Efficient and Correct Execution of Parallel Programs that Share Memory

DENNIS SHASHA
Courant Institute of Mathematical Sciences, New York University
and
MARC SNIR
IBM T. J. Watson Research Center

In this paper we consider an optimization problem that arises in the execution of parallel programs on shared-memory multiple-instruction-stream, multiple-data-stream (MIMD) computers. A program on such machines consists of many sequential program segments, each executed by a single processor. These segments interact as they access shared variables. Access to memory is asynchronous, and memory accesses are not necessarily executed in the order they were issued. An execution is correct if it is sequentially consistent: It should seem as if all the instructions were executed sequentially, in an order obtained by interleaving the instruction streams of the processors. Sequential consistency can be enforced by delaying each access to shared memory until the previous access of the same processor has terminated. For performance reasons, however, we want to allow several accesses by the same processor to proceed concurrently. Our analysis finds a minimal set of delays that enforces sequential consistency. The analysis extends to interprocessor synchronization constraints and to code where blocks of operations have to execute atomically. We use a conflict graph similar to that used to schedule transactions in distributed databases. Our graph incorporates the order on operations given by the program text, enabling us to do without locks even when database conflict graphs would suggest that locks are necessary. Our work has implications for the design of multiprocessors; it offers new compiler optimization techniques for parallel languages that support shared variables.

Categories and Subject Descriptors: C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*multiple-instruction-stream, multiple-data-stream processors (MIMD)*; D.1.3 [**Programming Techniques**]: Concurrent Programming; D.3.4 [**Programming Languages**]: Processors—*optimization*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages

General Terms: Design, Languages, Performance, Theory

Additional Key Words and Phrases: Networks, parallel languages, PRAM, shared memory, shared variables

1. INTRODUCTION

There is a well-established programming paradigm for serial computers: A stream of instructions is executed serially; the execution of each instruction is atomic and terminates before the execution of the succeeding instruction starts. In

The work of Shasha was partially supported by the National Science Foundation under grant DCR8501611 and by the Office of Naval Research under grant N00014-85-K-0046. Part of the work of Snir was done while he was at the Courant Institute.
Authors' addresses: D. Shasha, Courant Institute, New York University, New York, NY 10012; and M. Snir, IBM T. J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598.

Segment 1	*Segment 2*
test&set1(LOCK)	test&set2(LOCK)
read1(X)	read2(X)
write1(X)	write2(X)
reset1(LOCK)	reset2(LOCK)

Fig. 1. Serialization routines.

practice, the execution of several succeeding instructions may overlap in time. In particular, memory accesses may be pipelined, in order to overcome the high memory latency. In some machines it is even possible for accesses to occur in memory in an order that is different from the order they were issued by the processor. This concurrency is hidden from the user; he or she should not be aware that memory is accessed out of order. The control logic of the machine achieves this by enforcing an execution order that respects data dependencies: An instruction that uses a value returned from memory is delayed until this value is available; successive accesses to the same memory location occur in the order they were issued.

A shared-memory MIMD computer, such as the NYU Ultracomputer [9] or the IBM RP3 [16], consists of a set of processors connected to a shared memory. Each processor executes independently a (possibly distinct) serial program. Some of the instructions in these programs may be for load, store, or read-modify-write operations that access a shared-memory location. We call the code executed by each processor a *sequential program segment*; the union of all these segments is the parallel code executed by the machine as a whole.

The programming paradigm for such machines is provided by the *interleaving semantics* for parallel code:

> The outcome of an execution of a parallel code is as if all the instructions were executed sequentially and atomically. Instructions in the same program segment are executed in the order specified by this segment; the order of execution of instructions belonging to distinct segments is arbitrary [12, 14].

That is, the outcome of an execution is as if the instructions were all executed in an order obtained by arbitrarily interleaving the streams produced by the distinct program segments. We follow [12] and [14] in calling this condition *sequential consistency.*

In order to enforce sequential consistency, it is not sufficient to consider data dependencies within each program segment; *interdependencies* have to be taken into account as well. Suppose two program segments in a parallel program update a variable X, but serialize their accesses using locks.[1] The serialization routines are written as shown in Figure 1. The reset instruction is just a regular store. There is no data dependency between the write1(X) and the reset1(LOCK) instructions in the first program segment. Suppose these two memory accesses are allowed to occur out of order. Then the memory accesses of the parallel

[1] Our theory encompasses cases where access to shared variables need not be serialized as well.

program may occur in memory in the following order:

test&set1(LOCK) read1(X) reset1(LOCK) test&set2(LOCK) read2(X) write1(X) write2(X) reset2(LOCK).

The serialization failed, since both processors read the original value of X and only the update performed by the second processor is reflected in the shared data. This execution is not sequentially consistent. In a sequentially consistent execution, write1(X) should seem to occur before reset1(LOCK); this is not true of the execution above. Note that each operation was executed correctly in memory, and the instructions within each program segment were executed in an order that preserves data dependencies within the segment (test&set(LOCK) occurs in memory before reset(LOCK), and read(X) before write(X)).

Such hazards can be prevented by enforcing that accesses occur in memory in the order they are issued by the processors. One approach is that a processor does not initiate a new access to shared memory before the previous access has terminated. (This can be easily enforced if each memory request, including stores, returns a reply.) Another approach, suggested by Lamport [12], is that all memory requests are handled by one memory controller; the controller receives requests one at a time and processes them in FIFO order.

The shared memory of the NYU Ultracomputer and IBM RP3 machines consists of a set of memory modules; a packet switched multistage interconnection network connects processors to memory modules. A processor need not wait for a reply to its previous request to shared memory before issuing a new one, and several memory requests issued by the same processor may simultaneously proceed through the network.

Requests that arrive simultaneously to the same memory module conflict; further conflicts may occur in the network. These conflicts delay requests by unpredictable amounts. As a result, requests issued by a processor to distinct memory modules may execute out of order. Similarly, a memory module may execute a request m before it executes a request m', even though m' was issued first (m and m' are issued by different processors). Thus, the NYU Ultracomputer and IBM RP3 machines do not conform to the programming paradigm provided by interleaving semantics; "incorrect" executions, such as illustrated in our previous example, may occur on them. The same situation is likely to occur with other large shared-memory MIMD machines. Pipelining of memory requests is required in order to mask the (relatively) large latency of the interconnection network, and the throughput is greatly reduced if all accesses are delayed whenever a conflict delays some access.

While the NYU Ultracomputer and IBM RP3 do not enforce sequential consistency in hardware, they both provide control mechanisms that allow them to do so in software. For example, the IBM RP3 has a *fence* instruction; the execution of this instruction by a processor causes it to wait until all outstanding references to shared memory have completed.

1.1 Our Goal

A parallel computer architecture that is not sequentially consistent is extremely hard to understand; it does not fit a programming paradigm that can be comfortably used by software or programmer. We postulate that parallel programs

are created for an idealized parallel architecture, where machine instructions are executed atomically (an access to shared memory involves only one word). Such programs may come from two sources: They may be directly written by users, when efficiency dictates a programming style close to the machine architecture; or they may be the intermediate output of a compiler that compiles code for a shared-memory parallel computer. We consider the next compilation phase that maps this intermediate code into the real architecture: Control instructions such as fences are added so that the resulting code will execute on the real machine with the same behavior as the source code has on the idealized machine. The control instructions delay the execution of some shared-memory accesses until previous accesses have terminated. Since delays slow the program down, we seek to minimize the number of delays enforced.

An analysis of interdependencies can reduce the number of delays used. A simple observation is that hazards can be due only to variables that are shared read-write (i.e., accessed by more than one program segment and modified by at least one such segment). It is sufficient to delay each access to a shared read-write variable until the previous access to a shared read-write variable by the same processor has completed [8]. This policy, however, is still not optimal. The first goal of our work is to determine the minimal set of delays that enforce sequential consistency.

The second part of our work considers cases where several memory accesses have to behave atomically. We assume code is produced for an idealized machine that has "high-level" atomic operations. For example, the language may provide atomic accesses and assignments to structured variables (such as arrays or complex numbers); the hardware guarantees atomicity of accesses only at the word level. When the idealized code is mapped on the real architecture, extra synchronization code—for example, to acquire and release locks—is needed. As synchronization code is expensive and locking reduces concurrency, we desire to use locking as parsimoniously as possible. The second goal of our work is to minimize both locks and delays for this more general case. Note that this work is relevant even for machines that conform to the model provided by interleaving semantics.

The generalization to multiaccess atomicity bears strong resemblance to database concurrency control [4–6], but surprisingly our solution requires far less locking than database concurrency control theory would lead one to expect. The reason is that we make strong use of knowledge that is not available to the concurrency control designer. Since our algorithms run as part of a compilation stage, we know what accesses each program segment makes. The concurrency control designer, by contrast, does not normally know the analogous information, that is, what transactions will run concurrently with a given transaction.

To see how this can make a difference, suppose there are three program segments: Segment 1 writes x and y atomically, segment 2 reads x and some private variables, and segment 3 reads y and some private variables. We can ensure the atomicity of the accesses without requiring any locks. A concurrency control designer, asked to design a locking protocol for a transaction that writes x and y atomically, would be forced to use locking, because he or she would not know whether there were concurrent transactions that might access both x and y.

This example used knowledge of the global set of accesses. We also use the known order of occurrence of these accesses (e.g., see the discussion of Figures 6 and 7 in Section 3).

In the next section, we develop a precise formalism to reason about serializability and atomicity of concurrent code. This formalism is strongly influenced by the work of Lamport [11, 13], and Lynch and Fisher [14]. In Section 3 we consider systems with no atomicity constraints; a minimal set of delays that enforce sequential consistency is found. This work is extended in Section 4 to restricted systems with atomicity constraints where delays are sufficient to enforce sequential consistency. In Section 5 general solutions using locking and delays are analyzed. In Section 6 we consider practical issues in the implementation of our code transformations, and further applications. Section 7 concludes this paper.

2. PRELIMINARIES

We consider the concurrent execution of a program on a parallel machine. The program consists of *instructions*, each specifying the execution of one *operation*. Every operation in the program accesses one or more storage locations, or *variables*. These may be memory locations, general-purpose registers, or special-purpose registers—for example, an instruction counter or status register. An access is a *write* if it updates the value of the variable accessed; it is a *read* otherwise. Operations communicate only by accessing the same variable. We assume the state of the machine is completely defined by the value of its variables; the effect of each operation is defined as a mapping from the (old) values of the variables read by the operation to the (new) values of the variables written by the operation (the two sets of variables may overlap). Each operation's access to a variable is *serialized* (i.e., two accesses to the same variable behave as if they occur serially in some order). This implies the atomicity of operations that make one access to a single location.

Two accesses to the same variable *conflict* if at least one is a write; two operations conflict if they execute conflicting accesses. (More restrictively, two accesses conflict if the final value of the variable accessed, or the values computed by the accessing instructions may change when the order of accesses is reversed; two update accesses that commute, such as "increment counter," do not conflict, even though both are writes.) The *execution order* specifies the order in which conflicting accesses are executed. This order determines the behavior of the computation: Any sequential execution of the accesses in an order that extends the execution order (i.e., a topological sort of the execution order) exhibits the same behavior. Conflicting accesses are executed in the same relative order in any such sequence, so that the same final state is reached by all these sequences.

Operations should appear to execute atomically: for any two operations u and v, either all variable accesses of u appear to occur before any access of v, or vice versa. Hence, the effect of an execution should be as if the instructions were executed in some sequential order, with variable accesses of an instruction starting only after all variable accesses of the previous instruction have terminated. In addition, the program prescribes an order for the instructions (called the *program order*). We assume this order is fixed; there is no data-dependent branch of control (later we shall remove this restriction). For a parallel machine,

the operations on each processor are ordered sequentially, but there is no restriction on the order of operations executed by distinct processors. The program order, therefore, is the union of several disjoint chains. More general program orders may be used to represent synchronization constraints across processors; our results are valid for arbitrary order relations.

The program order imposes restrictions on the execution order: If u precedes v in the program order, then the execution order should make it appear as if u executes all its storage accesses before v executes any of its accesses. A computation is *correct* if operations appear to execute atomically, in the order specified by the program order.

In addition, one may have restrictions on the order storage accesses are executed within an operation (e.g., operands are read before results are written back). Such restrictions usually follow from the semantics of the operation, and are often enforced by the computer architecture, as part of a correct implementation of these operations. A correct execution must also respect this ordering of accesses within operations.

2.1 Definitions

We use in the sequel the following definitions and results: A (*partial*) *order* is an irreflexive, asymmetric, transitive relation. Such a relation is represented by a directed acyclic graph (DAG) with the property that if a directed path connects node u to node v then uv is an edge of the graph. An order is *total*, or *linear*, if every two distinct elements are ordered.

In the following definitions, it may help to think of C as the conflict relation on accesses, of E as the execution order, of P as the program order, and as A as the atomicity constraint (uAv if $\{u, v\}$ should appear to execute atomically). We make this association explicit in Section 2.2.

Let C be a symmetric relation. The relation E is an *orientation* of C if whenever uCv then either uEv or vEu holds. The relation E is a *proper orientation* of C if E is an acyclic orientation of C.

Let P be an irreflexive relation and A be an equivalence relation on the same set U. P/A is the irreflexive relation induced by P on the family U/A of equivalence classes of A: $\mathbf{u}\ P/A\ \mathbf{v}$ if $\mathbf{u} \neq \mathbf{v}$ and there exist $u \in \mathbf{u}$, $v \in \mathbf{v}$ such that uPv. If P is transitive, then P/A is transitive. The relation P is *closed* under A if

$$uPv, \quad uAu', \quad vAv', \quad \neg uAv \text{ implies } v'Pv'.$$

P is closed under A iff, for any u, v, $[u]P/A[v]$ implies uPv.

Let P_1 be an irreflexive relation on U/A, and $P_2 \subseteq A$ be an irreflexive relation within the equivalence classes of A. The *lexicographic product* $P = P_1 \times P_2$ of these two relations is the relation defined on U by

$$uPv \quad \text{if either} \quad \text{(i)} \neg uAv \text{ and } [u]P_1[v]$$
$$\text{or} \quad \text{(ii)} \ uAv \text{ and } uP_2v.$$

If P_1 and P_2 are both partial orders, then their lexicographic product is a partial order. P is closed under A iff $P = P/A \times P \cap A$.

Two relations P and R are *consistent* if $P \cup R$ can be extended to a total ordering. A relation can be extended to a total ordering iff its transitive closure

is irreflexive. Thus, P is consistent with R iff the graph of the relation $P \cup R$ has no cycles.

Let A be an equivalence relation, and P be a partial order. A relation E is *consistent* with P and A if it can be extended to a total order $\bar{E}$ that fulfills the following two conditions:

(2.1) $P \subseteq \bar{E}$, so that if uPv then u occurs before v in the sequence defined by $\bar{E}$; and

(2.2) equivalent elements occur in consecutive locations in the sequence defined by $\bar{E}$; that is, if uAv, but $\neg uAw$, then either $w\bar{E}u$ and $w\bar{E}v$, or $u\bar{E}w$ and $v\bar{E}w$.

When A is trivial (i.e., A is the equality relation), then this new definition of consistency reduces to the previous one: E is consistent with P and $=$ iff it is consistent with P.

We have the following lemma:

LEMMA 2.1 *Let A be an equivalence relation, P be a partial order, and E be a relation on the same set. Then the following assertions are equivalent:*

(1) *E is consistent with P and A;*

(2) *(i) E is consistent with P, and (ii) E/A is consistent with P/A;*

(3) *(i) $E \cap A$ is consistent with $P \cap A$, and (ii) E/A is consistent with P/A; and*

(4) *(i) $(E \cap A) \cup (P \cap A)$ has no cycles, and (ii) all cycles of $E \cup A \cup P$ are contained in A.*

PROOF. Assume (1). Take a total order $\bar{E}$ that extends E and fulfills conditions (2.1) and (2.2). According to condition (2.1), $P \subseteq \bar{E}$, so that P is consistent with E. Condition (2.2) implies that $\bar{E}$ induces a total order $\bar{E}/A$ on the set of equivalence classes of A; this order extends both E/A and P/A; it follows that these two relations are consistent. Hence, (1) $\Rightarrow$ (2).

Clearly, (2) $\Rightarrow$ (3).

$E \cap A$ is consistent with $P \cap A$ iff $(E \cap A) \cup (P \cap A)$ has no cycles; hence, (3i) and (4i) are equivalent. Any cycle of $E/A \cup P/A$ corresponds to a cycle of $E \cup P \cup A$ that is not wholly contained within an equivalence class of A; conversely, any such cycle in $E \cup P \cup A$ induces a cycle of length ≥ 2 in $E/A \cup P/A$. Hence, (3ii) is equivalent to (4ii), and (3) $\Leftrightarrow$ (4).

Assume (3) holds true. One can order the equivalence classes of A in a linear order E_1 that extends $E/A \cup P/A$; within each class, one can order the elements in a linear order E_2 that extends $(E \cap A) \cup (P \cap A)$. Let $\bar{E} = E_1 \times E_2$. Then $\bar{E}$ is a linear order that extends E and fulfills conditions (2.1) and (2.2). Hence, (3) $\Rightarrow$ (1). □

To simplify discussion in the sequel, we identify a relation with its underlying graph, and a graph with its set of edges. A simple cycle in a graph will interchangeably be represented as a cyclic list of nodes or as a set of edges.

2.2 The Model

We characterize the code by a tuple $\langle \mathbf{V}, \mathbf{A}, \mathbf{P}, \mathbf{C} \rangle$. $\mathbf{V}$ is the set of variable accesses executed by the program. The variable accesses are partitioned into sets, each

Segment 1	*Segment 2*
a1: X := 1;	a2: y := Y;
b1: Y := 1;	b2: x := X;

Fig. 2. Parallel code.

set consisting of the accesses executed by one atomic operation. This partition is represented by an equivalence relation **A** on **V**; the equivalence classes of **A** are the set of accesses of atomic operations. **P** is the order on variable accesses required by the program and by the semantics of the individual operations; it is a partial order on **V**. The order **P** is *closed under* **A**; that is,

$$uPv; \quad u\mathbf{A}u', \quad v\mathbf{A}v', \quad \neg u\mathbf{A}v \quad \text{implies} \quad u'Pv'.$$

Intuitively, if one access of an operation **u** is required to precede an access of an operation **v**, and $\mathbf{u} \neq \mathbf{v}$, then all accesses of **u** are required to precede any access of **v**.

C is the conflict relation on accesses; it is a symmetric relation on **V**. **C** is irreflexive, but not necessarily transitive. For example, if u and w are read operations and v is a write operation, all accessing the same variable, then we have $u\mathbf{C}v$ and $v\mathbf{C}w$, but $\neg u\mathbf{C}w$.

The order relation **P** represents both order required on operations by the program and order required on accesses within each operation by the semantics of these operations. The requirement that **P** be closed with respect to **A** implies that **P**/**A** is a partial order on the set **V**/**A** of atomic operations. Conversely, one can characterize a code by a tuple $\langle \mathbf{V}, \mathbf{A}, \mathbf{P}_1, \mathbf{P}_2, \mathbf{C} \rangle$, where $\mathbf{P}_1$ is a partial order on the set **V**/**A** of atomic operations, and $\mathbf{P}_2 \subseteq \mathbf{A}$ is a partial order on accesses within each operation; **P** is the lexicographic product of $\mathbf{P}_1$ and $\mathbf{P}_2$ as defined above.

An *execution* **E** is a proper orientation of the conflict relation **C**. Informally, $u\mathbf{E}v$ if u and v are conflicting accesses (i.e., they both access the same variable and at least one of them is a write), and the access u occurs in storage before the access v.

An execution order **E** is *correct* if it is consistent with **P** and **A**; that is, **E** is correct if it can be extended to a linear order such that the accesses occur in this linear sequence in the order indicated by **P**, and accesses that belong to one operation are executed consecutively, not interrupted by other accesses. The first condition states that the order of execution of the operations is consistent with the order specified by the program. The second condition implies that the effect of the execution is as if each atomic operation were executed indivisibly.

One may think of the tuple $\langle \mathbf{V}, \mathbf{A}, \mathbf{P} \rangle$ as a specification for correct execution. A correct execution is one that behaves as a linear ordering of **V** that is consistent with **P** and **A**: Accesses occur in the order specified by **P**, and all accesses from an equivalence class of **A** are contiguous. The conflict relation **C** specifies the external interface of an execution: The observable behavior of an execution is determined by the orientation it induces on **C**. Therefore, we may take an execution to consist of just this orientation.

Example. Consider the program segments in Figure 2 (the example is taken from Collier [7]). X and Y are variables shared by both program segments, and

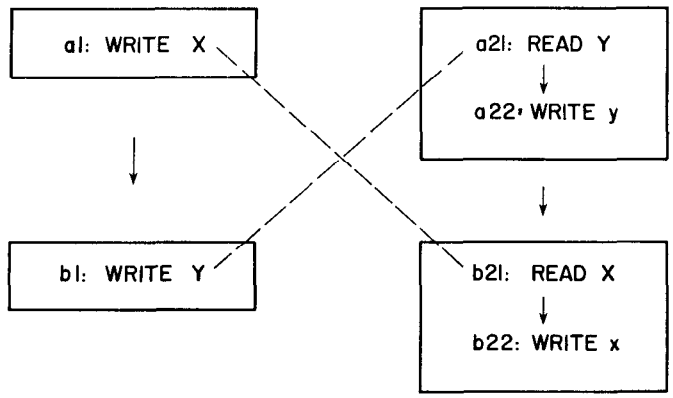


Fig. 3. Code specification.

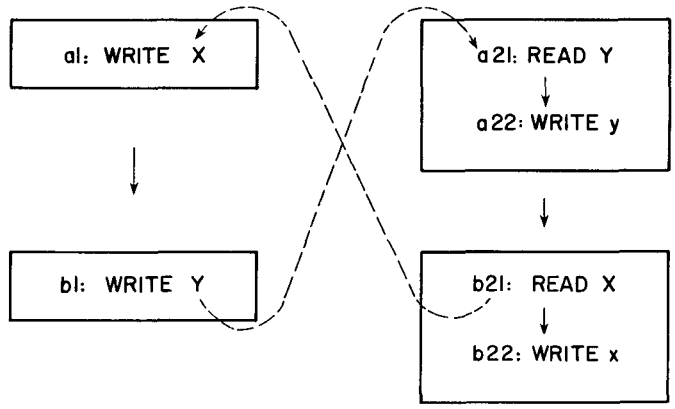


Fig. 4. Incorrect execution.

x and y are different registers. Assume that initially $X = Y = 0$. Each operation consists of a single assignment statement. No interleaving of these operations consistent with the order in each program can lead to a state where $x = 0$ and $y = 1$; such an execution would not be consistent with **P/A**. Indeed, if $x = 0$ then operation $b2$ was executed before $a1$; but then $a2$ should take effect before $b1$, so that $y = 0$.

The accesses executed by this code are shown in Figure 3. Here, and in the following figures, full arrows indicate the program order **P**, broken lines indicate conflict edges of **C**, and boxes enclose accesses of the same operation that should execute atomically. If the accesses to shared memory in either the first program segment or the second program segment are executed out of order, then it is quite possible to obtain this inconsistent result (Figure 4). For this case, $\mathbf{E} = \{(b1, a21), (b21, a1)\}$. So, $\mathbf{E} \cup \mathbf{P}$ has a cycle ($a1$, $b1$, $a21$, $a22$, $b21$, $a1$).

Note, however, that this is the only inconsistent result. For example, the access pattern $a21$, $b1$, $a1$, $b21$ would yield $x = 1$ and $y = 0$. This would be entirely

acceptable, since it produces the same results as the interleaving $a1$, $a21$, $a22$, $b21$, $b22$, $b1$.[2] For this case, $\mathbf{E} = \{(a1, b21), (a21, b1)\}$, and $\mathbf{E} \cup \mathbf{P}$ is acyclic.

One can control the order of execution of operations by introducing *delays*. A delay between two storage accesses u and v forces access u to complete before access v begins. The delay is enforced by the control logic of the computer, for example, by mechanisms that postpone the execution of a memory access until some previous load returned a value or a previous store was acknowledged. We denote by $u\mathbf{D}v$ the fact that access v is delayed until access u is executed. The relation **D** is a partial order.

If $u\mathbf{E}v$ then access u is executed earlier than access v; if $u\mathbf{D}v$ then access u is executed earlier than v; the temporal order of execution of accesses is clearly acyclic, and extends **E** and **D**. This implies the following lemma:

DELAY LEMMA. *For any execution,* **E** *is consistent with* **D**.

A delay relation **D** can be seen as a specification of the effect of some control mechanism: It restricts the executions to those that are consistent with **D**. A delay relation **D** *enforces correctness* if any execution order **E** that is consistent with **D** is correct (i.e., is consistent with **P** and **A**). A delay relation that enforces correctness is a control strategy that enforces correct execution of a given program. Such a delay relation always exists: One can force serial execution of all the storage accesses, in an order that fulfills conditions (2.1) and (2.2). This is not a very interesting solution, as it achieves correctness at the expense of complete loss of intraprocessor or interprocessor concurrency. We shall be interested in delay relations **D** that fulfill $\mathbf{D} \subseteq \mathbf{P}$. This restriction has two motivations: If $\mathbf{D} \subseteq \mathbf{P}$, any computation order that is consistent with **P** is also consistent with **D**, and **D** does not prohibit a computation that could occur within the constraints of the code. Also, in the situation where **P** represents the ordering within sequential program segments executed by distinct processors, then constraints in **P** are constraints on the order of accesses executed by the same processor; these can be enforced by control logic local to the processor. A delay pair that is not in **P** puts a constraint on the order of two accesses that are performed by distinct processors; the enforcement of such constraint requires expensive interprocessor coordination.

3. SYSTEMS WITH NO ATOMICITY CONSTRAINTS

A delay relation $\mathbf{D} \subseteq \mathbf{P}$ that enforces correctness does not always exist: Delays cannot always guarantee atomicity of operations. Consider, for example, the code in Figure 5. The first operation atomically reads the record (X, Y), whereas the second operation atomically updates this record. Even if a correct order of accesses is enforced, one can still have an execution order $a1$, $a2$, $b2$, $b1$ that violates the atomicity of the operations; the read returns a half updated record.

In this section we examine the situation where each operation performs a unique storage access; that is, each equivalence class of **A** is a singleton (**A** is the equality relation). In this case the issue of atomicity does not arise; **E** is correct

[2] The reader may observe that the sequential consistency requirement is much weaker than the serialization principle addressed in database theory [10], which would forbid this second outcome.

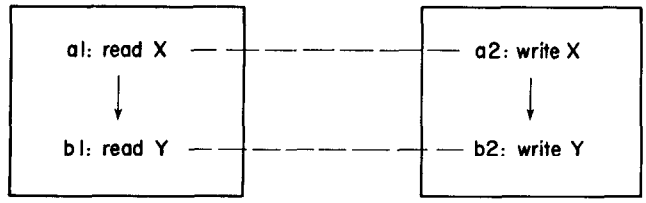


Fig. 5. Code with atomicity constraints.

Fig. 6. Code that does not require delays.

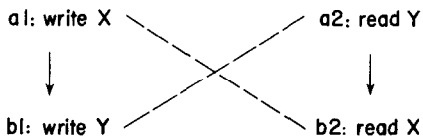


Fig. 7. Code that requires delays.

iff it is consistent with **P**. This is a simple case where a delay relation $\mathbf{D} \subseteq \mathbf{P}$ that enforces correctness exists, namely, $\mathbf{D} = \mathbf{P}$. In many cases a proper subset of **P** can, too, enforce correctness. We shall be looking for a *minimal* delay relation $\mathbf{D} \subseteq \mathbf{P}$ that enforces correctness.

We give a constructive proof that there exists a unique minimal partial order relation $\mathbf{D} \subseteq \mathbf{P}$ that enforces correctness. Our strategy for finding it is based on the following reasoning: An execution **E** is incorrect if $\mathbf{P} \cup \mathbf{E}$ has cycles. Since **E** is an orientation of **C**, all such cycles are also cycles of the graph of $\mathbf{P} \cup \mathbf{C}$. This graph is known in advance, so its cycles indicate potential violations of correctness at run time. Suppose we enforce a delay $u\mathbf{D}v$ for every pair of operations uv such that $u\mathbf{P}v$, and uv is an edge on a cycle of $\mathbf{P} \cup \mathbf{C}$. Then every cycle of $\mathbf{P} \cup \mathbf{C}$ is also a cycle of $\mathbf{D} \cup \mathbf{C}$. It follows that every cycle of $\mathbf{P} \cup \mathbf{E}$ would also be a cycle of $\mathbf{D} \cup \mathbf{E}$. $\mathbf{D} \cup \mathbf{E}$, however, is acyclic by the delay lemma. So choosing delays in this way ensures $\mathbf{P} \cup \mathbf{E}$ is acyclic and **E** is correct. Consider, for example, the code of Figure 6. Here, $\mathbf{P} \cup \mathbf{C}$ has no cycles; the four storage accesses may occur in arbitrary order, with no perceived violation of the required program order. In Figure 7 we have switched the order of the two reads; this is essentially the same code as presented in Figure 2. The graph of $\mathbf{P} \cup \mathbf{C}$ has one directed cycle, ($a1$, $b1$, $a2$, $b2$, $a1$). This cycle indicates a possible incorrect execution: the execution where $\mathbf{E} = \{(b1, a2), (b2, a1)\}$. If delays are imposed on the pairs ($a1$, $b1$) and ($a2$, $b2$), however (these are the **P** edges on this cycle), then the incorrect execution cannot occur as it would be inconsistent with the delays. In this case we must enforce all the order constraints given by **P** in order to enforce correct execution. Figure 6 was an example where this is not the case.

In order to construct **D**, it is sufficient but not necessary to consider all cycles of $\mathbf{C} \cup \mathbf{P}$; rather, one can consider "minimal counterexamples," that is, acyclic

subsets of **C** that are "minimally incorrect." Let Φ be the family of acyclic subsets of **C** that are not consistent with **P**; $S \in \Phi$ if $S \subseteq \mathbf{C}$ is acyclic and $\mathbf{P} \cup S$ contains a cycle. We call a minimal element of Φ a *minimal inconsistent execution* (minimality is by set containment). The family Φ of sets is closed under containment: If $S \in \Phi$ and $S' \supset S$, then $S' \in \Phi$. Hence, $S \in \Phi$ iff it contains a minimal inconsistent execution. We have the following theorem:

THEOREM 3.1 *Let* **D** *be a delay relation. Then* **D** *enforces correctness iff, for every minimal inconsistent execution* S, $\mathbf{D} \cup S$ *has a cycle.*

We need the following lemma to prove the theorem:

LEMMA 3.2 *Let* $G = \langle V, E \rangle$ *be a directed acyclic graph, and* u *and* v *be two nonadjacent nodes of* G. *Then either* $G_1 = \langle V, E \cup \{uv\} \rangle$ *or* $G_2 = \langle V, E \cup \{vu\} \rangle$ *is an acyclic graph.*

PROOF. If both G_1 and G_2 contain cycles, then there is a path in G from u to v and a path from v to u. Hence, G contains a cycle—a contradiction. □

PROOF OF THEOREM 3.1 $\Rightarrow$ Let **E** be an inconsistent execution. **E** contains a minimal inconsistent execution S. By assumption, $S \cup \mathbf{D}$ has a cycle, implying that $\mathbf{E} \cup \mathbf{D}$ have a cycle; but the latter is impossible.

$\Leftarrow$ Assume there exists a minimal inconsistent execution S such that $S \cup \mathbf{D}$ is acyclic. By repeated application of Lemma 3.2, S can be extended to an execution order **E** such that $\mathbf{E} \cup \mathbf{D}$ is acyclic. Since $S \subseteq \mathbf{E}$, and S is inconsistent, then **E** is inconsistent; so **D** does not enforce correctness. □

In the next subsection, we shall build delay relations that enforce consistency using the following, generic construction: Let Ψ be a family of cycles in $\mathbf{P} \cup \mathbf{C}$ such that each minimal inconsistent execution is contained in a cycle of Ψ; let **D** be the set of **P** edges on the cycles in Ψ. Then $S \cup \mathbf{D}$ has a cycle for each minimal inconsistent execution S, and hence, **D** enforces correctness.

3.1 Critical Pairs

A set σ is a *critical cycle* of (**P**, **C**) if it is a simple cycle of $\mathbf{P} \cup \mathbf{C}$ and has no chords in **P** ($rs \in \sigma$, $uv \in \sigma$, $ru \in \mathbf{P}$ implies $s = u$).

An edge $uv \in \mathbf{P}$ is a *critical pair* (of (**P**, **C**)) if it occurs in a critical cycle.

LEMMA 3.3 *Let* S *be a minimal inconsistent execution, and* σ *be a shortest cycle in* $S \cup \mathbf{P}$ (*i.e., a cycle with fewest number of edges*). *Then* σ *is a critical cycle.*

PROOF. Let $\sigma = (v_0, \ldots, v_{n-1}, v_0)$. If $v_i = v_j$, with $i < j$, then $(v_0, v_i, v_{j+1}, v_{n-1}, v_0)$ is a shorter cycle in $\mathbf{P} \cup S$; hence, σ is simple. If $v_i \mathbf{P} v_j$, with $i < j$, then $(v_0, \ldots, v_i, v_j, \ldots, v_{n-1}, v_0)$ is a shorter cycle in $\mathbf{P} \cup S$; if $v_i \mathbf{P} v_j$, with $j < i$ (and $i, j \neq n-1, 0$), then $(v_j, v_{j+1}, \ldots, v_i, v_j)$ is a shorter cycle in $\mathbf{P} \cup S$. Hence, σ has no **P** chords.

COROLLARY 3.4 *Let* **D** *be the "critical pair" relation. If* **E** *is an execution order that is consistent with* **D**, *then* **E** *is consistent with* **P**.

PROOF. The previous lemma implies that each minimal inconsistent execution S is contained in a critical cycle. Making **D** consist of all critical pairs ensures that no such execution can come to pass. □

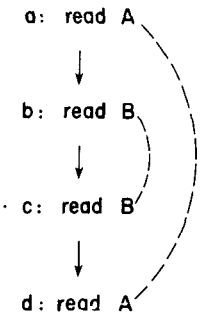


Fig. 8. Intraprocess data dependencies.

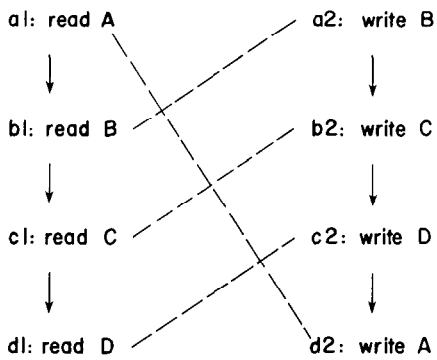


Fig. 9. Interprocess data dependencies.

3.2 Examples

Figure 8 illustrates that data dependencies within one program segment are handled correctly as a particular case of critical pairs. We have a critical cycle (a, d, a) and a critical cycle (b, c, b). The pairs ad and bc are critical pairs, so that d is delayed until a terminates, and c is delayed until b terminates.

Figure 9 illustrates that use of critical pairs can save delays. We have four simple cycles in $\mathbf{P} \cup \mathbf{C}$ (see Figure 10):

(i) $(a1, b1, a2, b2, c1, d1, c2, d2, a1)$,
(ii) $(a1, b1, a2, d2, a1)$,
(iii) $(a1, c1, b2, d2, a1)$, and
(iv) $(a1, d1, c2, d2, a1)$.

The first cycle is not critical, as it has $\mathbf{P}$ chords—for example, $(a1, c1)$. In fact, this cycle properly contains the edges from $\mathbf{C} - \mathbf{P}$ of each of the remaining three cycles and does not define a minimal inconsistent execution. The remaining three cycles are critical. The critical pairs are $\mathbf{D} = \{(a1, b1), (a1, c1), (a1, d1), (a2, d2), (b2, d2), (c2, d2)\}$. This set of delays enforces consistency. In the first program segment, the last three operations are delayed until $a1$ is executed. In the second program segment, $d2$ is delayed until all the other operations have been executed. On the other hand, the last three accesses in the first program segment and the first three accesses in the second program segment can be executed in arbitrary order, even though these are accesses to shared read-write variables.

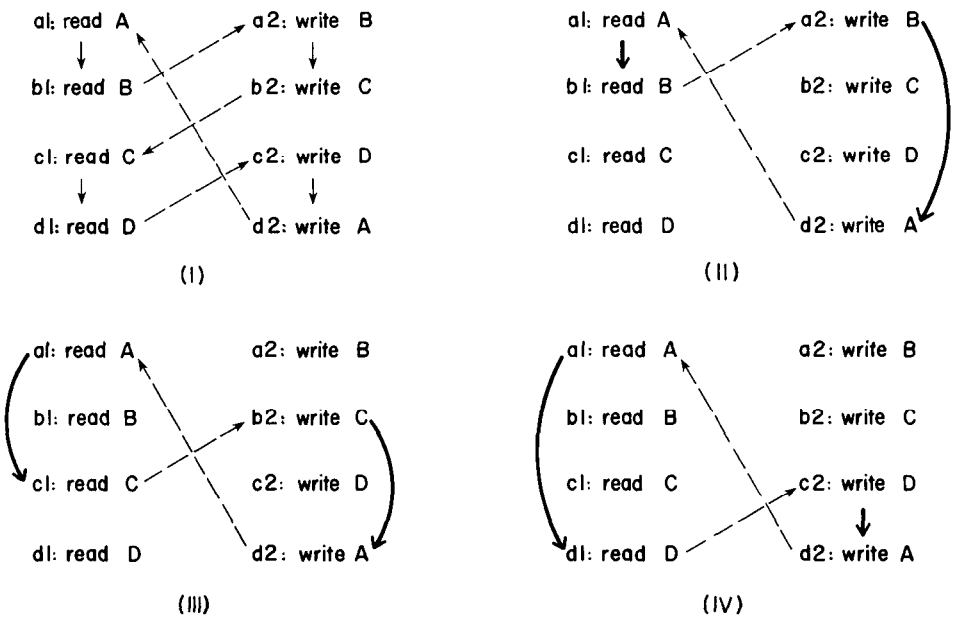


Fig. 10. Cycles and critical pairs.

3.3 Minimality

We have shown that each minimal inconsistent execution is contained in a critical cycle. We show now that the converse holds: The edges from $\mathbf{C} - \mathbf{P}$ in a critical cycle are a minimal inconsistent execution. It follows that the critical pair relation is a minimal delay relation that enforces correctness.

LEMMA 3.5 (*i*) *Let σ be a critical cycle of* $(\mathbf{P}, \mathbf{C})$, *and let* $S = \sigma - \mathbf{P}$. *Let π be an arbitrary simple cycle in* $S \cup \mathbf{P}$. *Then π is obtained from σ by replacing each* $\mathbf{P}$ *edge of σ by a simple path of* $\mathbf{P}$ *edges; all S edges of σ occur in π (see Figure* 11).

(*ii*) *The last result holds true even if* $\mathbf{P}$ *is an arbitrary transitive relation (not necessarily acyclic), provided that* $\pi \cap S \neq \emptyset$. *That is, if* $S \cap \pi \neq \emptyset$ *then* $S \subseteq \pi$.

PROOF. Let $\sigma = (v_0, \ldots, v_{n-1}, v_0)$. In (i) $\mathbf{P}$ is acyclic so that π must contain an edge from S, call it v_iv_{i+1}; such an edge is assumed to exist in (ii). Let v_kv_{k+1} be the following edge from S on the cycle π ($k = i$ if there is no other edge from S). Now there are two cases:

(1) If $k = i + 1$ then $v_{i+1}v_{i+2}$ is an S edge that immediately follows v_iv_{i+1} on σ; since π is simple, $v_{i+1}v_{i+2}$ also immediately follows v_iv_{i+1} on π.

(2) If $k \neq i + 1$ then v_{i+1} is connected in π to v_k by a simple path of $\mathbf{P}$ edges. It follows that $v_{i+1}\mathbf{P}v_k$ since $\mathbf{P}$ is transitive, and since σ has no $\mathbf{P}$ chords, $k = i + 2$. v_iv_{i+1} is immediately followed on σ by the $\mathbf{P}$ edge $v_{i+1}v_{i+2}$, which is immediately followed by the S edge $v_{i+2}v_{i+3}$; on π, v_iv_{i+1} is immediately followed by a simple path of $\mathbf{P}$ edges, which is immediately followed by the S edge $v_{i+2}v_{i+3}$.

The claim follows by induction. □

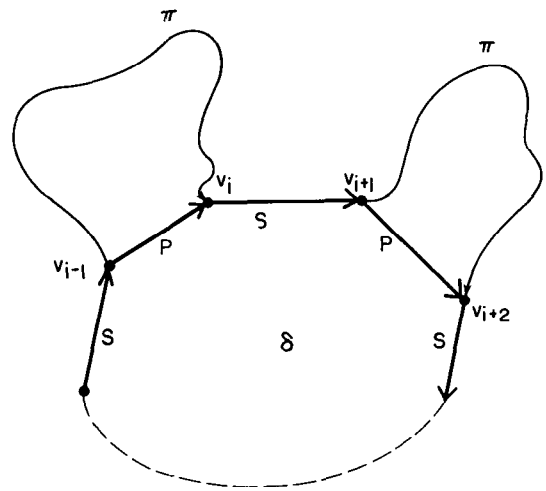


Fig. 11. Any simple cycle replaces P edges by simple paths of P edges. The nonprogram order conflict edges remain the same.

COROLLARY 3.6 *Let σ be a critical cycle, and $S = \sigma - \mathbf{P}$. Then S is a minimal inconsistent execution, and σ is the unique critical cycle in $S \cup \mathbf{P}$.*

PROOF. By the previous lemma, S is contained in the set of edges of any cycle in $S \cup \mathbf{P}$, so that S is a minimal inconsistent execution. If π is a critical cycle in $S \cup \mathbf{P}$, then π is obtained from σ by replacing $\mathbf{P}$ edges by paths of $\mathbf{P}$ edges. But π has no chords in $\mathbf{P}$, so that each such path has length 1. It follows that $\pi = \sigma$. □

THEOREM 3.7 *Let $R \subseteq \mathbf{P}$ be a relation that enforces consistency with $\mathbf{P}$. Then the critical pair relation $\mathbf{D}$ is contained in R^+, the irreflexive, transitive closure of R.*

PROOF. Let σ be a critical cycle, and $uv \in \mathbf{P}$ be an edge of σ. Let S be the set of edges from $\mathbf{C} - \mathbf{P}$ in σ. Then S is a minimal inconsistent execution by Corollary 3.6. Since R enforces correctness, $R \cup S$ contains a cycle by Theorem 3.1. It follows, by Lemma 3.5, that $uv \in R^+$. □

3.4 Simplified Definitions of Critical Cycles

The definition of a critical cycle can be further restricted, without affecting the definition of the critical relation $\mathbf{D}$. First, we can ignore cycles in $\mathbf{C} \cup \mathbf{P}$ that do not contain $\mathbf{P}$ edges; these do not contribute critical pairs. Second, we may require that critical cycles do not contain chords in $\mathbf{C}$, with the exception of "trivial" chords consisting of the reversal of an edge on the cycle. Indeed, let $\sigma = (v_0, \ldots, v_{n-1}, v_0)$ be a critical cycle, and assume $v_i \mathbf{C} v_j$, where $j \neq i \pm 1$. Then $\sigma_1 = (v_0, \ldots, v_i, v_j, v_{n-1}, v_0)$ and $\sigma_2 = (v_i, v_{i+1}, \ldots, v_j, v_i)$ are simple cycles of $\mathbf{P} \cup \mathbf{C}$ with no $\mathbf{P}$ chords (Figure 12); they both are critical cycles. Hence, all $\mathbf{P}$ edges of σ are $\mathbf{P}$ edges of these shorter critical cycles.

Examples. Consider the code shown in Figure 13. We have three critical cycles: $(a1, b1, a2, b2, a1)$, $(a1, b1, a2, a1)$, and $(a1, a2, b2, a1)$. The first cycle has a

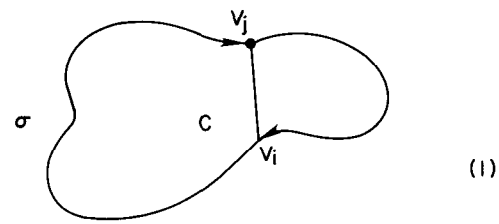
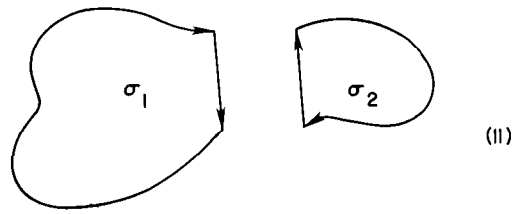


Fig. 12. Deleting C chords from critical cycles.



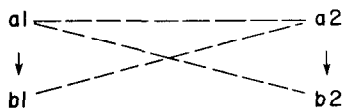
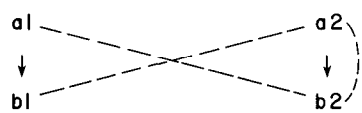


Fig. 13. Simplified critical cycles.

Fig. 14. Code for two critical cycles.



nontrivial **C** chord $(a1, a2)$ and can be ignored. We get, from the remaining two cycles, that $\mathbf{D} = \{(a1, b1), (a2, b2)\}$.

In the code shown in Figure 14, we have two critical cycles: $(a1, b1, a2, b2, a1)$ and $(a2, b2, a2)$. We get $\mathbf{D} = \{(a1, b1), (a2, b2)\}$. The first cycle cannot be ignored, even though it has a (trivial) **C** chord, namely, $(b2, a2)$; note, too, that a critical cycle may nodewise contain another critical cycle.

When storage accesses are the usual read and write operations, such that a write access conflicts with any other access to the same variable and read accesses do not conflict, the situation depicted in Figure 14 cannot occur: If $p_1\mathbf{C}p_2\mathbf{C}p_3\mathbf{C}p_4$ then all accesses involve the same variable and either p_2 or p_3 is a write; but then either $p_1\mathbf{C}p_3$ or $p_2\mathbf{C}p_4$. This "semitransitivity" of **C** implies the following result:

LEMMA 3.8 *Let* **C** *be the conflict relation for ordinary read and write operations. Then a pair* $uv \in \mathbf{P}$ *of accesses is critical iff it occurs on a cycle of* $\mathbf{P} \cup \mathbf{C}$ *with a minimal set of nodes.*

Critical cycles are particularly easy to characterize when code consists of the union of disjoint serial chains of accesses, and accesses are usual reads and writes. We leave to the reader the proof of the following result:

THEOREM 3.9 *A cycle* σ *in* $\mathbf{P} \cup \mathbf{C}$ *is critical iff it fulfills the following conditions*:

(*i*) σ *contains at most two accesses from any chain; these accesses occur at successive locations in* σ.

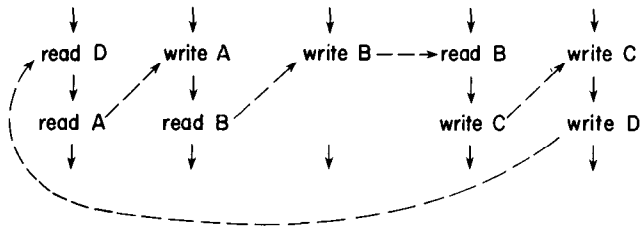


Fig. 15. Critical cycle in parallel program segments.

(*ii*) *σ contains either zero, two, or three accesses to any variable; the accesses occur in consecutive locations on σ. The possible configurations are read $x \rightarrow$ write x, write $x \rightarrow$ read x, write $x \rightarrow$ write x, or read $x \rightarrow$ write $x \rightarrow$ read x. This is illustrated in Figure* 15.

4. USING DELAYS IN GENERAL SYSTEMS

We now consider general systems $\langle \mathbf{V}, \mathbf{A}, \mathbf{P}, \mathbf{C} \rangle$, with nontrivial atomicity requirements ($\mathbf{A}$ is an equivalence relation, but is not the equality relation). We seek a delay relation $\mathbf{D}$ that enforces correctness; we shall relax our previous requirements on $\mathbf{D}$. In our generic situation, that of parallel serial program segments, the accesses of an atomic operation are executed by one processor; delays between such accesses can be easily enforced by the control logic of the processor executing them, with no loss of interprocessor concurrency. Therefore, $\mathbf{D}$ is allowed now to be an arbitrary (acyclic) subset of $\mathbf{P} \cup \mathbf{A}$.

The derivation of a solution is obtained by extending the definitions and results of the previous section. A set $S \subseteq \mathbf{C}$ is a *minimal inconsistent execution* if it fulfills the following two conditions:

(1) S is not consistent with $\mathbf{P}$ and $\mathbf{A}$; and

(2) S' is consistent with $\mathbf{P}$ and $\mathbf{A}$, for any proper subset S' of S.[3]

The family of sets inconsistent with $\mathbf{P}$ and $\mathbf{A}$ is closed under containment. Therefore, Theorem 3.1 is valid for this extended definition: $\mathbf{D}$ enforces correctness iff $\mathbf{D} \cup S$ has a cycle for every minimal inconsistent execution S.

According to Lemma 2.1, a set $S \subseteq \mathbf{C}$ is inconsistent with $\mathbf{P}$ and $\mathbf{A}$ if either $S \cap \mathbf{A}$ is inconsistent with $\mathbf{P} \cap \mathbf{A}$ or $S/\mathbf{A}$ is inconsistent with $\mathbf{P}/\mathbf{A}$. Hence, S is a minimal inconsistent execution iff either $S \subseteq \mathbf{A}$ and S is a minimal inconsistent execution with respect to the program order $\mathbf{P} \cap \mathbf{A}$, or $S/\mathbf{A}$ is a minimal inconsistent execution with respect to the program order $\mathbf{P}/\mathbf{A}$, and $S \cap \mathbf{A} = \emptyset$. The first type of inconsistencies (wrong execution order inside operations) can be prevented by critical delays as defined in Section 3.1. In order to handle the second type of inconsistencies (wrong execution order across operations), we consider how critical cycles of operations are related to cycles of accesses.

Let σ be a cycle of a relation R on $\mathbf{V}$. The *projection* of σ on $\mathbf{A}$ is the cycle σ of the relation $R/\mathbf{A}$ obtained by replacing each vertex by its equivalence class under $\mathbf{A}$ and deleting self-loops.

[3] Recall from Section 2.1 that S is consistent with $\mathbf{P}$ and $\mathbf{A}$ if there is a topological sort of $S \cup \mathbf{P}$ with $\mathbf{A}$ related elements clustered together.

LEMMA 4.1 *Let σ be a critical cycle of* $(\mathbf{P} \cup \mathbf{A}, \mathbf{C} - \mathbf{A})$, *and σ be the projection of σ on* $\mathbf{A}$. *Then σ is a critical cycle of* $(\mathbf{P/A}, \mathbf{C/A})$. *Conversely, every critical cycle σ of* $(\mathbf{P/A}, \mathbf{C/A})$ *is the projection of a critical cycle σ of* $(\mathbf{P} \cup \mathbf{A}, \mathbf{C} - \mathbf{A})$. *Moreover, if σ is the projection of a cycle π in* $\mathbf{P} \cup \mathbf{A} \cup \mathbf{C}$, *then σ can be chosen such that the edges of σ from* $\mathbf{C} - \mathbf{P} \cup \mathbf{A}$ *are the same as the edges of π from* $C - \mathbf{P} \cup \mathbf{A}$.

PROOF. $\Rightarrow$ Let $\sigma = (v_0, \ldots, v_{n-1}, v_0)$ be a critical cycle of $(\mathbf{P} \cup \mathbf{A}, \mathbf{C} - \mathbf{A})$, and σ be the projection of σ. σ has no chords in $\mathbf{A}$, so that $v_i \mathbf{A} v_j$ only if $j = i + 1$. It follows that σ is simple. If $[v_i]\mathbf{P/A}[v_j]$ then, since $\mathbf{P}$ is closed under $\mathbf{A}$, $v_i \mathbf{P} v_j$, and $j = i + 1$. It follows that σ has no chords in $\mathbf{P/A}$.

$\Leftarrow$ Let $\sigma = (\mathbf{v}_0, \ldots, \mathbf{v}_{n-1}, \mathbf{v}_0)$ be a critical cycle of $(\mathbf{P/A}, \mathbf{C/A})$. Let $\pi = (v_0, \ldots, v_{m-1}, v_0)$ be a cycle in $\mathbf{P} \cup \mathbf{C} \cup \mathbf{A}$ such that σ is the projection of π. If $v_i \mathbf{A} v_{i+1}$ and $v_{i+1} \mathbf{A} v_{i+2}$, then these two edges can be replaced by one edge, $v_i v_{i+2} \in \mathbf{A}$, without changing the projection of π. If $v_i \mathbf{A} v_{i+1}$ and $v_{i+1} \mathbf{P} v_{i+2}$, then, since $\mathbf{P}$ is closed under $\mathbf{A}$, $v_i \mathbf{P} v_{i+2}$, and $v_i v_{i+1}$, $v_{i+1} v_{i+2}$ can be replaced by $v_i v_{i+2}$. The same holds true if $v_i \mathbf{P} v_{i+1}$ and $v_{i+1} \mathbf{A} v_{i+2}$. Repeating this process, we obtain a cycle $\sigma = (u_0, \ldots, u_{k-1}, u_0)$ such that σ is the projection of σ, and an edge from $\mathbf{A}$ is not preceded or followed by an edge from $\mathbf{A}$ or $\mathbf{P}$ in σ. σ coincides with π on $\mathbf{C} - \mathbf{P} \cup \mathbf{A}$.

If $u_i \mathbf{A} u_j$ and $j \neq i + 1$, then σ is not simple; if $u_i \mathbf{P} u_j$ and $j \neq i + 1$, then σ has a $\mathbf{P/A}$ chord. Hence, σ is a critical cycle of $(\mathbf{P} \cup \mathbf{A}, \mathbf{C} - \mathbf{A})$. □

THEOREM 4.2 *Let $\mathbf{D}_\mathbf{A}$ be the set of critical pairs of* $(\mathbf{P} \cap \mathbf{A}, \mathbf{C} \cap \mathbf{A})$, *let $\mathbf{D}^\mathbf{A}$ be the set of critical pairs of* $(\mathbf{P} \cup \mathbf{A}, \mathbf{C} - \mathbf{A})$, *and let $\mathbf{D}_0 = \mathbf{D}_\mathbf{A} \cup \mathbf{D}^\mathbf{A}$. Then each minimal inconsistent execution S is contained in a simple cycle of $S \cup \mathbf{D}_0$; in particular, $\mathbf{D}_0$ enforces correctness.*

PROOF. Let S be a minimal inconsistent execution. If $S \subseteq \mathbf{A}$, and S is not consistent with $\mathbf{P} \cap \mathbf{A}$, then there is a critical cycle σ of $(\mathbf{P} \cap \mathbf{A}, \mathbf{C} \cap \mathbf{A})$ such that $S = \sigma - \mathbf{P}$. But $\sigma \cap \mathbf{P} \subseteq \mathbf{D}_\mathbf{A} \subseteq \mathbf{D}_0$, so that σ is a simple cycle in $S \cup \mathbf{D}_0$ that contains S.

If $S \cap \mathbf{A} = \emptyset$, and $S/\mathbf{A}$ is not consistent with $\mathbf{P/A}$, then there is a critical cycle σ of $(\mathbf{P/A}, \mathbf{C/A})$ such that $S/\mathbf{A} = \sigma - \mathbf{P/A}$. By the previous lemma, there exists a critical cycle σ of $(\mathbf{P} \cup \mathbf{A}, \mathbf{C} - \mathbf{A})$ such that σ is the projection of σ on $\mathbf{A}$, and $S = \sigma - \mathbf{P} \cup \mathbf{A}$. But $\sigma \cap (\mathbf{P} \cup \mathbf{A}) \subseteq \mathbf{D}^\mathbf{A} \subseteq \mathbf{D}_0$, so that σ is a simple cycle in $S \cup \mathbf{D}_0$ that contains S. □

4.1 Examples

Consider the code shown in Figure 16. We have one critical cycle of $(\mathbf{P} \cap \mathbf{A}, \mathbf{C} \cap \mathbf{A})$, the cycle $(a3, b3, a3)$. Hence, $\mathbf{D}_\mathbf{A} = \{(a3, b3)\}$; this delay enforces correct execution within atomic sets. We have one critical cycle of $(\mathbf{P} \cup \mathbf{A}, \mathbf{C} - \mathbf{A})$, the cycle $(a1, b1, a2, b2, a1)$. Hence, $\mathbf{D}^\mathbf{A} = \{(a1, b1), (a2, b2)\}$. This set of delays enforces correct execution across atomic sets. We obtain $\mathbf{D}_0 = \{(a1, b1), (a2, b2), (a3, b3)\}$. If these three delays are enforced, any execution will look as if ai occurred before bi, $i = 1, \ldots, 3$, and the first and last program segments executed atomically, even though no locking was done to enforce atomicity. Note that $\mathbf{D}_0 \subseteq \mathbf{P}$ in this example.

The code of Figure 17 consists of the first two program segments of the previous example; however, we do not require any more than $a1$ occurs before $b1$. We still get the same "external" critical cycle $(a1, b1, a2, b2, a1)$ and the critical pairs

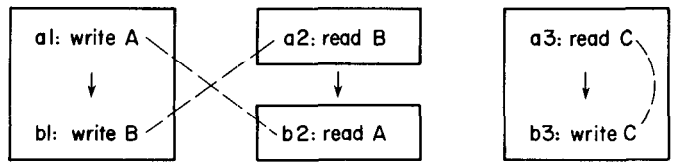


Fig. 16. Example code for nontrivial atomicity constraints.

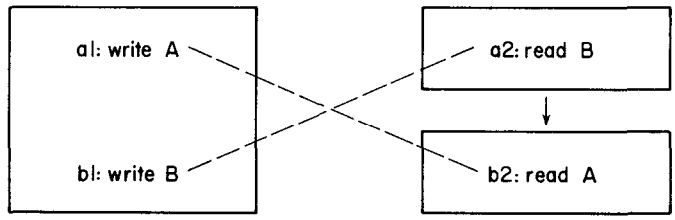


Fig. 17. Code for the first two program segments of Figure 16.

$\mathbf{D}_0 = \{(a1, b1), (a2, b2)\}$. In this case we have to impose a delay between pairs of accesses that are not **P** related. If these delays are enforced, then the outcome of any execution will be as if $(a1, b1)$ were executed atomically, even though no locking is used. Note that the accesses may occur in the order $a1, a2, b2, b1$. This does not result in a perceived violation of atomicity as it leads to the same state as the sequence $a2, a1, b1, b2$.

The code of Figure 18 is identical to the first two program segments of Figure 16, except that the accesses in the first program segment are required to occur in the reverse order. We still get the same critical cycle (of $(\mathbf{P} \cup \mathbf{A}, \mathbf{C} - \mathbf{A})$) $(a1, b1, a2, b2, a1)$ and the same set of critical pairs $\mathbf{D}_0 = \{(a1, b1), (a2, b2)\}$. In this case, in order to enforce a correct execution, we force accesses to occur in an order that is the reverse of the order specified by the program! This apparent paradox can be understood by going back to the example in Figure 6 and the discussion therein. The code given there differs from the code of Figure 18 only in its atomicity requirements. No execution order can lead to a violation of the program order requirements; hence, we are free to reorder accesses in an arbitrary manner. On the other hand, an execution of $b1$ before $a1$ may lead to a violation of atomicity, for example, if the accesses occur in the order $b1, a2, b2, a1$. No violation of atomicity may occur when $a1$ executes before $b1$ (and $a2$ before $b2$).

We conclude by going back to the code shown in Figure 5. In this example we have two critical cycles, $(a1, a2, b2, b1, a1)$ and $(a1, b1, b2, a2, a1)$(one cycle is the reversal of the other). The set of critical pairs is $\mathbf{D}_0 = \{(a1, b1), (b1, a1), (a2, b2), (b2, a2)\}$. $\mathbf{D}_0$ requires delaying $b1$ until $a1$ occurred, and vice-versa; this is clearly impossible.

The delay relation $\mathbf{D}_0$ we defined is a subset of $\mathbf{P} \cup \mathbf{A}$; **A** has cycles, and $\mathbf{D}_0$ may also have cycles. If $\mathbf{D}_0$ has cycles, then it cannot be physically enforced. Theorem 4.2 is still formally correct: Any execution order that is consistent with $\mathbf{D}_0$ is consistent with **P**. If $\mathbf{D}_0$ has cycles, however, then no execution order is consistent with $\mathbf{D}_0$; the antecedent is always false, and the implication is vacuous.

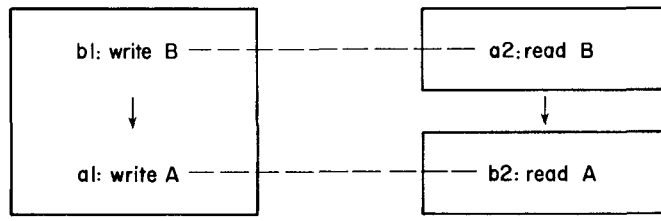


Fig. 18. For this example, delays must force accesses to be in reverse of program order.

We shall later show that $\mathbf{D}_0$ is the minimal subset of $\mathbf{P} \cup \mathbf{A}$ that enforces correctness. If $\mathbf{D}_0$ has cycles, then correct executions cannot be enforced using only delays in $\mathbf{P} \cup A$; some other mechanism, such as locking, is required. This is examined in Section 5. The pairs of "contradictory" delays obtained by the critical pair analysis will be used to determine where locking is necessary.

4.2 Systems Where Delays Enforce Correctness

Let $\mathbf{D}_0$ be the delay relation defined in Theorem 4.2. $\mathbf{D}_0$ is acyclic, and hence enforceable whenever $\mathbf{D}_0 \subseteq \mathbf{P}$. An edge $uv \in \mathbf{A} - \mathbf{P}$ may occur in $\mathbf{D}_0$ only if it occurs in a critical cycle σ of $(\mathbf{P} \cup \mathbf{A}, \mathbf{C} - \mathbf{A})$. Let ru and vw be, respectively, the preceding and following edges on σ. Since σ is critical, we have $ru, vw \in \mathbf{C} - \mathbf{P} \cup \mathbf{A}$. We define below a natural condition that prevents the occurrence of such a situation.

An access u is called *external* if there exists another access v such that $\neg u\mathbf{A}v$, $u\mathbf{C}v$, but neither $u\mathbf{P}v$ nor $v\mathbf{P}u$. An access is external iff it conflicts with an access of another operation and the code does not specify the order of execution of these two conflicting accesses. Such an access represents a nondeterministic interaction with another operation. A code has the *single external access* property if each operation contains at most one external access.

In the case of a parallel program that consists of several sequential program segments, accesses executed by the same program segment are ordered by $\mathbf{P}$. An access is external iff it conflicts with an access executed by another program segment. This implies that an access is external iff it accesses a shared read-write variable. The code has the single external access property iff each operation accesses at most one shared read-write variable.

LEMMA 4.3 *Suppose σ is a critical cycle of $(\mathbf{P} \cup \mathbf{A}, \mathbf{C} - \mathbf{A})$, in a program with the single external access property. Then σ does not contain edges from $\mathbf{A}$.*

PROOF. Let $\sigma = (v_0, \ldots, v_{n-1}, v_0)$ be a critical cycle of $(\mathbf{P} \cup \mathbf{A}, \mathbf{C} - \mathbf{A})$. Assume $v_i\mathbf{A}v_{i+1}$. If $v_{i-1}\mathbf{P}v_i$ then, since $\mathbf{P}$ is closed under $\mathbf{A}$, $v_{i-1}v\mathbf{P}_{i+1}$, and σ has a $\mathbf{P}$ chord. If $v_i\mathbf{P}v_{i-1}$ then v_iv_{i-1} is a $\mathbf{P}$ chord of σ. It follows that $\neg v_{i-1}\mathbf{P}v_i$ and $\neg v_i\mathbf{P}v_{i-1}$. Similarly, neither $v_{i+1}\mathbf{P}v_{i+2}$, nor $v_{i+2}\mathbf{P}v_{i+1}$. Thus, both v_i and v_{i+1} are external accesses—a contradiction. □

COROLLARY 4.4 *Let $\mathbf{E}$ be an execution order for a code with the single external access property. If $\mathbf{E}$ is consistent with $\mathbf{P}$, then $\mathbf{E}/\mathbf{A}$ is consistent with $\mathbf{P}/\mathbf{A}$.*

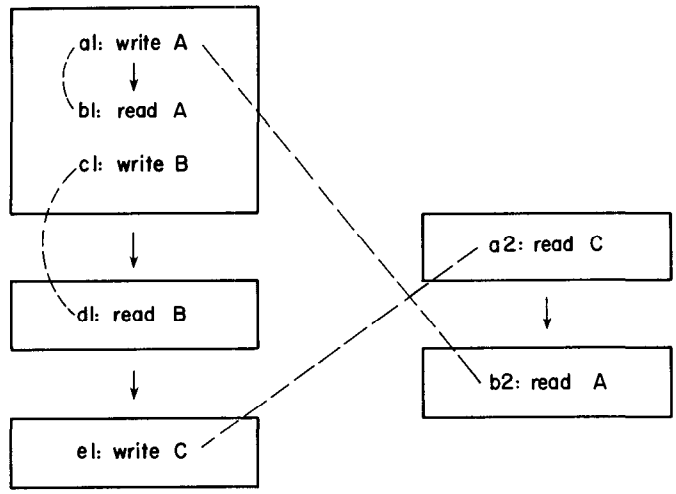


Fig. 19. Code with single external access property.

PROOF. If $\mathbf{E/A}$ is not consistent with $\mathbf{P/A}$, then $(\mathbf{P} \cup \mathbf{A}, \mathbf{E} - \mathbf{A})$ has a critical cycle. This cycle has no $\mathbf{A}$ edges by the previous lemma and, hence, is a cycle in $\mathbf{P} \cup \mathbf{E}$. Thus, $\mathbf{E}$ is not consistent with $\mathbf{P}$. □

Thus, in a code with the single external access property, an execution order that respects program order constraints also respects atomicity constraints. Violation of atomicity constraints may occur only when an operation has two distinct accesses that conflict nondeterministically with other operations.

When a code has the single external access property, atomicity constraints can be ignored. It follows that the critical pair relation defined in Section 3.2 enforces correctness; this relation will coincide with the relation $\mathbf{D}_0$ defined in the current section.

Example. Consider the code in Figure 19. This code has the single external access property. There is one internal critical cycle, the cycle $(a1, b1, a1)$. It defines the internal delay $(a1, b1)$. There are two external critical cycles, $(a1, e1, a2, b2, a1)$ and $(c1, d1, c1)$ (remember that $\mathbf{P}$ is closed under $\mathbf{A}$). These cycles define the external delays $(a1, e1)$, $(c1, d1)$, and $(a2, b2)$. These delays enforce correct execution. The same delays would be obtained by ignoring $\mathbf{A}$ and computing critical pairs for $(\mathbf{P}, \mathbf{C})$.

4.3 Minimality

We prove in this subsection that the delay relation defined in Theorem 4.2 is minimal. The proof is similar to the proof of Theorem 3.7 (which handles the case where there are no atomicity constraints).

LEMMA 4.5 *Let σ be a critical cycle of $(\mathbf{P} \cup \mathbf{A}, \mathbf{C} - \mathbf{A})$, and let $S = \sigma - \mathbf{P} \cup \mathbf{A}$. Then S is a minimal inconsistent execution.*

PROOF. Clearly, S is not consistent with $\mathbf{P}$ and $\mathbf{A}$. We shall show that any proper subset of S is consistent with $\mathbf{P}$ and $\mathbf{A}$.

Let σ be the projection of σ. Then σ is a critical cycle of $(\mathbf{P/A}, \mathbf{C/A})$, be Lemma 4.1. We have $S/\mathbf{A} = \sigma - \mathbf{P/A}$, so that $S/\mathbf{A}$, the projection of S, is a minimal inconsistent execution (with respect to $\mathbf{P/A}$). If rs, uv are two distinct edges in S, then $\neg r\mathbf{A}u$, since σ has no $\mathbf{A}$ chords. Thus, no two edges from S are equivalent under $\mathbf{A}$, and the mapping from S to $S/\mathbf{A}$ is one-to-one.

Let S' be a proper subset of S. Then $S'/\mathbf{A}$ is a proper subset of $S/\mathbf{A}$ and, hence, consistent with $\mathbf{P/A}$. It follows that S' is consistent $\mathbf{P}$ and $\mathbf{A}$. □

THEOREM 4.6 *Let* $\mathbf{D}_0$ *be the delay relation defined in Theorem* 4.2. *Let* $R \subseteq \mathbf{P} \cup \mathbf{A}$ *be a relation such that for any minimal inconsistent execution* S *there is a simple cycle* π *of* $R \cup S$ *and* $\pi \not\subseteq R$. *Then* $\mathbf{D}_0$ *is contained in the transitive, irreflexive closure of* R.

COROLLARY 4.7 *Let* $R \subseteq \mathbf{P} \cup \mathbf{A}$ *be an acyclic relation that enforces correctness (with respect to* $\mathbf{P}$ *and* $\mathbf{A}$). *Then* $\mathbf{D}_0 \subseteq R^+$.

PROOF. Let S be a minimal inconsistent execution. Then S is inconsistent with R, and $S \cup R$ has a simple cycle σ. Since R is acyclic, then $\sigma \not\subseteq R$. Hence, R fulfills the condition of Theorem 4.6, so that $\mathbf{D}_0 \subseteq R^+$. □

PROOF OF THEOREM 4.6. Let $uv \in \mathbf{D}_0$. There are two cases to consider:

(i) $uv \in \mathbf{P} \cap \mathbf{A}$ occurs in a critical cycle σ of $(\mathbf{P} \cap \mathbf{A}, \mathbf{C} \cap \mathbf{A})$. Then $S = \sigma - \mathbf{P}$ is a minimal independent execution, by Corollary 3.6. Thus, $R \cup S$ contains a simple cycle π, and $\pi \not\subseteq R$. By Lemma 3.5 ii, it follows that $uv \in R^+$.

(ii) $uv \in \mathbf{P} \cup \mathbf{A}$ occurs in a critical cycle σ of $(\mathbf{P} \cup \mathbf{A}, \mathbf{C} - \mathbf{A})$. By Lemma 4.5, $S = \sigma - \mathbf{P} \cup \mathbf{A}$ is a minimal independent execution. Thus, $R \cup S$ contains a simple cycle π such that $\pi \not\subseteq R$. By Lemma 3.5 ii, this implies $uv \in R^+$. □

The last result implies that, if $\mathbf{D}_0$ contains a cycle, then there exists no acyclic delay relation $R \subseteq \mathbf{P} \cup \mathbf{A}$ that enforces correctness. Indeed, R^+ would contain $\mathbf{D}_0$ and, hence, would contain a cycle; but then R itself contains a cycle. Hence, when the delay relation defined by Theorem 4.2 is not enforceable, one either has to use delays that are outside $\mathbf{P} \cup \mathbf{A}$, or use a different mechanism to enforce correctness.

5. LARGE ATOMIC OPERATIONS AND LOCKS

In this section we consider the problem of enforcing correct execution for code when delays do not suffice. For example, we consider a concurrent execution of program segments where an operation may access more than one shared variable. We assume two mechanisms can be used to enforce correctness:

(1) An access may be delayed until some other access has terminated (these are the delays used in the previous sections).
(2) A locking protocol may be used to guarantee atomic execution of sets of accesses; the accesses in the code are partitioned into disjoint *locking sets*, and the protocol protects the execution of the accesses in each locking set. We represent that partition by an equivalence relation $\mathbf{L}$; the locking sets are the equivalence classes of $\mathbf{L}$.

Atomic execution of sets of accesses can be obtained using *locks*. A lock is set on a variable on behalf of a locking set. Several *read locks* may be set simultaneously on a variable; a write lock on a variable is exclusive of any other lock. We use the following protocol to execute these accesses:

Let **u** be a locking set. A lock on behalf of **u** is set on all variables accessed by **u** before the execution of any access from **u**. A read lock is secured on each variable for which there are only read accesses in **u**; a write lock is secured on each variable that is updated in **u**. After execution of all accesses in **u**, the locks are released.

This protocol guarantees that a locking set accesses a variable only if it has a lock on it, and a locking set updates a variable only if it has a write lock on it. (If all the locks cannot be obtained, they are released, and we try again, avoiding the possibility of deadlock.)

A locking protocol can be combined with the enforcement of a delay relation **D**. We assume **D** and **D/L** are acyclic. Delays on locking sets are enforced by suitably ordering the acquisition and release of locks: If **uD/Lv**, locks on behalf of **v** are secured only after all locks on behalf of **u** have been released. Delays within locking sets are enforced by suitably ordering the accesses.

Note that if $u\mathbf{D}v$ and $\neg u\mathbf{L}v$ then each access in the locking set of v is delayed until all accesses in the locking set of u have taken place. This enforces a set $\overline{\mathbf{D}}$ of delays that may be larger than **D**; $\overline{\mathbf{D}}$ is the closure of **D** under **L**; that is, $\overline{\mathbf{D}} = \mathbf{D/L} \times (\mathbf{D} \cap \mathbf{L})$, where $\times$ is the lexicographic product defined in Section 2.1.

DELAY AND LOCKING LEMMA. *Assume the previous locking protocol is used for* **D** *and* **L**. *Then any execution* **E** *is consistent with* **D** *and* **L**.

PROOF. Assume **E** is not consistent with **D** and **L**. According to Lemma 2.1, there are two cases to consider:

(i) $(\mathbf{E} \cap \mathbf{L}) \cup (\mathbf{D} \cap \mathbf{L})$ has a cycle. Then **E** is not consistent with **D**, which is impossible by the delay lemma.

(ii) $\mathbf{E} \cup \mathbf{L} \cup \mathbf{D}$ has a cycle $\sigma = (v_0, \ldots, v_{n-1}, v_0)$ that is not contained in an equivalence class of **L**. If $v_i\mathbf{E}v_{i+1}$ then access v_{i+1} occurs after access v_i; the same holds true if $v_i\mathbf{D}v_{i+1}$.

If $v_{i-1}\mathbf{L}v_i\mathbf{E}v_{i+1}$ and $\neg v_i\mathbf{L}v_{i+1}$, then v_i and v_{i+1} are conflicting accesses and require the acquisition of conflicting locks. The lock on behalf of $[v_i]$ is secured before v_{i-1} occurs and released after v_i occurs. It follows that either both v_{i-1} and v_i occur before v_{i+1} or both occur after v_{i+1}. Since $v_i\mathbf{E}v_{i+1}$, the former is the case. Similarly, if $v_{i-1}\mathbf{E}v_i\mathbf{L}v_{i+1}$ and $\neg v_{i-1}\mathbf{L}v_i$, then v_{i-1} and v_i both occur before v_{i+1}.

If $v_{i-1}\mathbf{L}v_i\mathbf{D}v_{i+1}$ and $\neg v_i\mathbf{L}v_{i+1}$, then the locks acquired on behalf of $[v_i]$ are released after v_{i-1} occurred, but before v_{i+1} occurs. Hence, v_{i-1} and v_i both occur before v_{i+1}. Similarly, if $v_{i-1}\mathbf{D}v_i\mathbf{L}v_{i+1}$ and $\neg v_{i-1}\mathbf{L}v_i$, then v_{i-1} and v_i both occur before v_{i+1}.

Let $v_{i_0}, \ldots, v_{i_{r-1}}$ be the subset of nodes in σ such that $\neg v_{i_j}\mathbf{L}v_{i_j+1}$. Since σ is not contained in one locking set, this list has length ≥ 2. The previous argument implies that v_{i_j} occurs before $v_{i_{j+1}}$, $j = 0, \ldots, r-1$ (addition is taken modulo r); this is a contradiction since "occurs before" is irreflexive and transitive. □

The pair of relations (**D**, **L**) can be seen as a specification of the effect of some control mechanism: It restricts the executions to those that are consistent with **L** and **D**. The following discussion does not make any assumption on the protocol used; our results hold for any protocol for which the delay and locking lemma is valid. (**D** and **D**/**L** should be acyclic, but it is not required that **D** be closed under **L**.)

The problem is formalized as follows: As before, a program is represented by a tuple $\langle \mathbf{V}, \mathbf{A}, \mathbf{P}, \mathbf{C} \rangle$. An execution **E** is a proper orientation on **C**. Correctness of execution is enforced using a delay relation **D** and a locking relation **L**.

The pair of relations (**D**, **L**) *enforces correctness* if any execution order that is consistent with **D** and **L** is correct (i.e., consistent with **P** and **A**). Such a pair always exists: (**A**, **P**) enforces correctness. We show in this section how to improve on this trivial solution. Following the spirit of the previous sections, we seek a solution (**D**, **L**) such that $\mathbf{L} \subseteq \mathbf{A}$ and (i) $\mathbf{D} \subseteq \mathbf{P}$, or (ii) $\mathbf{D} \subseteq \mathbf{P} \cup \mathbf{A}$, and **D** is acyclic. In case (i) no extraneous atomicity or order constraints are imposed on the accesses; any execution order that is correct is consistent with **D** and **L**. In case (ii) the constraint imposed by **D** and **L** may rule out correct computations, but are still easy to enforce in the case that interests us in particular; that is, when the code consists of the disjoint union of chains (serial program segments) and each operation is contained in a segment (an atomic "operation" consists of code executed by one processor). Accesses within a locking set are executed by one processor; this processor will acquire and release locks for this access; it is sufficient to label a lock with the identifier of the processor that acquires it. Delays also occur between accesses executed by the same processor and are enforced by the control logic of that processor.

The major constraint on concurrency is locking; this is also likely to be the more expensive operation. Therefore, we shall seek a solution that is lexicographically minimal: **L** is the smallest possible locking relation, and given **L**, **D** is the smallest possible delay relation.

LEMMA 5.1 *Let* $\mathbf{L} \subseteq \mathbf{A}$ *be a locking relation, and* $\mathbf{D} \subseteq \mathbf{P} \cup \mathbf{A}$ *be a delay relation such that* (**L**, **D**) *enforces correctness. Then* $\mathbf{D}_0 \subseteq (\mathbf{L} \cup \mathbf{D})^+$, *where* $\mathbf{D}_0$ *is the relation defined in Theorem* 4.2.

PROOF. Let S be a minimal inconsistent execution. Then S is inconsistent with **D** and **L**. Thus, either $S \subseteq \mathbf{L}$, and S is contained in a simple cycle of $(S \cap \mathbf{L}) \cup (\mathbf{D} \cap \mathbf{L})$, or $S \cap \mathbf{L} = \varnothing$, and S is contained in a simple cycle of $S \cup \mathbf{L} \cup \mathbf{D}$. In either case S is contained in a simple cycle of $S \cup (\mathbf{D} \cup \mathbf{L})$. Since $\mathbf{D} \cup \mathbf{L} \subseteq \mathbf{P} \cup \mathbf{A}$, this implies, by Theorem 4.6, that $\mathbf{D}_0 \subseteq (\mathbf{D} \cup \mathbf{L})^+$. □

Let $\mathbf{L} \subseteq \mathbf{A}$ be a fixed locking relation. The previous lemma suggests the following construction for **D**, a delay relation that enforces correctness together with **L**:

(5.1) Let $\mathbf{D}^{\mathbf{L}} = \mathbf{D}_0 - \mathbf{L}$,

(5.2) let $\mathbf{D}_{\mathbf{L}}$ be the set of **P** edges on critical cycles of $(\mathbf{P} \cap \mathbf{L}, \mathbf{C} \cap \mathbf{L})$, and

(5.3) let $\mathbf{D}(\mathbf{L}) = \mathbf{D}^{\mathbf{L}} \cup \mathbf{D}_{\mathbf{L}}$.

Note that $\mathbf{D}^{\mathbf{L}} \subseteq \mathbf{D}_0 \subseteq \mathbf{P} \cup \mathbf{A}$ and $\mathbf{D}_{\mathbf{L}} \subseteq \mathbf{P} \cap \mathbf{L}$; hence, $\mathbf{D}(\mathbf{L}) \subseteq \mathbf{P} \cup \mathbf{A}$. We now present the following theorem:

THEOREM 5.2 *Let* $\mathbf{L} \subseteq \mathbf{A}$ *be a locking relation, and let* $\mathbf{D}(\mathbf{L})$ *be defined by* (5.1)–(5.3). *Then,*

(*i*) *the pair of relations* $(\mathbf{D}(\mathbf{L}), \mathbf{L})$ *enforces correctness; and*

(*ii*) *let* $\mathbf{D}' \subseteq \mathbf{P} \cup \mathbf{A}$ *be an acyclic relation such that* $(\mathbf{D}', \mathbf{L})$ *enforces correctness; then* $\mathbf{L} \cup \mathbf{D}(\mathbf{L}) \subseteq (\mathbf{L} \cup \mathbf{D}')^+$; *and if we furthermore have* $\mathbf{L} \cap \mathbf{D}' \subseteq \mathbf{P}$, *then* $(\mathbf{L} \cap \mathbf{D}(\mathbf{L})) \subseteq (\mathbf{L} \cap \mathbf{D}')^+$.

PROOF. (i) Let S be a minimal inconsistent execution. By Theorem 4.2, S is contained in a simple cycle σ of $S \cup \mathbf{D}_0$. Since $\mathbf{D}_0 \subseteq \mathbf{D}(\mathbf{L}) \cup \mathbf{L}$, σ is a cycle of $S \cup \mathbf{D}(\mathbf{L}) \cup \mathbf{L}$. If $\sigma \not\subseteq \mathbf{L}$ then, according to Lemma 2.1, S is not consistent with $\mathbf{D}(\mathbf{L})$ and $\mathbf{L}$. If $\sigma \subseteq \mathbf{L}$ then $S \subseteq \mathbf{L} \subseteq \mathbf{A}$. This implies that S is not consistent with $\mathbf{P}$. But, if $S \cup \mathbf{P}$ has a cycle, then $S \cup (\mathbf{P} \cap \mathbf{L})$ has a cycle. Thus, S is not consistent with $\mathbf{P} \cap \mathbf{L}$ so that, by Corollary 3.4, S is not consistent with $\mathbf{D}_{\mathbf{L}}$ and, hence, with $\mathbf{D}(\mathbf{L})$. It follows that $(\mathbf{D}(\mathbf{L}), \mathbf{L})$ enforces correctness.

(ii) According to Theorem 5.1, we have $\mathbf{D}_0 \subseteq (\mathbf{L} \cup \mathbf{D}')^+$. It follows that $\mathbf{L} \cup \mathbf{D}(\mathbf{L}) = \mathbf{L} \cup \mathbf{D}_0 \subseteq (\mathbf{D}' \cup \mathbf{L})^+$. Assume $\mathbf{D}' \cap \mathbf{L} \subseteq \mathbf{P}$. $\mathbf{D}' \cap \mathbf{L}$ enforces consistency with $\mathbf{P} \cap \mathbf{L}$. This implies, by Theorem 3.7, that $\mathbf{D}' \cap \mathbf{L} \subseteq (\mathbf{D}_{\mathbf{L}})^+ \subseteq (\mathbf{D}(\mathbf{L}) \cap \mathbf{L})^+$. □

The last theorem shows that $\mathbf{D}(\mathbf{L})$ is a minimal delay relation that enforces correctness, together with $\mathbf{L}$. We show now how to construct a minimal locking relation $\mathbf{L}$. Suppose we require that $\mathbf{D}(\mathbf{L}) \subseteq \mathbf{P}$.

THEOREM 5.3 *Let* $\mathbf{L}$ *be the symmetric, transitive closure of* $\mathbf{D}_0 - \mathbf{P}$. *Let* $\mathbf{D}(\mathbf{L})$ *be defined by* (5.1)–(5.3). *Then,*

(*i*) $\mathbf{L} \subseteq \mathbf{A}$, $\mathbf{D}(\mathbf{L}) \subseteq \mathbf{P}$, *and* $(\mathbf{L}, \mathbf{D}(\mathbf{L}))$ *enforce correctness; and*

(*ii*) *if* $(\mathbf{L}', \mathbf{D}')$ *enforces correctness, where* $\mathbf{L}' \subseteq \mathbf{A}$ *and* $\mathbf{D}' \subseteq \mathbf{P}$, *then* $\mathbf{L} \subseteq \mathbf{L}'$.

PROOF. (i) We have $\mathbf{D}_0 \subseteq \mathbf{P} \cup \mathbf{A}$; hence, $\mathbf{D}_0 - \mathbf{P} \subseteq \mathbf{A}$, so that $\mathbf{L} \subseteq \mathbf{A}$. We have $\mathbf{D}^{\mathbf{L}} = \mathbf{D}_0 - \mathbf{L} \subseteq \mathbf{P}$, so that $\mathbf{D}(\mathbf{L}) \subseteq \mathbf{P}$. According to Theorem 5.2, $(\mathbf{D}(\mathbf{L}), \mathbf{L})$ enforces correctness.

(ii) We have, according to Lemma 5.1, $\mathbf{D}_0 \subseteq (\mathbf{L}' \cup \mathbf{D}')^+$. But $(\mathbf{L}' \cup \mathbf{D}')^+ \subseteq \mathbf{L}' \cup P$; thus, $\mathbf{D}_0 - \mathbf{P} \subseteq \mathbf{L}'$. It follows that $\mathbf{L} \subseteq \mathbf{L}'$. □

Suppose we merely require that $\mathbf{D}(\mathbf{L})$ be enforceable; that is, $\mathbf{D}(\mathbf{L})$ and $\mathbf{D}(\mathbf{L})/\mathbf{L}$ are acyclic.

THEOREM 5.4 *Let* $\mathbf{L}$ *be the relation defined by* $u\mathbf{L}v$ *if* u *and* v *are in the same strongly connected component of* $\mathbf{D}_0$. ($u\mathbf{L}v$ *if both* $uv \in \mathbf{D}_0^+$ *and* $vu \in \mathbf{D}_0^+$.) *Let* $\mathbf{D}(\mathbf{L})$ *be defined by* (5.1)–(5.3). *Then,*

(*i*) $\mathbf{L} \subseteq \mathbf{A}$, $\mathbf{D}(\mathbf{L}) \subseteq \mathbf{P} \cup \mathbf{A}$, *and* $\mathbf{D}(\mathbf{L})$ *and* $\mathbf{D}(\mathbf{L})/\mathbf{L}$ *are acyclic*; *and* $(\mathbf{D}(\mathbf{L}), \mathbf{L})$ *enforces correctness; and*

(*ii*) *if* $(\mathbf{D}', \mathbf{L}')$ *enforces correctness, where* $\mathbf{L}' \subseteq \mathbf{A}$, $\mathbf{D}' \subseteq \mathbf{P} \cup \mathbf{A}$, *and both* $\mathbf{D}'$ *and* $\mathbf{D}'/\mathbf{L}$ *are acyclic, then* $\mathbf{L} \subseteq \mathbf{L}'$.

PROOF. (i) We have $\mathbf{D}_0 \subseteq \mathbf{P} \cup \mathbf{A}$. Any strongly connected component of $\mathbf{D}_0$ is contained in a strongly connected component of $\mathbf{P} \cup \mathbf{A}$; since $\mathbf{P}/\mathbf{A}$ is a partial

order, any strongly connected component of $\mathbf{P} \cup \mathbf{A}$ is contained in an equivalence class of $\mathbf{A}$. It follows that $\mathbf{L} \subseteq \mathbf{A}$. A graph induces an acyclic orientation on its strongly connected components; hence, $\mathbf{D}(\mathbf{L})/\mathbf{L} = \mathbf{D}_0/\mathbf{L}$ is acyclic. Also, $\mathbf{D}(\mathbf{L}) \cap \mathbf{L} = \mathbf{D}_L \subseteq \mathbf{P}$ is acyclic. It follows that $\mathbf{D}(\mathbf{L})$ is acyclic. By Theorem 5.2, $(\mathbf{D}(\mathbf{L}), \mathbf{L})$ enforce correctness.

(ii) We have, by Lemma 5.1, $(\mathbf{D}_0)^+ \subseteq (\mathbf{L}' \cup \mathbf{D}')^+$. Let $\bar{\mathbf{D}}$ be the closure of $\mathbf{D}'$ under $\mathbf{L}'$; that is, $\bar{\mathbf{D}} = \mathbf{D}'/\mathbf{L}' \times \mathbf{D}' \cap \mathbf{L}$. Then $\bar{\mathbf{D}}$ is acyclic, and $(\mathbf{L}' \cup \mathbf{D}')^+ = \mathbf{L}' \cup (\bar{\mathbf{D}})^+$. Assume both $uv \in \mathbf{D}_0^+$ and $vu \in \mathbf{D}_0^+$. Since $(\bar{\mathbf{D}})^+$ is acyclic, either $uv \in \mathbf{L}'$, or $vu \in \mathbf{L}'$. It follows that $\mathbf{L} \subseteq \mathbf{L}'$. □

Example. Consider anew the code in Figure 18. We have $\mathbf{D}_0 = \{(a1, b1), (a2, b2)\}$. If we insist that $\mathbf{D}(\mathbf{L}) \subseteq \mathbf{P}$, we take $\mathbf{L}$ to be the symmetric, transitive closure of $\mathbf{D}_0 - \mathbf{P} = (a1, b1)$. The locking sets are $\{a1, b1\}$, $\{a2\}$, and $\{b2\}$, and there is one delay pair, $(a2, b2)$.

On the other hand, if we allow delays in $\mathbf{A} - \mathbf{P}$, then we take the locking sets to be the strongly connected components of $\mathbf{D}_0$. In this case $\mathbf{D}_0$ is acyclic, so that all locking sets are singletons, and $\mathbf{D}(\mathbf{L}) = \{(a1, b1), (a2, b2)\}$. Note that this delay relation prevents some correct executions, such as $a2, b2, b1, a1$.

6. FROM ABSTRACT CODE TO REAL PROGRAMS

6.1 Delay and Locking Mechanisms

The preceding sections provided a framework for the detection and prevention of hazards in concurrent code. The actual use of this framework will depend on the extent of information that can be extracted from the code at compile time, and on the control mechanisms provided by the machine. We shall first address the second factor, and then turn to the first.

Suppose that a delay relation $\mathbf{D}$ that enforces correctness has been computed, and suppose that $u\mathbf{D}v$, $v\mathbf{D}w$, and $u\mathbf{D}w$. Then it is not necessary to record and enforce the delay $u\mathbf{D}w$, as it will be implied by the delays $u\mathbf{D}v$ and $v\mathbf{D}w$. In general, any delay relation $\mathbf{D}'$ such that $\mathbf{D}$ is contained in the transitive closure of $\mathbf{D}'$ will do. In particular, one can take the *transitive reduction* of $\mathbf{D}$, which is the smallest relation R with the property that $R \subseteq \mathbf{D} \subseteq R^+$: R consists of all pairs $uv \in \mathbf{D}$ such that the longest path from u to v in the graph of $\mathbf{D}$ has length 1 [2]. The same idea applies in the general case, where locks are used.

In some cases the hardware (or firmware) of the machine may impose a set $\mathbf{D}_1$ of delays; for example, the hardware may ensure correct ordering of accesses within atomic operations. One needs then to find a minimal relation $\mathbf{D}_2$ such that $\mathbf{D}_1 \cup \mathbf{D}_2$ enforces correctness. Let $\mathbf{D}$ be a minimal delay relation that enforces correctness, let $\mathbf{D}' = (\mathbf{D} \cup \mathbf{D}_1)^+$, and let R' be the transitive reduction of $\mathbf{D}'$. Finally, let $\mathbf{D}_2 = R' - \mathbf{D}_1$. Then $\mathbf{D}_2$ is a minimal delay relation that enforces correctness, together with $\mathbf{D}_1$.

On the other hand, the control mechanisms of the computer may not enforce arbitrary delays. We then look for a minimal delay relation that is enforceable by the machine, and contains the relation $\mathbf{D}$ in its transitive closure. For example, the RP3 has a fence instruction that delays the execution of the following memory access until all preceding accesses have executed. We wish to minimize the number of "fences" used.

The fences in the code of a processor divide the instructions of this processor into a disjoint sequence $S_0, \ldots, S_r$ of sets such that all accesses in S_i are executed before any access in S_{i+1}; r is the number of fences used. Thus, given a delay relation **D**, we wish to find a minimal partition such that, if $u\mathbf{D}v$, $u \in S_i$, $v \in S_j$, then $i < j$. Such a partition is easy to compute: Let $level(u)$ be the length of the longest path reaching u in the graph of **D**. Define S_i to be the set of nodes at level i. The partition $S_1, S_2, \ldots$ then has the required property. The number of fences required equals the length of the longest path in the graph of **D**, and this is optimal.

Computers may also differ in their support for locking protocols. Locking may often be optimized when locking sets contain a unique access. Since such a locking set is executed atomically, it is sufficient to check that the location accessed by that operation is not currently locked by another processor. (In fact, we can do even better: If the operation only reads the location, then it is only necessary to check that no other processor holds a write lock on that location.) This has the same effect as a lock, access, and unlock, so our previous theorems still hold with this optimization. If memory is tagged, and hardware supports "test-and-load" and "test-and-store" operations, then only one access to memory is needed. In the even more special case when a location is always accessed by an equivalence class of size 1, no locks on that location are needed at all.

6.2 Conflicts and Branching Programs

We assumed the conflict relation **C** is known in advance. More often, a compiler can do only a partial job in extracting this relation: Data dependency analysis yields a set of pairs of instructions that contains all conflict pairs; we use this set as our conflict relation, and perform a conservative, safe optimization.

More important, we have assumed that code is straight line; in the general case, our program segments will contain jump statements, due to branch and loop constructs. We represent possible control flow in each program segment by a flow graph (see [1]). An execution of a program corresponds to a (possibly self-crossing) path in the flow graph.

To simplify the discussion, we consider code with no atomicity constraints; storage access operations are regular reads and writes. One can reduce this general setting to the case of straight-line code by requiring that the execution of a new block does not start until memory references issued by instructions in other blocks of this program segment have been satisfied: A fence is set at the entry of each block. For each tuple of blocks, each belonging to another program segment, delays are introduced that enforce correct concurrent execution of these blocks.

It is also possible to do global optimization of delays. Delays are introduced between pairs of instructions; each such instruction may have several executing instances. If a delay is inserted between instructions u and v, then instruction v is not issued as long as there is an executing instance of instruction u; a delay $u\mathbf{D}u$ indicates that instruction u is not issued as long as the execution of the previous instance of u has not terminated. A delay relation **D** *enforces correctness* if for any choice of an execution path within each program segment the concurrent execution of these paths is correct.

Let $u\mathbf{P}v$ if u and v are instructions in the same program segment, and there is a path from u to v in the flow graph of this program. $\mathbf{P}$ is transitive. $\mathbf{P}$, however, may not be acyclic; in particular, we may have $u\mathbf{P}u$. The conflict relation $\mathbf{C}$ is defined as usual.

An execution path of a program segment corresponds to a (possibly self-crossing) path in the graph of $\mathbf{P}$. A critical cycle σ for a set of parallel execution paths corresponds to a cycle σ in $\mathbf{P} \cup \mathbf{C}$. The cycle σ is not necessarily simple. The delay relation $\mathbf{D}$ enforces correctness if all $\mathbf{P}$ edges of such cycles occur in $\mathbf{D}$. We may use the characterization given in Theorem 3.9 to identify these cycles.

THEOREM 6.1 *Let* $\mathbf{D}$ *be the set of* $\mathbf{P}$ *edges in cycles* σ *of* $\mathbf{P} \cup \mathbf{C}$ *with the following properties*:

(*i*) σ *contains at most two instructions from the same program segment; these instructions are consecutive in* σ.

(*ii*) σ *contains either zero, two, or three accesses to each variable; these accesses are consecutive in* σ. *The possible configurations are read* $x \rightarrow$ *write* x, *write* $x \rightarrow$ *read* x, *write* $x \rightarrow$ *write* x, *or read* $x \rightarrow$ *write* $x \rightarrow$ *read* x.

Then $\mathbf{D}$ *enforces correctness.*

It may not always be possible to resolve references and exactly identify conflicting accesses. In this case, one has to consider all cycles that have the above form, for some possible resolution of the references.

6.3 Code Motion

The delay analysis of the previous sections can be helpful even if the computer enforces sequential consistency, that is, there is no overlapping of successive memory accesses. The performance of the code can be enhanced by changing the order in which operations are executed. Indeed, code motion is a standard optimization technique for sequential code: Computations are moved out of loops, memory loads are executed earlier in order to mask memory latency, etc. Such optimizations are safe if they do not change the order in which conflicting operations are executed.

Our analysis of parallel code shows that this safety criterion is not sufficient any more when a program segment interacts with others via shared variables. For example, in the code shown in Figure 6 there are no data dependencies within the program segments; if we reverse the order of the operations in the second segment, we get the code shown in Figure 7. A (sequentially consistent) execution of this second code may yield an outcome that is incorrect for the first code.

A naive approach to local optimization would be to assume that any two accesses to shared variables potentially conflict and the order of two such operations cannot be reversed. The delay analysis yields a more accurate criterion: If there is no delay between operations u and v of the same program segment, then the order of execution of these two operations in memory is arbitrary. In particular, we can interchange the order they are issued. Any order of issuing that satisfies the requirement that if $u\mathbf{D}^+v$ then v is not issued until u has been completed is acceptable.

Segment 1	*Segment 2*	*Segment 3*
a1: read A	a2: read B	a3: read C
barrier	barrier	barrier
b1: write B	b2: write B	b3: write D

Fig. 20. Barrier example.

6.4 Synchronization Operations

Our abstract code model can be used to represent synchronization operations across processes; these are represented as order constraints between operations in distinct program segments. For example, suppose a communication event between two processes is represented by two operations: a *send* operation in one segment and a *receive* operation in another segment. If the communication mechanism is asynchronous, then we have the added constraint

$send\mathbf{P}receive.$

If the communication is synchronous, we have the following stronger constraints:

$u\mathbf{P}send$ iff $u\mathbf{P}receive$, and
$send\mathbf{P}u$ iff $receive\mathbf{P}u$.

This extends to a *barrier* that is a synchronization involving an arbitrary number of processors. If $b_1, \ldots, b_k$ are the barrier synchronization operations in the different processes, then we have

$u\mathbf{P}b_i$ iff $u\mathbf{P}b_j$, and
$b_i\mathbf{P}u$ iff $b_j\mathbf{P}u$.

Consider the example in Figure 20. The barrier implies that

$$\mathbf{P} = \{(a1, b1), (a2, b2), (a3, b3)\} \cup \{(a1, b2), (a1, b3), (a2, b1), (a2, b3), (a3, b1), (a3, b2)\}.$$

where the second set of **P** relationships comes from the barrier. The **P** ordering can be implemented by creating the delay relation $\mathbf{D} = \{(a2, b1), (a2, b2)\}$.

We thus obtain a weaker synchronization constraint than suggested by **P**; one only needs to delay $b1$ and $b2$ until $a2$ has terminated. This can be used to reduce the strength of the synchronization operations, thus increasing concurrency and decreasing synchronization overhead, without changing the meaning of the program.

7. CONCLUSION

This paper presents a method of enforcing efficient and sequentially consistent execution of concurrent processes on a shared-memory multiprocessor when memory access is asynchronous. Our method determines when consecutive operations in the same program segment of a parallel program may execute concurrently, without violating the programmer's view that each segment

executes in its given program order. An actual implementation of this method depends on two factors that were not discussed.

First, one should be able to detect data dependencies. This is a problem faced by any optimizing compiler, and sophisticated methods have been developed for that purpose [1, 15]. In particular, index analysis to discover data dependencies across loop iterations [3] are relevant to our purpose. In general, the compiler will not be able to detect existing data dependencies accurately, but will have to assume further dependencies. These extra data dependencies reduce the efficiency of the code produced, but do not affect its correctness.

Second, one has to find all the minimal cycles in a graph. This requires time exponential in the number of nodes in a general graph. The graphs arising from program segments, however, have a constrained structure that makes the problem easier. The characterization of critical cycles given in Theorem 3.9 implies the existence of a polynomial-time algorithm for detection of critical pairs in a code that consists of a fixed number of serial program segments; this algorithm extends to the code obtained from a fixed number of high-level language program segments with bounded nesting of loops and conditionals.

Our analysis presupposes that the processor has the ability to delay the issuing of an instruction until some previous instruction has been executed. Pipelined processors often have such locking mechanisms. The processor detects data dependencies by itself and enforces correct sequencing of data-dependent operations.

The results of this paper suggest that such a mechanism is inadequate in a shared-memory parallel computer. Instead, it would be useful to be able to tag the serial machine code with dependency information; the processor would enforce correct sequencing of tagged instructions. Such a mechanism could be used to enforce both data dependencies in serial code, and delays needed to avoid hazards in parallel code.

ACKNOWLEDGMENTS

Many thanks to Boris Aronov and to Allan Gottlieb for critical comments on earlier drafts. Many thanks also to the referees for their comments on the original manuscript.

REFERENCES

1. AHO, A., SETHI, R., AND ULLMAN, J. *Compilers Principles, Techniques and Tools.* Addison-Wesley, Reading, Mass., 1986.
2. AHO, A. V., GAREY, M. R., AND ULLMAN, J. D. The transitive reduction of a directed graph. *SIAM J. Comput. 1*, 2 (Dec. 1972), 131–137.
3. ALLEN, J. R., AND KENNEDY, K. Automatic translation of Fortran programs to vector form. COMP TR84-9, Dept. of Computer Science, Rice Univ., Houston, Tex., July 1984.
4. BEERI, C., BERNSTEIN, P. A., AND GOODMAN, N. A model for nested transaction systems. Tech. Rep. CS-86-1, Computer Science Dept., Hebrew Univ., Jerusalem, 1986.
5. BERNSTEIN, P. A., AND GOODMAN, N. Concurrency control in distributed database systems. *ACM Comput. Surv. 13*, 2 (June 1981), 185–221.
6. BERNSTEIN, P. A., SHIPMAN, D. W., AND ROTHNIE, J. B., JR. Concurrency control in a system for distributed databases (SDD-1). *ACM Trans. Database Syst. 5*, 1 (Mar. 1980), 18–51.
7. COLLIER, W. Principles of architecture for systems of parallel processes. Tech. Rep. TR00.3100, IBM, T. J. Watson Research Center, Yorktown Heights, N.Y., Mar. 1981.

8. EDLER, J., GOTTLIEB, A., KRUSKAL, C. K., MCAULIFFE, K. P., RUDOLPH, L., SNIR, M., TELLER, P., AND WILSON, J. Issues related to MIMD, shared-memory computers: The NYU Ultracomputer approach. In *Proceedings of the 12th International Conference on Computer Architecture* (Boston, Mass., June). IEEE Press, New York, 1985, pp. 126–135.
9. GOTTLIEB, A., GRISHMAN, R., KRUSKAL, C. K., MCAULIFFE, K. P., RUDOLPH, L., AND SNIR, M. The NYU Ultracomputer—Designing a MIMD, shared-memory parallel computer. *IEEE Trans. Comput. C-32*, 3 (Feb. 1983), 175–189.
10. KUNG, H. T., AND PAPADIMITRIOU, C. H. An optimality theory of concurrency control for databases. In *Proceedings of the ACM–SIGMOD International Conference on Management of Data* (Boston, May 1979). ACM, New York, 1979, pp. 116–126.
11. LAMPORT, L. Time, clocks and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (July 1978), 558–565.
12. LAMPORT, L. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput. C-28*, 9 (Sept. 1979), 690–691.
13. LAMPORT, L. On interprocess communication, parts I and II. *Dist. Comput. 1* (Jan. 1986), 77–101.
14. LYNCH, N. A., AND FISHER, M. J. Semantics of concurrent computations. *Theor. Comput. Sci. 13*, 1 (Jan. 1981), 17–43.
15. PADUA, D. A., AND WOLFE, W. J. Advanced compiler optimizations for supercomputers. *Commun. ACM 29*, 12 (Dec. 1986), 1184–1201.
16. PFISTER, G. F., BRANTLEY, W. C., GEORGE, D. A., HARVEY, S. L., KLEINFELDER, W. J., MCAULIFFE, K. P., MELTON, E. A., NORTON, V. A., AND WEISS, J. The IBM research parallel processor prototype (RP3): Introduction and architecture. In *Proceedings of the IEEE 1985 International Conference on Parallel Processing* (Boston, June 1985). IEEE Press, New York, 1985, pp. 764–771.

Received March 1986; revised July 1987; accepted December 1987