PLUS: A Distributed Shared-Memory System

Roberto Bisiani and Mosur Ravishankar
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract.
PLUS is a multiprocessor architecture tailored to the fast execution of a single multithreaded process; its goal is to accelerate the execution of CPU-bound applications. PLUS supports shared memory and efficient synchronization. Memory access latency is reduced by non-demand replication of pages with hardware-supported coherence between replicated pages. The architecture has been simulated in detail and the paper presents some of the key measurements that have been used to substantiate our architectural decisions. The current implementation of PLUS is also described.

1. Introduction.
Shared memory is one of the most popular parallel processing models because of the ready availability of bus-based shared-memory systems. Bus-based systems are relatively easy to build, but, because of their limited bus-bandwidth, do not perform well with fast processors, large number of processors, or algorithms that have a poor cache hit-ratio. In fact, most shared memory systems are limited to 10-20 processors or to slow processors and their main application is executing a non-parallel multi-user Unix load. On the other hand, the system we are building, called PLUS, is aimed at efficiently executing a single multithreaded process by using distributed memories, hardware supported memory coherence and synchronization mechanisms.

In order to maintain reasonable memory performance with a large number of fast processors it is necessary to distribute the memory among the processors and connect them with a scalable communication mechanism. The implementation of such a physically distributed, but logically shared, memory system is difficult, because communication latency hinders fast access to remote memories. The operations that cause performance degradation are remote memory access and synchronization.

Caching is the key to implementing fast remote memory access. Usually, caching is performed by hardware on demand, together with a protocol that guarantees memory coherence. In order to be effective in a medium/large system, this requires a substantial amount of fast and expensive hardware. PLUS relies on software-controlled, non-demand caching of data among multiple memories, and a simple protocol implemented in hardware to insure memory coherence. This mechanism is described in Section 2.

The latency of synchronization operations often cannot be reduced by caching, because these operations require exclusive access to synchronization variables. Systems that use caching with invalidation must ensure no other cache has a copy of the synchronization variable before performing the synchronization. PLUS makes it possible to hide the elapsed time between the start of a synchronization operation and its completion by providing separate mechanisms for each. This mechanism and its uses are described in Section 3.

This paper is mainly concerned with architectural issues, see [4] for a description of the software environment.

Although PLUS is aimed at the execution of a single, multithreaded program, some of the ideas proposed in this paper can be applied to a general-purpose, multi-user system.

Memory access protocols can be described according to the kind of ordering they enforce and the caching mechanisms they use. We will touch on these questions in general before we describe PLUS’s protocol.

2.1 Strong and Weak Ordering.
Most programmers take for granted that the result of a sequence of write operations performed by a processor will be immediately observable by any other processor exactly in the same order as it was performed. On a bus-based system, all write operations can be easily made visible to all the processors either by immediately announcing them on the bus (as in a write-through protocol) or by announcing them only when necessary (as with copy-back protocols). These systems are
sequentially consistent [16] and strongly ordered [8], and let programmers use ordinary read/write memory operations to implement synchronization operations.

When there are multiple physical memory units and there is no global communication medium, enforcing strong ordering requires time consuming protocols. Fortunately, a weaker form of ordering, in which actions performed by one processor do not immediately have to become visible to all the other processors, is usually sufficient.

Typically, a parallel program alternates between a long sequence of normal read and write operations on shared data structures and synchronization operations (e.g. P and V). Enforcing strong ordering among normal read and write operations is not necessary, if the programmer understands system synchronization operation should be used whenever two concurrent computations have to obey a specific order in accessing the shared data. A memory system with these characteristics is said to implement weak ordering. See [18, 8] for a formal definition.

When a system implements a weak-ordering model, programmers must explicitly flag synchronization operations for the system to implement them correctly. For example, a buffer shared between a producer and a consumer process is usually associated with a flag that is set when the buffer is full and cleared when it is empty. In a weakly ordered system, if the flag and buffer are allocated in separate memory modules and only normal read and write operations are used, it is possible for the consumer to observe the flag as full before the writes to the buffer have completed. For a weakly ordered system to work correctly, operations on the flag must be strongly ordered. We believe most programs will port to a weakly ordered system without any change. For example, we have examined the code of the Mach operating system kernel and found that a synchronization had been implemented as a normal access only in one case.

In general, a computation only needs to insure the ordering of some of its actions in relation to a few other selected computations. This is very hard to express and very hard to implement in the general case, but possible in some special cases. For example, if the data that are accessed in a critical region are all stored in a cache line, the QOSB, Test and Set and Unset operations proposed for the Wisconsin Multitube [12] guarantee correct behavior without strictly ordering. This is due to the fact that these operations return the cache line upon successful locking. The semaphore is part of the line and a cache line is guaranteed to be coherent by the caching protocol.

Another form of weak ordering, called release consistency, has been proposed for the DASH multiprocessor [10]. Under this model only certain kinds of synchronization operations enforce a specific ordering, allowing for more flexibility than in the weak-ordering model.

PLUS provides three ways of improving the performance of synchronization operations. First, an explicit fence operation (see Section 2.3) is available to implement strong ordering of synchronization operations when necessary (as opposed to the more restrictive approach of always using strong ordering at synchronization time). Second, PLUS splits an atomic read-modify-write operation into an issue primitive and a verify primitive that reads the result; the execution of the operation can proceed concurrently with other computation. Finally, in contrast to other RISC systems, the set of read-modify-write operations comprises rather complicated operations.

### 2.2 Caching Mechanisms

In order to obtain good performance, a bus-based system has to maintain multiple copies of read/write data and minimize the number of write operations. In pursuing these goals, snooping protocols have evolved from simple write-through to the increasingly sophisticated write-once [11], write-invalidate [16] and write-update [17] protocols.

For example, the DRAGON [17] protocol achieves very good performance, because it keeps multiple copies of shared data even if the data are being written by more than one processor, thus avoiding the ping-ponging of shared data. As a consequence, it has far fewer write misses (fewer by a factor of six, compared to a simple write-through-with-invalidate protocol, for one of the applications evaluated in [1]). The DRAGON protocol broadcasts an update only if other caches indicate (with a bus signal) that the line is shared, as opposed to simpler protocols that either blindly broadcast every write or blindly invalidate all copies on the first write.

Protocols that use fewer write operations are beneficial in distributed-memory bus systems, because they use less memory and bus bandwidth. In a distributed-memory non-bus-based system both memory and communication bandwidth grow with the number of processors, so it is not always necessary for a protocol to minimize the frequency of writes.

Moreover, since latency in moving data is much larger in distributed-memory systems than in bus-based systems, using a protocol that does not invalidate other copies, but instead updates them, is very useful in minimizing the cost of cache misses.

PLUS uses a write-update protocol in which all writes to shared data are propagated to all the copies. The latency of write operations does not stall the processor, because multicast operations to update the copies are carried out independently by dedicated hardware.

### 2.3 PLUS's Coherence and Caching Mechanism

PLUS caching mechanism is an extension of the mechanisms described in [2,3]. Each PLUS node (see Figure 2-1) contains one processor with its cache, some local memory and a memory-coherence manager that is
linked to other nodes through a fast interconnection network. The local memory is used both as main memory and as a cache for data in other nodes' memories. Note that any data in the local memory may also be cached in the processor cache. In order to avoid confusion, we use the terminology replicated data to mean data stored in more than one node's local memory, and cached data to mean data stored in the processor cache.

Figure 2-1: A PLUS Node.

PLUS uses a non-demand, write-update coherence protocol for the replicated data. The unit of replication is a page (whose size is dictated by the memory management system of the off-the-shelf CPU, 4 Kbytes in the current implementation). However, the unit of memory access and coherence maintenance is one (32-bit) word. Pages are replicated at the request of software, but the coherence manager hardware is aware of this replication and automatically keeps copies coherent. The rest of this Section describes the replicated memory structure and the implementation of coherent read and write operations.

A virtual page corresponds to a list of physical pages replicated on different nodes. The first item of this list is called the master copy. (An unreplicated page only has the master copy). A node maps each virtual page to the most convenient physical copy, i.e. the closest copy. The global address of a physical page is a <node-id, page-id> pair and is generated directly by the memory-mapping mechanism of the processor.

The operating system kernel orders the copy-list to minimize the network path length through all the nodes in the list. On each node, the replication structure is made visible to the coherence manager via the master and nextcopy tables, which are maintained by the operating system. For each locally replicated physical page, the master table identifies the global physical page address of the master copy, and the next-copy table identifies the successor, if any, of the local copy along the copy-list.

Read operations are implemented as follows. The node-id field of the translated physical address determines which node is addressed, and the page-id field specifies the page within that node. If the local node is indicated, the local memory is read. Otherwise, the coherence manager sends a read request to its counterpart in the specified remote node, waits for the response, and passes the returned data to the processor.

Write operations are more complicated, because they must take effect on every copy. Writes are always performed first on the master copy and then propagated down the ordered copy-list. This insures that all copies eventually contain the same data when all writes issued by all processors have completed. (This property is called general coherence [18]). The coherence manager handles writes as described below.

If the physical address indicates a remote node, the coherence manager sends a write request to that node (note that the remote node might not be the master). Otherwise, it checks the master table to determine the master-copy location. If the master copy is local, it carries out the write on the local memory, and sends an update request to the next copy, if any (determined by looking up the next-copy table). If the master copy is not local, it sends a write request to the node that has the master copy.

A coherence manager, that receives a write request, also goes through the process of making sure the write begins at the master copy. A coherence manager, that receives an update request, updates its memory and sends another update request to the next copy, if any. Finally, the last copy in the copy-list returns an acknowledge to the processor that originated the write operation, thus completing that operation. General coherence is guaranteed since copies of a given location are always written in the same order.

Write operations do not block the issuing processor while they propagate through the copies and a processor can issue several writes before blocking. However, reading a location that is currently being written blocks until the write completes. This is achieved by remembering the address of incomplete write operations in the pending-writes cache of the coherence manager and guarantees strong ordering within a single processor independently of replication (in the absence of concurrent writes by other processors). There is no such guarantee with respect to another processor.
particular, actions by one processor may be observed out of order by other processors. When strong ordering is necessary between processors, e.g., for correct synchronization, a processor must use a fence operation, which causes the coherence manager to block any subsequent write by the processor, until all its earlier ones have completed (this is implemented by waiting until the pending-writes cache is empty). The processor can then proceed with the synchronization operation. A similar technique was proposed for the RP3 [6] multiprocessor. Three kinds of fence operations have been described in [10]: write-fences that block all subsequent writes, full fences that block all subsequent reads and writes, immediate fences that block only the next operation. PLUS implements explicit write fences that wait for all previous writes. A more aggressive implementation could use immediate fences. PLUS does not enforce full fences as part of synchronization operations, as in DASH. Instead, the user can explicitly issue the fence operation when appropriate (see Section 3-1).

Note that contents of a processor’s local memory can be cached in the processor’s primary cache. In order for writes to eventually propagate to all copies, all writes by a processor must be visible to its coherence manager. Hence, replicated pages must be cached with a write-through policy. At the remote site, a snooping protocol on the node bus ensures coherency between memory and cache whenever the coherence manager carries out a write or update operation.

2.4 Memory Mapping.
Since PLUS executes one multithreaded process at a time, all nodes use the same virtual memory space, and, because of replication, different nodes might map the same virtual page to different physical copies. Although the page tables could be fully shared, it is more efficient if each node maintains its own page tables. These tables do not have to contain all the possible mappings but only those that are actively used by the node. If a node accesses a page that is not mapped by the local page tables, the exception handler checks in a centralized table if the mapping is legal and then updates the local tables. This lazy evaluation of page tables limits the amount of interference caused by dynamic memory allocation and requires less memory than if the tables were fully replicated.

Software is responsible for page placement and replication policies, but the hardware helps in performing them. Deleting a copy is akin to removing a page in a paging operating system, since all the nodes that have a copy of the page must update their address translation tables and flush their TLBs. Replicating a page can be done almost entirely as a background activity: first the new entry is added to the copy-list at a convenient point, and then the hardware copies the data from the previous copy. (Note that the copy operation can be overlapped with writes to the same page by any processor in the system, without destroying the page integrity.) When the new page has been fully written, each node can update the address translation tables to use the new copy. Page migration is achieved simply by creating a copy and then deleting the old one.

Page replication and migration can be used in three ways, possibly at the same time:
- If the access pattern is known to the programmer, it is possible to request a memory layout that minimizes network traffic and latency (for example by means of language-level pragmas).
- If the access pattern is not data dependent, it can be measured during one run of the application and the results of the measurement used to optimally allocate memory in subsequent runs.
- If the access pattern is unknown, it is possible to use competitive algorithms [5] that try to optimize memory references by replicating or migrating pages. The basic idea is to keep track of remote references, and, when the cumulative cost of remote references to a page exceeds the cost of creating a page copy, actually create a copy locally. In PLUS, competitive algorithms are supported by hardware that counts the number of references from each processor to each page and interrupts the node processor if any counter overflows.

Stack and code areas are usually not replicated, and can be kept in local physical memory.

2.5 Evaluation of Replication.
The performance of PLUS depends critically on the application and there is currently no set of parallel benchmarks that cover a wide spectrum of applications. In order to evaluate the design before starting the implementation, we have used a few applications that we knew very well and that stress the characteristics of the architecture. These include a production system applications, a shortest-path program, and a speech recognition system. We also carried out some experiments with synthetic loads as reported in [2]. We built a PLUS simulator that is driven by an application program in C language. A library package provides functions to create simulated shared memory and to allocate it on the nodes specified by the user. When the program reads or writes data allocated in shared memory, the simulator emulates the actions of the coherence manager and the network. Caching, coherence management, routing and memory access are simulated and instrumented in detail. From the instruction stream, the simulator also computes an approximate estimate of execution time between simulated shared memory references. Unfortunately, such a detailed simulation cannot be performed for systems larger than about 100 processors, because of the time and memory space needed.

The Single Point Shortest Path problem is a good example that requires many synchronization operations.
The problem involves finding the minimum cost to traverse a graph from one vertex to any other vertex. Both sequential and concurrent algorithms for this problem work by propagating the distance cost from one vertex and updating it until no more updates are possible.

Table 2-1: Effect of Replication on Messages.

<table>
<thead>
<tr>
<th>Number of Copies</th>
<th>Reads Local/Remote</th>
<th>Writes Local/Remote</th>
<th>Ratio Total/Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.25</td>
<td>3.40</td>
<td>6.18</td>
</tr>
<tr>
<td>2</td>
<td>1.70</td>
<td>1.18</td>
<td>2.91</td>
</tr>
<tr>
<td>3</td>
<td>1.64</td>
<td>0.70</td>
<td>2.24</td>
</tr>
<tr>
<td>4</td>
<td>2.14</td>
<td>0.45</td>
<td>1.89</td>
</tr>
<tr>
<td>5</td>
<td>2.32</td>
<td>0.36</td>
<td>1.68</td>
</tr>
</tbody>
</table>

Our implementation uses multiple queues since, owing to queue bandwidth limitation, a single queue introduces serialization and requires long remote accesses. The vertices are evenly distributed among the nodes and there is one queue associated with each node. If a processor extracted work only from its queue, some processors would remain idle for part of the time, especially if the ratio of number of vertices to the number of processors is low. For a better load balance each processor must extract work from other queues when its local queue is empty. The shared memory model and the ability to replicate data are very helpful in this case. We have replicated the queues and vertices on more than one processor and found a substantial performance increase due to better load balancing. Figure 2-1 shows the efficiency of the algorithm and the utilization, i.e. ratio of average useful processor time to elapsed time for different levels of replication. With no replication, the utilization decreases substantially when more than 2 processors are used; while with replication it remains high until the number of processors exceeds 32. When more than 32 processors are used, most processors are idle waiting for work, since the problem is not large enough to occupy all processors.

Table 2-1 shows how the ratio of local to remote operations changes with replication in the 16-processor case of Figure 2-1. An increase in replication causes a drop in the number of remote reads and an increase in the number of remote writes and updates. Such a trade-
off is usually beneficial because of the overall decrease in remote read latency.

As replication increases, so does the total number of network messages (last column in Table 2-1) and a larger percentage of them is used to update copies. In this application there were no bad consequences, because the network was only lightly loaded. In general, however, uncontrolled replication can result in the system getting flooded with update requests, slowing down useful computation.

3. Synchronization.
Caching is only marginally useful in improving the latency of synchronization operations, since these operations always involve competitive and exclusive accesses to a variable.

For example, synchronization operations can severely degrade the performance of a system that uses a snooping coherence protocol. Constructs such as test-and-test-and-set were invented to minimize the overhead caused by the interference between the coherence protocol and the synchronization operations.

3.1 PLUS's Delayed Operations.
PLUS provides several variants of interlocked read-modify-write memory operations. Like writes, these operations take effect at all copies of the addressed location, beginning with the master and propagating down the copy-list. However, the master, in addition to executing the operation atomically and forwarding update requests to the next copy, also returns the old contents of memory to the originating node. Since this result always has to come from the master copy, there can be a substantial delay between the initiation of the operation and the availability of its result.

PLUS allows the user or the compiler to hide this latency by separating the initiation of an operation from the checking of its result, as described in [2].

We call these operations delayed operations, since the execution of the operation overlaps regular processing, as is the case of delayed branches. The processor can continue with normal instruction execution in the meantime. PLUS also lets a processor have more than one delayed operation in progress at any time (8 in the current implementation), thereby further reducing their average latency.

In PLUS, synchronization instructions return an identifier that the program can later use to retrieve the result of the operation. This identifier is simply the address of a location in the delayed-operations cache. This location is automatically allocated when the processor executes a delayed operation and deallocated when it reads the result. If the result is not available when the processor reads it, the read blocks (since the software can inspect the status of these locations, it is also possible to implement a non-blocking read).

There is no implicit fence operation associated with these delayed operations. Instead, there is a separate, explicit fence operation available to the programmer. It is the responsibility of the programmer or the compiler to use these primitives to implement synchronization correctly. This scheme leaves substantial room for speed improvement through code scheduling and selective use of the fence operation. (For instance, there is usually no need to issue a fence before a P operation).

The delayed operations available in the current implementation of PLUS are described in Table 3-1. The cost of a delayed operation comprises three components: the time taken by the processor to issue the operation, the

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Execution cycles by the coherence manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>xchng</td>
<td>Return current value and write 30-bit unsigned word.</td>
<td>39</td>
</tr>
<tr>
<td>cond-xchng</td>
<td>Return current value of memory. If top bit set, write 30-bit unsigned word.</td>
<td>39</td>
</tr>
<tr>
<td>fetch-and-add</td>
<td>Return current value of memory. Increment memory by given signed word.</td>
<td>39</td>
</tr>
<tr>
<td>fetch-and-set</td>
<td>Return the current value and set top bit.</td>
<td>39</td>
</tr>
<tr>
<td>queue</td>
<td>(Addressed location contains offset in addressed page to tail of queue.) Return current word at tail. If top bit of tail clear, write given word there, set top bit, and increment the offset (modulo maximum queue size) to next word in queue.</td>
<td>52</td>
</tr>
<tr>
<td>dequeue</td>
<td>(Addressed location contains offset in addressed page to head of queue.) Return current word at head. If top bit of head set, clear it and increment the offset (modulo maximum queue size) to next word in queue.</td>
<td>52</td>
</tr>
<tr>
<td>min-xchng</td>
<td>Return current value, store given value if smaller than the original value.</td>
<td>52</td>
</tr>
<tr>
<td>delayed-read</td>
<td>Return current value, no modification</td>
<td>39</td>
</tr>
</tbody>
</table>

Table 3-1: PLUS's Delayed Operations.
time taken by the coherence manager to execute the operation and obtain the result (the processor is not involved in this), and the time taken by the processor to read the result. The first step takes approximately 25 cycles (each cycle in the current PLUS implementation takes 40ns). The time to perform the second step can be divided into communication time and processing time. The round trip communication time between two adjacent nodes is about 24 cycles; if the nodes are not adjacent each extra hop adds 4 cycles (these numbers were measured on the router used in the current implementation). The processing time depends on the operation; an estimate based on the detailed hardware design is shown in Table 3-1. The last step, the processor reading the result, takes about 10 cycles, assuming the result is available. As a comparison, the cost of a remote (blocking) read is about 32 cycles plus the round-trip network delay.

The last step, the processor reading the result, takes about 10 cycles, assuming the result is available. As a comparison, the cost of a remote (blocking) read is about 32 cycles plus the round-trip network delay.

Table 3-2: Lock with Queue.

| LOCK: if (fadd (lock, 1) != 0) { /* lock unavailable, queue myself for obtaining lock */
| while (queue (QP, myID) & 0x80000000); /* spin if queue is full, unlikely */
| wait(); /* go to sleep until someone wakes me up and gives me the lock */
| } /* my thread has lock */

| UNLOCK: if (fadd (lock, -1) > 1) {/* some other thread waiting for lock, pop its ID from queue */
| while (! (k = dequeue (DQP)) & 0x80000000)); /* loop if queue is empty */
| wake_up (k &= 0x7fffffff); /* k == ID of next process in queue, wake it up; */
| } /* thread k now has lock */

2.3 Complex is Better.
It is often argued that simpler operations are better: the implementation is simpler and faster and the user has an easier time mastering them. In principle, many existing primitives like test-and-set and fetch-and-add are, alone, sufficient to implement all synchronization operations. (See [12] for a summary of state-of-the-art synchronization primitives). However, we should keep in mind that speed is the main concern in synchronization and not ease of use. Hardware synchronization primitives should not be used directly by users and should be either encapsulated in higher level constructs or directly generated and optimized by a compiler.

If we look at the implementation of queue operations with a fetch-and-add (see the paper by Gottlieb et.al.[13], Appendix) we see that, although a single kind of primitive is used (fetch-and-add), it must be used three times to correctly perform a queuing operation. As we saw before, there is a substantial delay until the result of each synchronization operation is available. If we execute a number of synchronization primitives in short succession, the issuing processor will be idle most of the time, and hiding this latency will be hard.

Performance can be improved if more powerful primitives such as the queue and dequeue operations in Table 3-1 are used. These primitives reduce the total number of read-modify-write operations compared to queues implemented with simpler primitives. We believe that a reasonable set of complicated synchronization operations is a better choice for distributed-memory systems. We evaluated different sets of synchronization operations by writing various algorithms and measuring their performance on our simulator. The

/* lock variable "lock" initialized to 0, i.e. busy, variables QP and DQP contain offsets within page to tail and head of queue, QP and DQP initialized to 0 and all queue words initially empty (top bit clear) */

LOCK: if (fadd (lock, 1) != 0) { /* lock unavailable, queue myself for obtaining lock */
while (queue (QP, myID) & 0x80000000); /* spin if queue is full, unlikely */
wait(); /* go to sleep until someone wakes me up and gives me the lock */
} /* my thread has lock */

UNLOCK: if (fadd (lock, -1) > 1) {/* some other thread waiting for lock, pop its ID from queue */
while (! (k = dequeue (DQP)) & 0x80000000)); /* loop if queue is empty */
wake_up (k &= 0x7fffffff); /* k == ID of next process in queue, wake it up; */
} /* thread k now has lock */
approximation of the minimum or maximum value of some variable.

- The delayed-read operation is like an ordinary read, except that it proceeds asynchronously and the result can be retrieved later. Since several such operations can be in progress simultaneously, this is useful for hiding the latency of remote read operations. However, it needs careful, handcrafted code or a clever optimizing compiler. We do not currently have such a compiler.

### 3.3 Software Pipelining and Context Switching

Delayed synchronization allows two latency-avoidance techniques: software pipelining and context switching.

Software pipelining is typically implemented by the compiler but, in some cases, it can be exploited directly by the programmer. For example, we programmed a primitive that returns a pointer to a free element in a queue with very little latency, because it eagerly asks for a new element every time the user consumes the previous element (the first time it is called, it retrieves two elements). In another case we have been able to issue a number of lock requests in advance, so that part of the latency was absorbed by the computation.

Although we do not have a compiler that takes advantage of delayed synchronization, we have been able to make good use of delayed synchronization in many cases.

Context switching is an extremely attractive way of hiding latency and a few systems based on fast context switching have been built [14] or proposed [19]. The usefulness of context switching depends mostly on the ratio between the context switch time and the time between context switches. If a context switch were to cost only a few processor cycles, it would solve all latency problems, and could be used whenever remote memory is accessed. Unfortunately, this is not possible with off-the-shelf processors.

![Efficiency of the Beam Search Application with Different Synchronization Costs](image)

**Figure 3-1:** Efficiency of the Beam Search Application with Different Synchronization Costs.

### 3.4 Evaluation of Delayed Operations

We show the performance of a beam search algorithm that searches a Hidden Markov Model representation of the speech process (a directed graph) and returns the most likely sequence of words. Beam search requires a very fine-grain parallel decomposition and a substantial amount of synchronization. Typically, a processor must dequeue one vertex from the list of vertices to be processed, lock all the vertices that follow it and finally queue a new vertex. This inner loop can be coded in about 70 RISC instructions and requires about 10 memory references per iteration, which cause about 3 cache misses if the cache line contains four words.
algorithm has spatial locality but almost no temporal locality.) Queue operations on a central queue cause too much serialization, owing to bandwidth limitation at the queue. As with the shortest-path problem, the queue is split into local queues, one at each processor, to avoid this bottleneck. In this case, because of the highly data-dependent behavior of beam search, it is likely that some queues will become empty before others and some processors will remain idle and create a load imbalance. This load imbalance can be overcome by sharing a queue among a number of processors instead of keeping them all disjoint.

Figure 3-1 compares the performance with different context switching overheads and with delayed operations. The blocking synchronization curve has been computed by running a program that waits for synchronization primitives to return a result before proceeding. The delayed operations curve has been computed by explicitly programming the pipelining of synchronization operations:

- the next vertex is dequeued in parallel with the processing of the current state;
- the locking of all next vertices is performed in parallel.

The programming burden of these changes was easily hidden in two macros, so the code is not very different from the blocking-synchronization case. The context-switch curves were computed by simulating a context switch every time a synchronization operation was issued. The cost of the context switch was set to 16,40 and 140 processor cycles (the curves in Figure 3-1 are labeled with this cost).

As expected, very fast context switching has the best performance but delayed operations are more effective than a context switching mechanism with a 40-cycle overhead. To put things in perspective, a state-of-the-art RISC processor might need to save and restore about 15 registers. If this operation can be performed entirely in cache, a context switch would cost about 40 cycles (including the instructions necessary to decide which is the new context). If the processor misses in cache, it would cost about 140 cycles because of the bus and memory latency (we are assuming a four-word line fetch takes 15 cycles). Future commercial processors might include wide busses between registers and local memory. In this case the save/restore operation could be completed in fewer cycles and the performance of the top curve of Figure 3-1 might become possible.

The results in Figure 3-1 are more pessimistic than those reported by Weber and Gupta in [19]. This is probably due to two factors. First, our application has a very short inner loop. Second, our assumptions about network delay and interface overhead (sending and receiving a message) are more conservative than theirs.

4. Related Work.
PLUS represents a specific trade-off in the space of distributed-memory architectures that range from large-granularity LAN-based machines, to message-passing machines, to hierarchical-bus machines to full-fledged directory-based shared-memory machines.

Operating system researchers have devised techniques to implement shared memory across distributed systems, for example by means of shared memory servers [9]. The problem with these solutions is that, regardless of network and processor speed, they result in large software overhead because the basic mechanism is paging. Faster networks will improve the performance of these systems to the point where the physical transfer of a page will take a negligible amount of time but the software overhead (a few milliseconds on one-VAX-MIP machines) will remain. Of course the usability of such systems depends heavily on the application.

Intelligent message coprocessors that relieve the main processor of the task of sending a message are now common in message passing machines. Nevertheless, the overhead to send and receive a message is still larger than 10 microseconds for state-of-the-art systems. Most of this overhead is again due to software. PLUS also uses hardware to interface processor and network, but the hardware does not require any software interface, e.g. a send function, because it is triggered directly by a memory reference.

Multiple-bus systems like the Wisconsin Multicube and Encore’s extension of the Multimax, which employ an extended form of snoopy caching, and directory-based systems like DASH and the proposed SCI standard are all true shared-memory systems, and all employ sophisticated caches and cache-coherence protocols. PLUS relies on software-controlled non-demand replication of pages to achieve the same goals without the high hardware cost. We believe that in many single-user applications PLUS will perform as well as any of these machines.

5. Current Implementation.
The current implementation of PLUS uses a general purpose Motorola 88000 processor (25 MHz) with 32 Kbytes of cache and 8 or 32 Mbytes of main memory at each node. The memory is organized in two interleaved banks to sustain the burst bandwidth needed for cache line accesses. Global memory mapping, coherence management and atomic operations are performed by a hardware module that is implemented with Xilinx PLD’s and PAL’s. (It is possible to implement this module in a single ASIC device.) In this implementation, each node can have up to 8 writes and 8 delayed operations in progress. The interconnection network uses a mesh router designed at Caltech [7]. Each router has five pairs of I/O links: one for the processor and one for each of its mesh neighbors. Links operate at 20 Mbyte/second in each direction. SCSI devices, audio peripherals and host computers can be attached to each node. The implementation of PLUS is at an advanced stage. A one-node prototype has been running since November 1989,
and we expect to have a working multinode system in the Summer of 1990.

Acknowledgments.
Some ideas were originated by discussions with Lawrence Butcher and Andreas Nowatzyk. Raj Reddy and Duane Adams have been, as always, instrumental to the survival of this project. We are also extremely grateful to George White of Apple Computer, Inc. for his support. The first prototype was built with the help of John Figueroa of On Target Associates.

References