How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs

LESLIE LAMPORT

Abstract—**Many large sequential computers execute operations in a different order than is specified by the program. A correct execution is achieved if the results produced are the same as would be produced by executing the program steps in order. For a multiprocessor computer, such a correct execution by each processor does not guarantee the correct execution of the entire program. Additional conditions are given which do guarantee that a computer correctly executes multiprocess programs.**

Index Terms—**Computer design, concurrent computing, hardware correctness, multiprocessing, parallel processing.**

A high-speed processor may execute operations in a different order than is specified by the program. The correctness of the execution is guaranteed if the processor satisfies the following condition: the result of an execution is the same as if the operations had been executed in the order specified by the program. A processor satisfying this condition will be called *sequential*. Consider a computer composed of several such processors accessing a common memory. The customary approach to designing and proving the correctness of multiprocess algorithms [1]–[3] for such a computer assumes that the following condition is satisfied: the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program. A multiprocessor satisfying this condition will be called *sequentially consistent*. The sequentiality of each individual processor does not guarantee that the multiprocessor computer is sequentially consistent. In this brief note, we describe a method of interconnecting sequential processors with memory modules that insures the sequential consistency of the resulting multiprocessor.

Manuscript received September 28, 1977; revised May 8, 1979.

The author is with the Computer Science Laboratory, SRI International, Merlo Park, CA 94025.

We assume that the computer consists of a collection of processors and memory modules, and that the processors communicate with one another only through the memory modules. (Any special communication registers may be regarded as separate memory modules.) The only processor operations that concern us are the operations of sending fetch and store requests to memory modules. We assume that each processor issues a sequence of such requests. (It must sometimes wait for requests to be executed, but that does not concern us.)

We illustrate the problem by considering a simple two-process mutual exclusion protocol. Each process contains a *critical section*, and the purpose of the protocol is to insure that only one process may be executing its critical section at any time. The protocol is as follows.

```
process 1
    a := 1;
    if b = 0 then critical section;
                    a := 0
                else  ···  fi

process 2
    b := 1;
    if a = 0 then critical section;
                    b := 0
                else  ···  fi
```

The else clauses contain some mechanism for guaranteeing eventual access to the critical section, but that is irrelevant to the discussion. It is easy to prove that this protocol guarantees mutually exclusive access to the critical sections. (Devising a proof provides a nice exercise in using the assertional techniques of [2] and [3], and is left to the reader.) Hence, when this two-process program is executed by a sequentially consistent multiprocessor computer, the two processors cannot both be executing their critical sections at the same time.

We first observe that a sequential processor could execute the "b := 1" and "fetch b" operations of process 1 in either order. (When process 1's program is considered by itself, it does not matter in which order these two operations are performed.) However, it is easy to see that executing the "fetch b" operation first can lead to an error—both processes could then execute their critical sections at the same time. This immediately suggests our first requirement for a multiprocessor computer.

Requirement R1: Each processor issues memory requests in the order specified by its program.

Satisfying Requirement R1 is complicated by the fact that storing a value is possible only after the value has been computed. A processor will often be ready to issue a memory fetch request before it knows the value to be stored by a preceding store request. To minimize waiting, the processor can issue the store request to the memory module without specifying the value to be stored. Of course, the store request cannot actually be executed by the memory module until it receives the value to be stored.

Requirement R1 is not sufficient to guarantee correct execution. To see this, suppose that each memory module has several ports, and each port services one processor (or I/O channel). Let the values of "a" and "b" be stored in separate memory modules, and consider the following sequence of events.

1) Processor 1 sends the "a := 1" request to its port in memory module 1. The module is currently busy executing an operation for some other processor (or I/O channel).
2) Processor 1 sends the "fetch b" request to its port in memory module 2. The module is free, and execution is begun.
3) Processor 2 sends its "b := 1" request to memory module 2. This request will be executed after processor 1's "fetch b" request is completed.
4) Processor 2 sends its "fetch a" request to its port in memory module 1. The module is still busy.

There are now two operations waiting to be performed by memory module 1. If processor 2's "fetch a" operation is performed first, then both processes can enter their critical sections at the same time, and the protocol fails. This could happen if the memory module uses a round robin scheduling discipline in servicing its ports.

In this situation, an error occurs only if the two requests to memory module 1 are not executed in the same order in which they were received. This suggests the following requirement.

Requirement R2: Memory requests from all processors issued to an individual memory module are serviced from a single FIFO queue. Issuing a memory request consists of entering the request on this queue.

Condition R1 implies that a processor may not issue any further memory requests until after its current request has been entered on the queue. Hence, it must wait if the queue is full. If two or more processors are trying to enter requests in the queue at the same time, then it does not matter in which order they are serviced.

Note. If a fetch requests the contents of a memory location for which there is already a write request on the queue, then the fetch need not be entered on the queue. It may simply return the value from the last such write request on the queue. □

Requirements R1 and R2 insure that if the individual processors are sequential, then the entire multiprocessor computer is sequentially consistent. To demonstrate this, one first introduces a relation $\rightarrow$ on memory requests as follows. Define $A \rightarrow B$ if and only if 1) A and B are issued by the same processor and A is issued before B, or 2) A and B are issued to the same memory module, and A is entered in the queue before B (and is thus executed before B). It is easy to see that R1 and R2 imply that $\rightarrow$ is a partial ordering on the set of memory requests. Using the sequentiality of each processor, one can then prove the following result: each fetch and store operation fetches or stores the same value as if all the operations were executed sequentially in any order such that $A \rightarrow B$ implies that A is executed before B. This in turn proves the sequential consistency of the multiprocessor computer.

Requirement R2 states that a memory module's request queue must be serviced in a FIFO order. This implies that the memory module must remain idle if the request at the head of its queue is a store request for which the value to be stored has not yet been received. Condition R2 can be weakened to allow the memory module to service other requests in this situation. We need only require that all requests *to the same memory cell* be serviced in the order that they appear in the queue. Requests to different memory cells may be serviced out of order. Sequential consistency is preserved because such a service policy is logically equivalent to considering each memory cell to be a separate memory module with its own request queue. (The fact that these modules may share some hardware affects the rate at which they service requests and the capacity of their queues, but it does not affect the logical property of sequential consistency.)

The requirements needed to guarantee sequential consistency rule out some techniques which can be used to speed up individual sequential processors. For some applications, achieving sequential consistency may not be worth the price of slowing down the processors. In this case, one must be aware that conventional methods for designing multiprocess algorithms cannot be relied upon to produce correctly executing programs. Protocols for synchronizing the processors must be designed at the lowest level of the machine instruction code, and verifying their correctness becomes a monumental task.

References

[1] E. W. Dijkstra, "Hierarchical ordering of sequential processes," *Acta Informatica*, vol. 1, pp. 115–138, 1971.
[2] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 125–143, Mar. 1977.
[3] S. Owicki and D. Gries, "Verifying properties of parallel programs: an axiomatic approach," *Commun. Assoc. Comput. Mach.*, vol. 19, pp. 279–285, May 1976.

0018-9340/79/0900-0691$00.75 © 1979 IEEE