *n* PEs (bytes/s per PE)**
Broadcast	78K	78K
Permutation	20K	78K / *n*
Arbitrary Multibroadcast	20K-78K	78K / (*number of sources*)
Global OR	78K	78K

The primary observation is that the total UNIX-process to UNIX-process latency is remarkably low. For example, we benchmarked a 64-processor nCUBE2 at 206 milliseconds for a simple barrier synchronization — 83,000 times slower than the TTL_PAPERS cluster! In fact, the minimum latency for sending a single message on the nCUBE2 was reported to be between 32 and 110 microseconds [BrG94], which is between 13 and 44 times as long as it takes for a TTL_PAPERS barrier synchronization. The same paper [BrG94] quotes that the Intel Paragon XP/S has a minimum message latency of 240 microseconds, which is 97 times the TTL_PAPERS barrier latency. Of course, any conventional workstation network will have a minimum message latency of at least hundreds of microseconds, simply due to the overhead of UNIX context switching and the socket protocol.

The second observation is that the bandwidth is not very impressive. However, bandwidth for TTL_PAPERS isn't really measuring the same thing as bandwidth for a conventional network. If you want to perform asynchronous block transfers, TTL_PAPERS cannot compete with a conventional network. However, the bandwidth for the aggregate operations (listed above) is much higher for TTL_PAPERS than for nearly any other network attempting to synthesize these operations. In summary, if a program needs to perform an aggregate operation, use TTL_PAPERS. If it needs to do some other type of communication, use the conventional network — having TTL_PAPERS on a cluster does not interfere with any other network's operation.

6. Conclusion

In this paper, we have detailed the design of the TTL_PAPERS hardware. This public domain design represents the simplest possible mechanism to efficiently support barrier synchronization, aggregate communication, and group interrupt capabilities — using unmodified conventional workstations or personal computers as the processing elements of a fine-grain parallel machine. We do not view TTL_PAPERS as the ultimate mechanism, but rather as an introductory step toward the more general and higher performance barrier and aggregate communication engines that have been at the core of our research since 1987. The key thing to remember about TTL_PAPERS is not what it is, but rather *why* it is.

Why is TTL_PAPERS so much lower latency than other networks? Because it doesn't have a layered hardware and software interface. *Why* is the hardware so simple? Because it isn't a network; the fact that TTL_PAPERS communications are a side-effect of barrier synchronization eliminates the need for buffering, routing, arbitration, etc. *Why* is it useful? Because, although shared memory and message passing hardware is very common, the most popular high-level language and compiler models for parallelism are all based on aggregate operations — exactly what TTL_PAPERS provides. Further, barrier synchronization is the key to efficiently implementing MIMD, SIMD, and VLIW mixed-mode execution. *Why* didn't somebody do it earlier? We and others did. The problem is that tightly coupled design of hardware and compiler is not the standard way to build systems, so earlier designs (e.g., PASM, TMC CM-5) tended to use too much hardware and interface software, cost too much, and perform too poorly.

Higher-performance versions of PAPERS are on the way. In fact, this TTL_PAPERS design is the sixth PAPERS design we have built and tested, and three of the other designs easily outperform it... but they use much more hardware. We are currently working on a smart parallel port card for the ISA bus that will roughly quadruple the performance of TTL_PAPERS without any change to its hardware. We are also pursuing versions of the higher-performance designs based on Texas Instruments FPGAs, and anticipate a PCI interface to future PAPERS units. Also watch for releases of various compilers targeting PAPERS. The WWW URL `http://garage.ecn.purdue.edu/~papers/` will be the primary place for announcing and distributing future releases of both hardware and software.

References

[BrG94] U. Bruening, W. K. Giloi, and W. Schroeder-Preikschat, "Latency Hiding in Message-Passing Architectures," *8th International Parallel Processing Symposium*, Cancun, Mexico, April, 1994, pp. 704-709.

[CoD94] W. E. Cohen, H. G. Dietz, and J. B. Sponaugle, *Dynamic Barrier Architecture For Multi-Mode Fine-Grain Parallelism Using Conventional Processors; Part I: Barrier Architecture,* Purdue University School of Electrical Engineering, Technical Report TR-EE 94-9, March 1994.

[CoD94a] W. E. Cohen, H. G. Dietz, and J. B. Sponaugle, *Dynamic Barrier Architecture For Multi-Mode Fine-Grain Parallelism Using Conventional Processors; Part II: Mode Emulation,* Purdue University School of Electrical Engineering, Technical Report TR-EE 94-10, March 1994.

[CoD94b] W. E. Cohen, H. G. Dietz, and J. B. Sponaugle, "Dynamic Barrier Architecture For Multi-Mode Fine-Grain Parallelism Using Conventional Processors," *Proc. of 1994 Int'l Conf. on Parallel Processing,* St. Charles, IL, pp. I 93-96, August 1994.

[Cra93] *Cray T3D System Architecture Overview*, Publication HR-04033, Cray Research, Inc., 2360 Pilot Knob Road, Mendota Heights, MN 55120, 1993.

[DiC94] H. G. Dietz, W. E. Cohen, T. Muhammad, and T. I. Mattox, "Compiler Techniques For Fine-Grain Execution On Workstation Clusters Using PAPERS," *7th Annual Workshop on Languages and Compilers for Parallel Computing* (also to appear as a book chapter from Springer Verlag), Cornell University, August 1994.

[DiM94] H. G. Dietz, T. Muhammad, J. B. Sponaugle, and T. Mattox, *PAPERS: Purdue's Adapter for Parallel Execution and Rapid Synchronization*, Purdue University School of Electrical Engineering, Technical Report TR-EE 94-11, March 1994.

[DiO92] H. G. Dietz, M.T. O'Keefe, and A. Zaafrani, "Static Scheduling for Barrier MIMD Architectures," *The Journal of Supercomputing*, vol. 5, pp. 263-289, 1992.

[Jor78] H. F. Jordon, "A Special Purpose Architecture for Finite Element Analysis," *Proc. Int'l Conf. on Parallel Processing*, pp. 263-266, 1978.

[OKD90] M. T. O'Keefe and H. G. Dietz, "Hardware barrier synchronization: static barrier MIMD (SBM)," *Proc. of 1990 Int'l Conf. on Parallel Processing,* St. Charles, IL, pp. I 35-42, August 1990.

[OKD90a] M. T. O'Keefe and H. G. Dietz, "Hardware barrier synchronization: dynamic barrier MIMD (DBM)," *Proc. of 1990 Int'l Conf. on Parallel Processing,* St. Charles, IL, pp. I 43-46, August 1990.

[SiN87] T. Schwederski, W. G. Nation, H. J. Siegel, and D. G. Meyer, "The Implementation of the PASM Prototype Control Hierarchy," *Proc. of Second Int'l Conf. on Supercomputing,* pp. I 418-427, 1987.

Table of Contents