Efficient and Correct Execution of Parallel Programs

DENNIS SHASHA & MARC SNIR

1988

CONTENTS

INTRODUCTION

- Programming paradigm for serial computers is well-established and easy to describe.

- Not the case for parallel machines (multiple processors computers with a shared memory).

- Sequential Consistency is introduced in 1979, and we shall use it.

- SC in a nutshell:

> 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].

Sequential Consistency – a Reminder Ex.

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

Not SC: test&set1(LOCK) read1(X) reset1(LOCK) test&set2(LOCK) read2(X) write1(X) write2(X) reset2(LOCK).

reset1(LOCK) and write1(x) executed out of order

Our Goals

Our model uses code delays and locks in order to achieve concurrency. Under the constraints of SC we state our goals:

- 1. Determine the minimal set of delays that enforce correctness.
- 2. Minimize the number of locks & delays required to allow “high-level” atomic operations on parallel instruction-based machines (e.g. array access).

We do these by taking advantage of insight gathered in compile time.

Basic Notions & Informal Definitions

1. Our program consists of ***instructions***.

2. Instructions specify/compose ***operations***.

3. Every operation accesses one or more storage locations, called ***variables***.

4. An access is a either a ***write*** or a ***read***. Operations communicate solely via accesses.

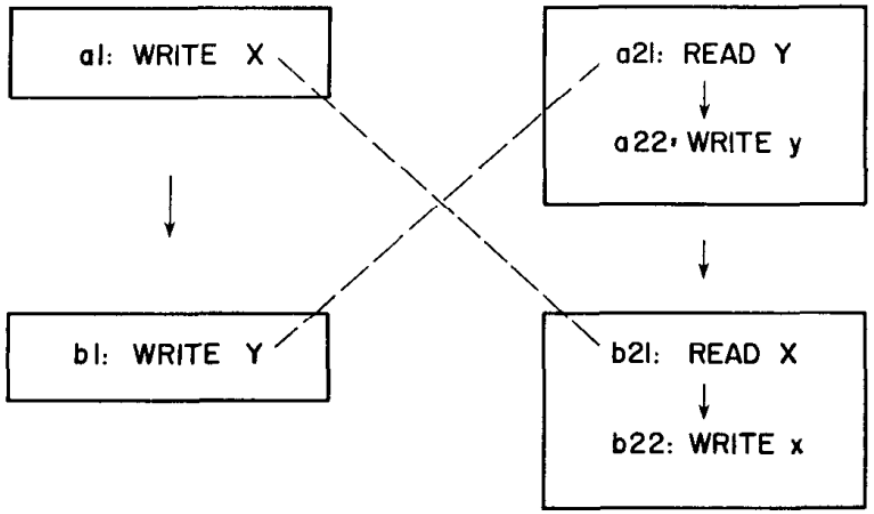
5. Two accesses ***conflict*** if they access the same variable and at least one of them is a write (unless they commute).

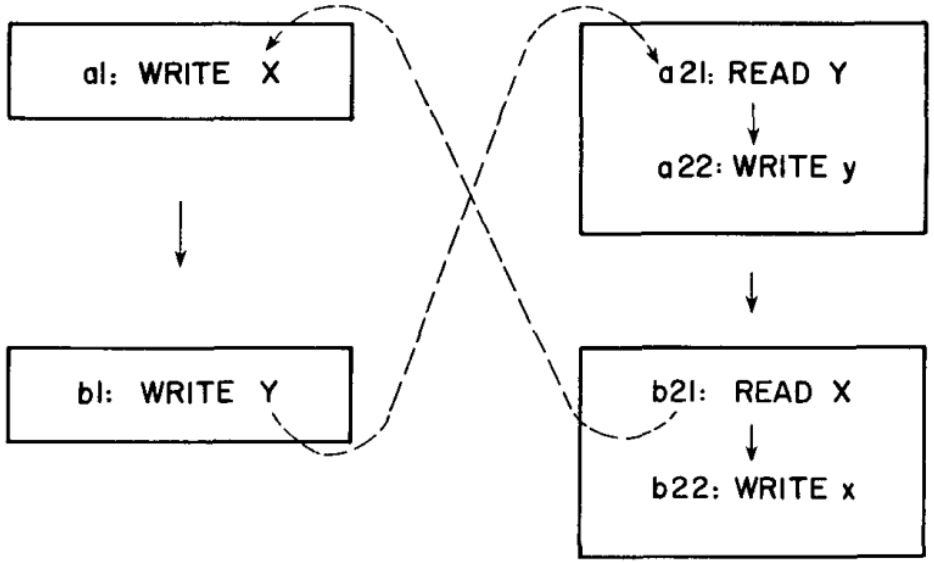
6. The ***execution order*** specifies the order of conflicting accesses.

7. The ***program order*** is a fixed ordering of the instructions. The execution order must respect it.

8. A computation is ***correct*** if operations appear to execute atomically, in the order specified by the program order.

A Glimpse at a Program





Full arrows – Program Order

Broken lines – Conflict Edges

Boxes - Operations

Directed Broken Lines – Execution Order

Diving into Formalism

Before presenting our model we would need a few basic definitions:

1. An order is an irreflexive, asymmetric, transitive relation. (e.g. >)

2. Let **C** be a symmetric relation. The relation **E** is a proper orientation of **C** if whenever u**C**v then either u**E**v or v**E**u.

3. Two relations **P** and **R** are ***consistent*** if PUR can be extended to a total order (their graph has no cycles).

Definitions – Cont.

Another basic definition, as is from the the paper:

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 $v\bar{E}w$.

Our Model

A code is a tuple <**V, A, P, C**>.

1. **V –** The set of variable accesses.

2. **A –** An equivalence relation on **V**, each equivalence class represents some atomic operation.

3. **P** – The program order. It is a partial order on **V**. **P** is close under **A:**

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

4. **C –** The conflict relation on **V**. Symmetric, irreflexive, but not necessarily transitive.

The Execution Order

In addition to <**V, A, P, C**>, we introduce **E –** The execution order.

E represents the execution order. It is a proper orientation of **C**, meaning u**E**v implies access **u** occurs in storage before access **v**, for any u,v that conflict (u**C**v).

An execution order **E** is ***correct*** if it is consistent with **P** and **A**:

- **E** can be extended to a total order on **C** that respects **P** (PUE has no cycles).

- Accesses that belong to one operation (contained in a class of **A**) execute indivisibly.

Example

Segment 1

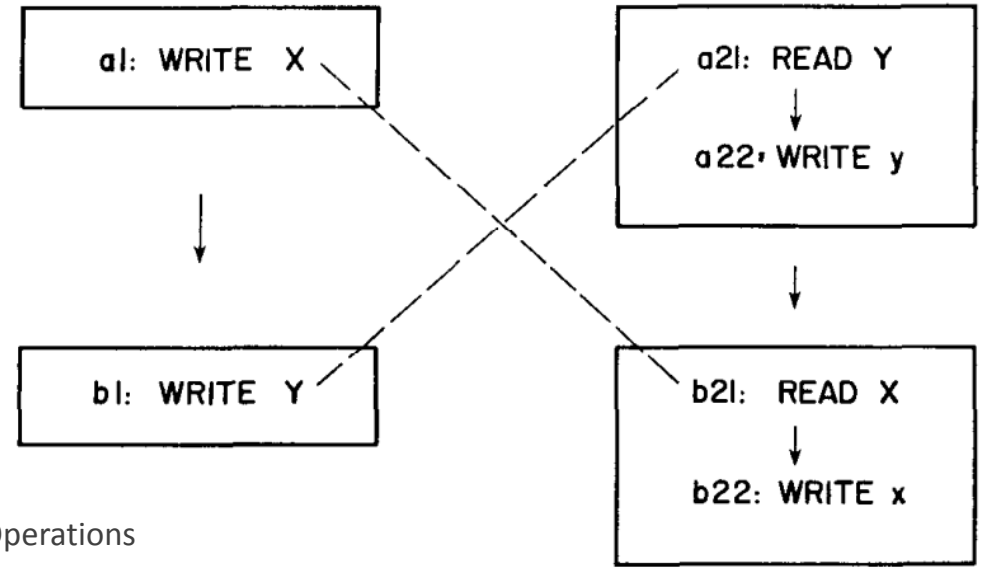
a1: X := 1;

b1: Y := 1;

Segment 2

a2: y := Y;

b2: x := X;



Full arrows – Program Order

Broken lines – Conflict Edges

Boxes - Operations

Directed Broken Lines – Execution Order

Example - Cont.

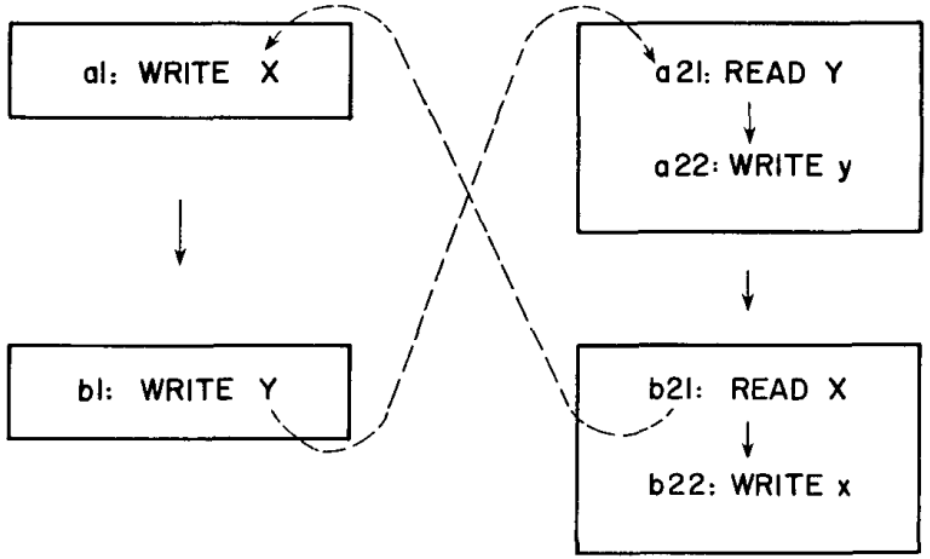
Segment 1	*Segment 2*
a1: X := 1;	a2: y := Y;
b1: Y := 1;	b2: x := X;

X, Y are shared variables (initially X=Y=0).

x, y are registers.

No interleaving of operations can yield x = 0 and y = 1 and be SC, since it will imply the order of occurrences: **b2 -> a1 -> a2 -> b1**.

Example – Cont. 2



The illegal result in our terminology: an **E** that leads to x=0, y=1 is not consistent with **P** and **A**, namely since no total order of **E** and **P** can be achieved (we have a cycle in **E**∪**P**):

$$(a1, b1, a21, a22, b21, a1)$$

Full arrows – Program Order

Broken lines – Conflict Edges Order

Boxes - Operations

Directed Broken Lines – Execution

Delays

We introduce the delay relation **D**: for every two accesses u,v∈ **V**, u**D**v implies v is delayed until access u is executed.

Since for every u,v∈**V**, u**E**v also implies that u is executed before v, it is immediate that for every execution, **E** is consistent with **D**.

In other words the relation graph of **D**∪**E** is acyclic (this is called the Delay Lemma).

Delays Cont.

Delays can be used to enforce correctness of concurrent programs.

We say that **D** ***enforces correctness*** if any execution **E** that is consistent with **D** is correct.

We can always use trivial mass of delays to enforce correctness at the expanse of complete loss of concurrency (the program deteriorates to a serial execution).

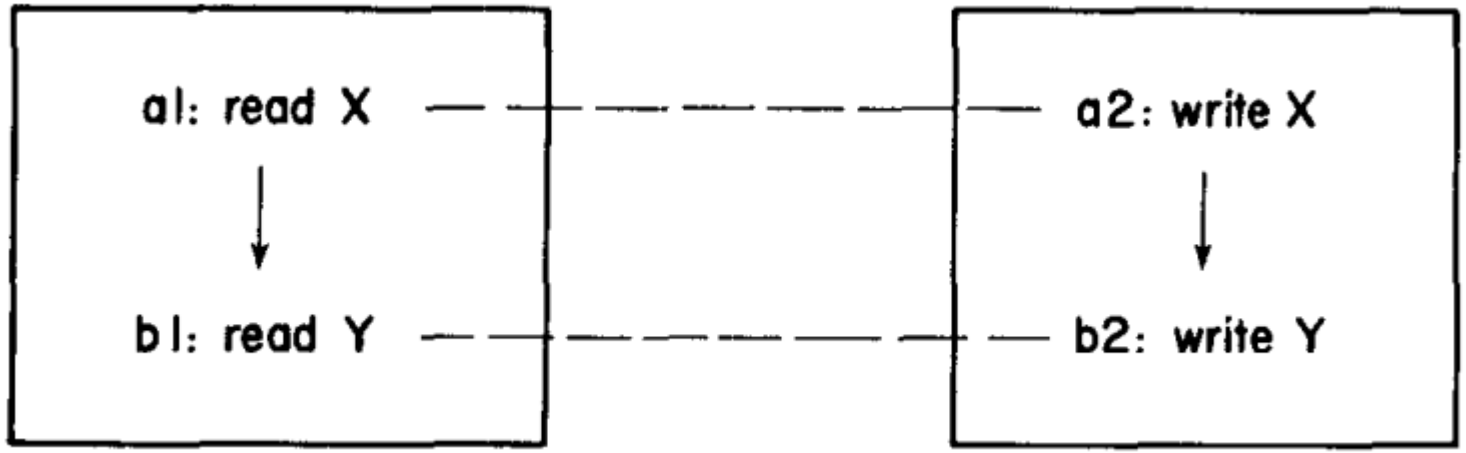
Delays Cont. 2

We will mostly be interested in delays that enforce correctness and fulfil **D**⊆**P**.

This has two advantages:

1. if **D**⊆**P** any computation order consistent with **P** is also consistent with **D**, and **D** does not prohibit computations that could occur within the limits of **P**.
2. If we allow delays outside of **P**, we then require delays between different processors operations, which might be complicated to implement.

However…



A delay **D**⊆**P** enforcing correctness does not always exists.

Even if we enforce a correct order of execution (consistent with **P**), we still might violate the atomicity constraint.

Which leads us to the next section..

Systems with no Atomic Constraints

In this section we examine code in which **A** contains singletons only.

Namely, every operation contains a single instruction, and in fact **A** is the equality relation.

Since we moved the atomicity issue out of the picture, **E** is now correct iff it is consistent with **P.**

Systems with no Atomic Constraints

In this special case, a **D**⊆**P** always exists – simply take **D** = **P**.

This however is obviously not the best solution.

We will try to find a ***minimal*** **D**⊆**P** (subset of **P**) that enforces correctness.

It turns out that under the assumption of **A** being =, such a subset can always be found.

Towards Finding a Minimal D⊆P

We note the following logic about cycles in our relations, which lead to incorrect executions:

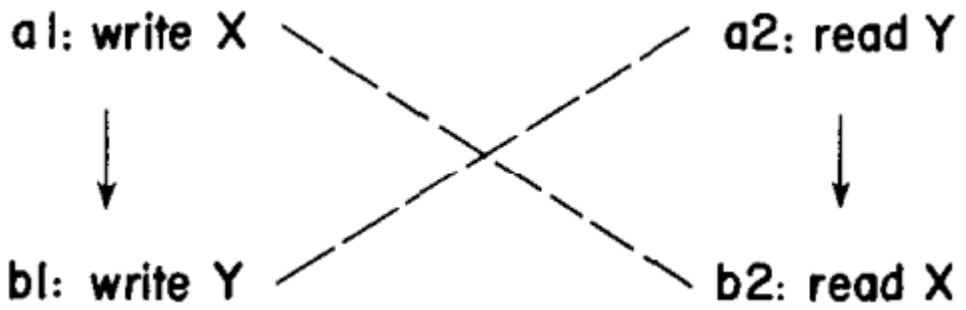
1. **E** is correct if **E** ∪ **P** has no cycles.
2. All cycles of **P** ∪ **E** are cycles of **P** ∪ **C**.
3. Suppose we impose u**D**v on every edge uv in such a cycle, whenever u**P**v.
4. Then every cycle of **P** ∪ **C** is a cycle of **D** ∪ **C**.
5. Therefore every cycle of **P** ∪ **E** is a cycle of **D** ∪ **E**.
6. However, **D** ∪ **E** is acyclic by the Delay Lemma, so we eliminated all cycles.

Such cycles will interest us when constructing the relation D⊆P.

Example

There are no cycles in **P** ∪ **C**, therefore no delays are required (can be no cycles in **P** ∪ **E**).

Example 2



P ∪ **C** has a cycle (a1, b1, a2, b2, a1), therefore we need delays. We choose (a1, b1), (a2, b2) according to our construction.

Note that this cycle implies a possible incorrect execution (a cycle in **P** ∪ **E**):

E = {(b1, a2), (b2, a1)}.

Minimal Inconsistent Executions

In order to construct **D**, it is sufficient (but not necessary) to take under consideration all cycles in **P**∪**C**. We would like to do better, and include only "necessary" cycles so **D** enforces correctness.

We define ϕ to be the family of acyclic subsets of **C** that are not consistent with **P**:

S∈ ϕ if S⊆**C** is acyclic and **P** ∪ S contains a cycle.

A minimal element of ϕ is called a ***minimal inconsistent execution*** (minimality by set containment).

Theorem 3.1

Theorem 3.1 :

Let **D** be a delay relation. Then **D** enforces correctness iff, for every minimal inconsistent execution **S**, **D** ∪ **S** has a cycle.

Theorem 3.1 – Partial Proof

$\Leftarrow$:

Suppose that for every **S**, **D** $\cup$ **S** has a cycle. Now assume **D** does not enforce correctness. Then there exists an **E** which is inconsistent. **E** contains a minimal inconsistent execution S' (acyclic subset of **C**, that together with edges from **P**, closes a cycle), and so by assumption **D**$\cup$ S' has a cycle. But then **E** $\cup$ **D** must have a cycle, which contradicts the Delay Lemma.

$\Longrightarrow$: A bit messy, but not very interesting, see paper for proof.

Critical Pairs

Definitions:

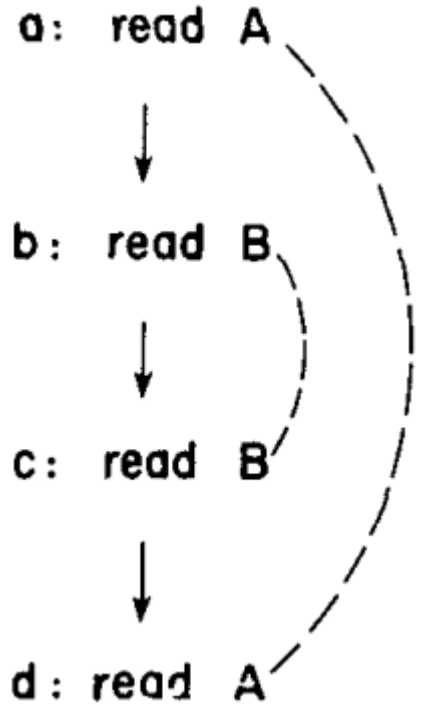
1. A set σ is a ***critical cycle*** of (**P, C**) if it is a simple cycle of **P** ∪ **C** and has no chords in **P.**
2. An edge uv ∈ **P** is a ***critical pair*** (of (**P**, **C**)) if it occurs in a critical cycle.

Example 1

Critical cycles and pairs in a single processor code.

(a, d, a) and (b, c, b) are critical cycles.

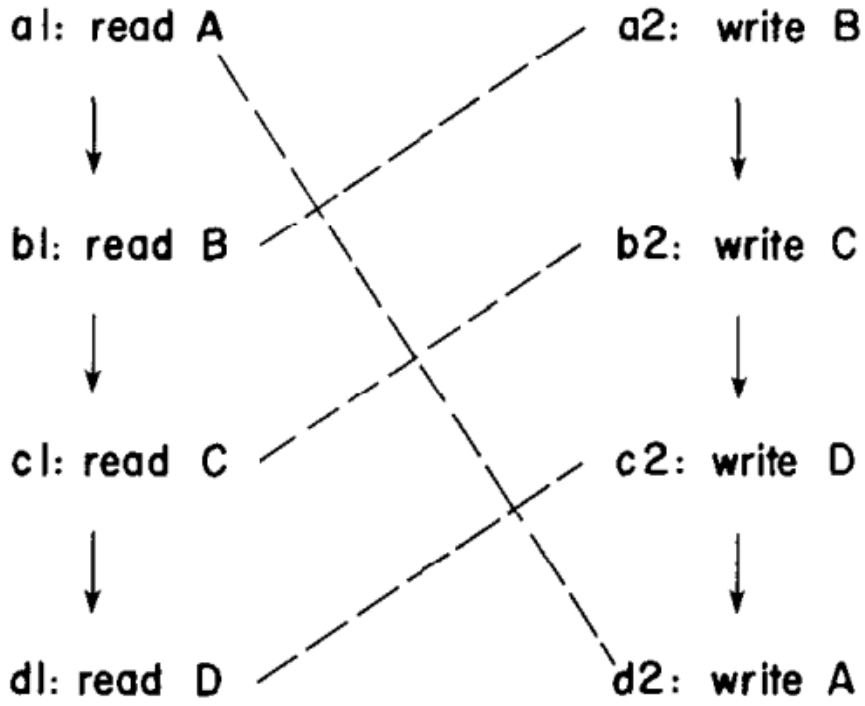
ad and **bc** are critical pairs.



Example 2

We have 4 simple cycle in **P** ∪ **C**:

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



Note that (i) is not a critical cycle (**P** is ***transitive***!)

Critical Pairs: {(a1,b1), (a1,c1), (a1, d1), (a2, d2), (b2, d2), (c2, d2)}.

Lemma 3.3

Let **S** be a minimal inconsistent execution (minimal in number of edges in **C**), and σ be a shortest cycle in **P** ∪ **S** (fewest number of edges). Then σ is a critical cycle (simple cycle in **P** ∪ **C**).

Straightforward proof from the paper (we simply show σ can have no **P** chords, and is **simple**):

> 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.

Conclusion (corollary 3.4)

Let **D** be a delay relation consisting of all the critical pairs. If **E** is an execution order that is consistent with **D**, then **E** is consistent with **P**.

Proof:

According to the previous Lemma, each minimal inconsistent execution is contained inside a critical cycle. Making **D** consists of all critical pairs, together with Theorem 3.1, ensures correctness (make sure all the MIEs are inside some cycle of **D** ∪ **C**).

Sanity Check

So what have we achieved?

Our goal was finding the exact edges **P** of all cycles in **P** ∪ **C**, which delaying them ensures correctness.

We showed which cycles of those interest us (critical cycles, Lemma 3.3).

And we showed why including the **P** edges in those cycles in our **D** relation actually works to enforce correctness (Theorem 3.1).

No minimality shown yet…

Example 2 - Revisited

The cycles listed below, displayed in full.

Note that (i) is still not a critical cycle due to its (a1, c1) edge in **P**.

P

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

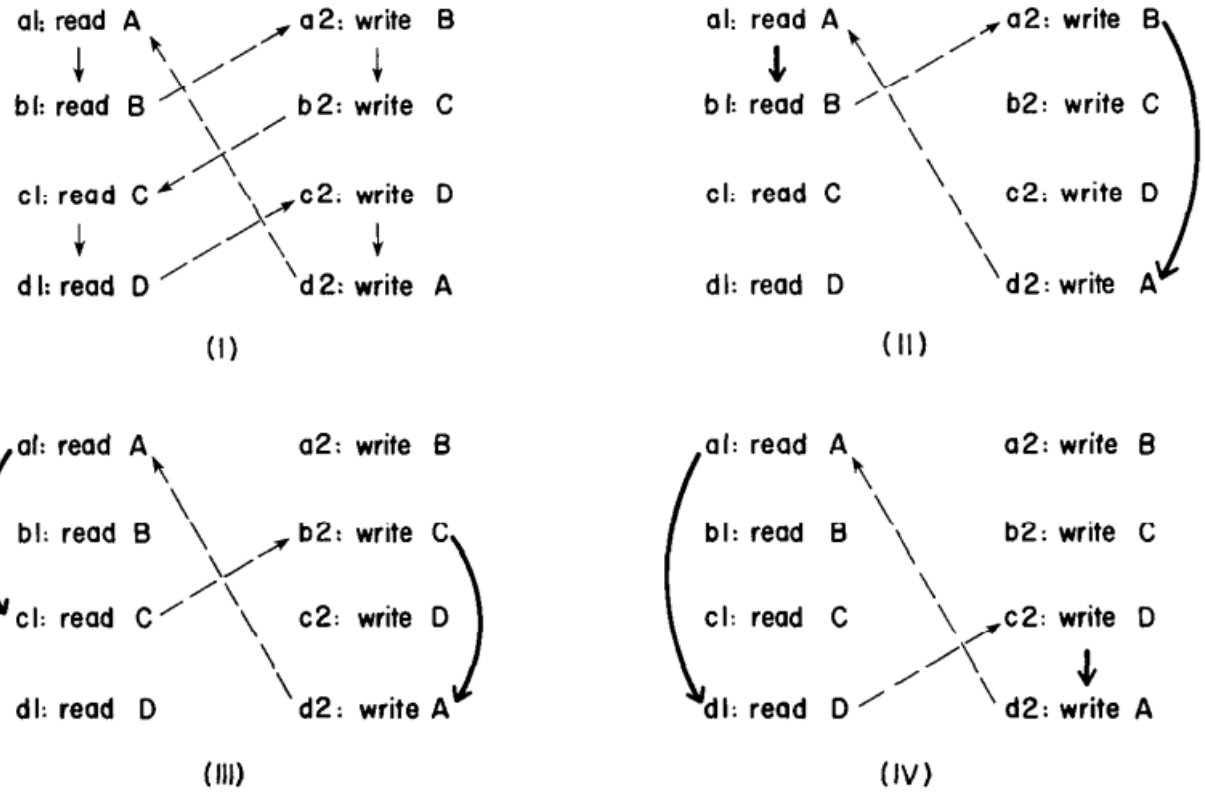
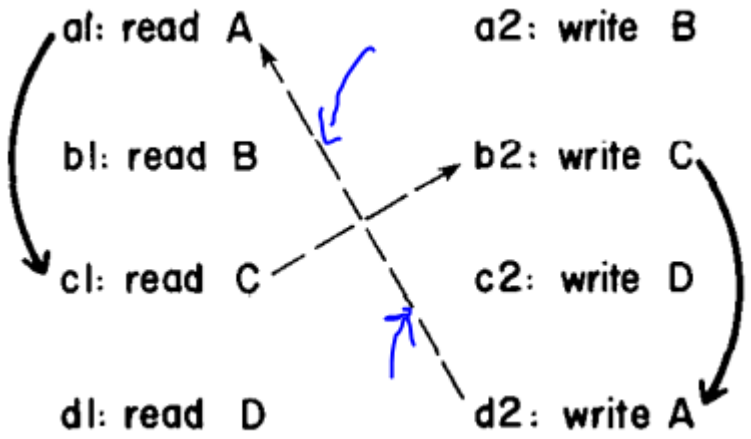


Fig. 10. Cycles and critical pairs.

A Word About Minimality

We showed that every ***minimal inconsistent execution*** is contained in a ***critical cycle***.

We now show that the converse also holds: The edges **C – P** of a ***critical cycle*** are a ***minimal inconsistent execution***.



Therefore a **D** relation containing all the critical pairs is both ***sufficient*** and ***necessary*** to enforce correctness.

Lemma 3.5

Let σ be a critical cycle in (**P**, **C**), let S = σ – **P**. Then any arbitrary simple cycle π in **P** ∪ S is obtained by replacing the **P** edges in σ by a simple path edges of **P**, and all S edges in σ are in π.

Conclusion

Let σ be a critical cycle in (**P**, **C**), let S = σ – **P**. Then S is a ***minimal inconsistent execution***, and σ is the ***unique critical cycle*** in *S* ∪ ***P***.

Proof:

By the previous lemma, S is contained in all cycles in **P** ∪ S, thus is **MIE**. If π is a different critical cycle under **P** ∪ S, then π is obtained by replacing some **P** edges paths of σ, but σ has no chords, so all its paths are of length 1, and thus $\boldsymbol{\sigma = \pi}$.

Bottom Line: By fixing **D** to be the "critical pairs" relation, we do not cover superfluous **P** edges.

Simplified Definition of Critical Cycles

We can define critical cycles in a more restricted way, without losing any critical pairs.

First, we can ignore cycles in **C** ∪ **P** that have no **P** edges (do not contribute critical pairs).

Secondly, we can require that critical cycles have no **C** chords. Under certain conditions, this we'll lose no critical pairs.

Thus we are able to ignore superfluous cycles and simplify our analysis.

Illustration

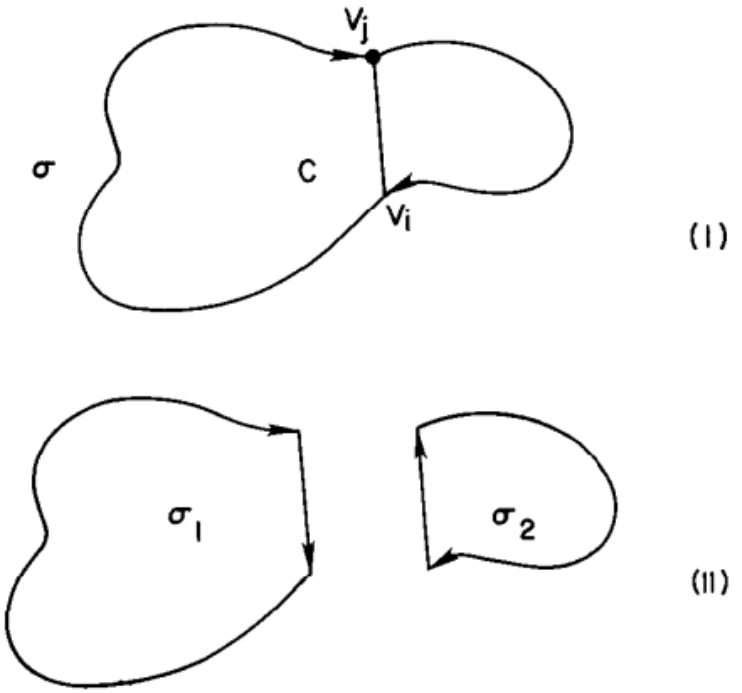
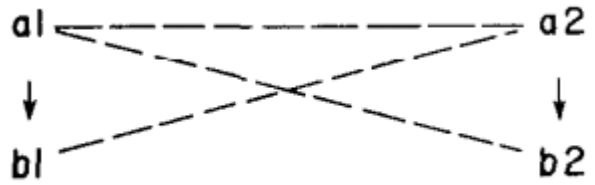


Fig. 12. Deleting C chords from critical cycles.

We lose no **P** edges (no critical pairs).

Example

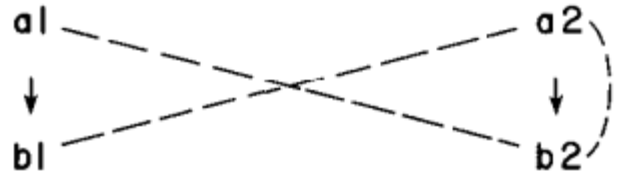


Immediate critical cycles: **(a1, b1, a2, b2, a1)**, (a1, b1, a2, a1), (a1, a2, b2, a1).

We can lose the first one, since it has a C chord (a1, a2), and still keep the same

Set of critical pairs: (a1, b1), (a2, b2).

Example – Where It Fails



Immediate critical cycles: (a1, b1, a2, b2, a1), (a2, b2, a2).

The first one has a C chord, (a2, b2), however it cannot be ignored since (a2, b2) is a trivial C-chord, that is it consists of the reversal of and edge on a cycle (in our case – (a2, b2, a2)).

Ignoring it we lose the essential critical pair (a1, b2).

A Sufficient Condition

We'll require that all accesses to variables are ordinary read and write operations.

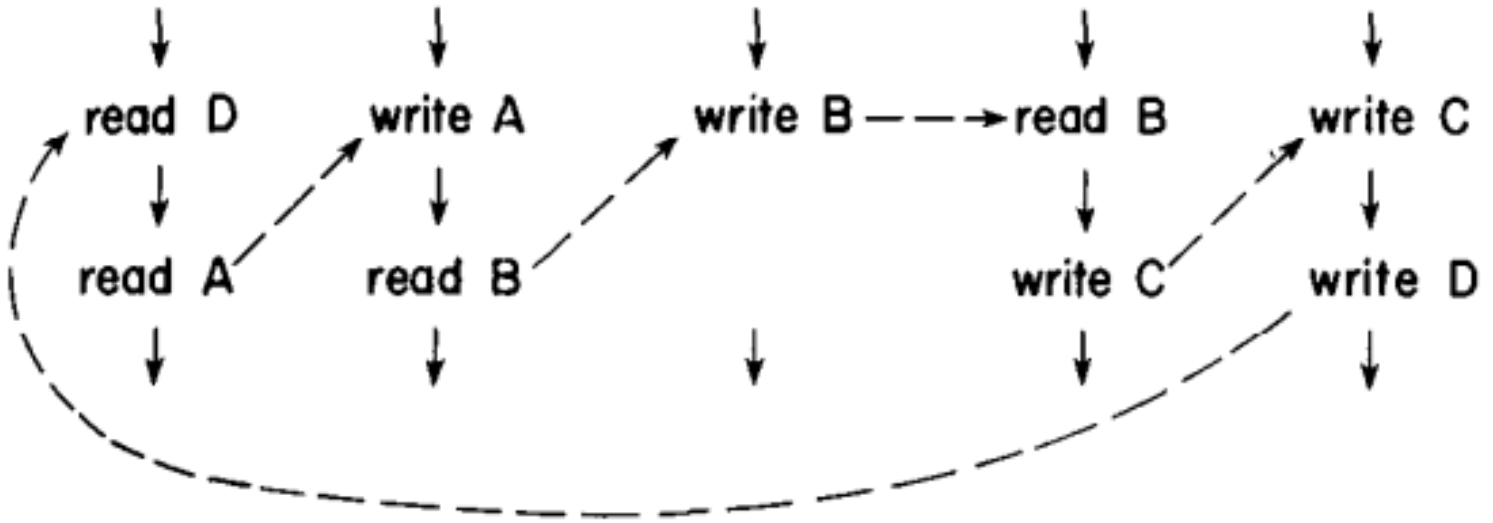
Then under this condition, the following Theorem holds:

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.*

(*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.*

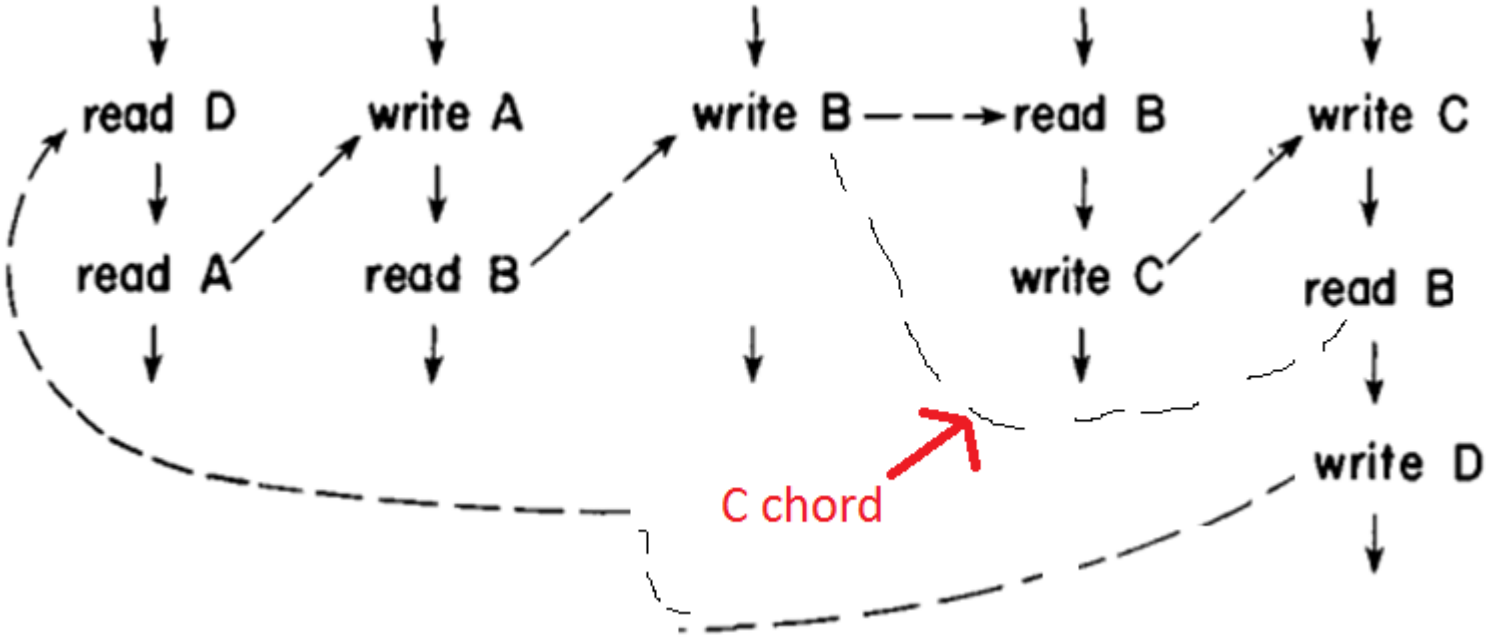
Example (positive)



No **C** chords since we maintain the requirements of the Theorem.

Simple Negative Example

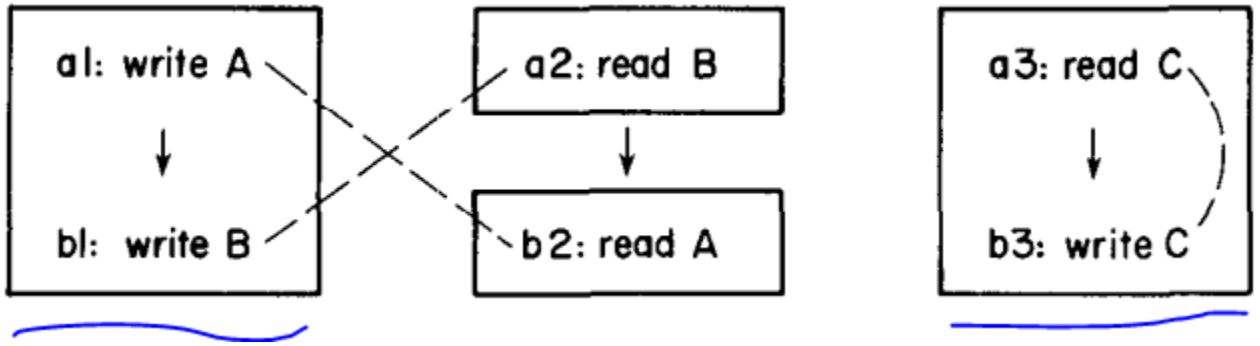
If we violate the requirement that each access to the same variable is consecutive we get a **C** chord.



Delays in General Systems

We now consider general systems <**V**, **A**, **P**, **C**>. **A** is an equivalence relation, but not the equality relation.

We again show existence of a minimal **D** that enforces correctness (**E** now needs be consistent with **P** *and* **A**). **D** is now a subset of **P** ∪ **A**.



Proofs in this section are very similar to those in the previous section, and we shall skip most of them.

E consistent with P and A – a reminder

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 $v\bar{E}w$.

In the context of our world, **E** will be consistent with **P** and **A** if it respects both the ***program order*** and the ***atomicity*** of variable accesses inside each operation.

Minimal Consistent Executions

We extend our previous definition. S $\subseteq$ **C** is now a minimal consistent execution if:

(1) S is inconsistent with **P** and **A**.

(2) S is minimal (any subset of S is consistent with **P** and **A**).

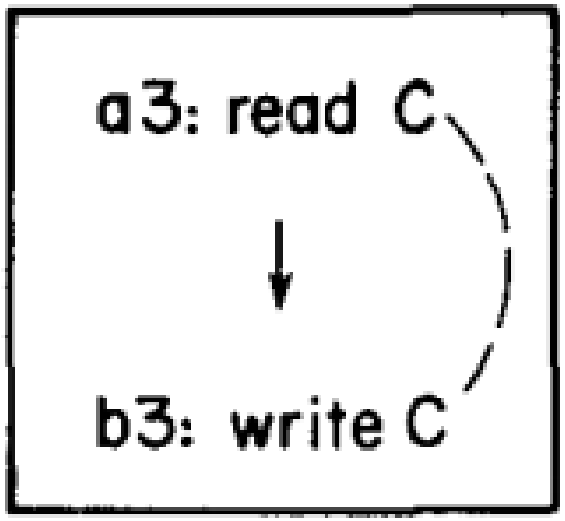
It also turns out that Theorem 3.1 still holds:

D enforces correctness iff **D** $\cup$ S has a cycle, for every minimal consistent execution S.

Possible Issues with our Program

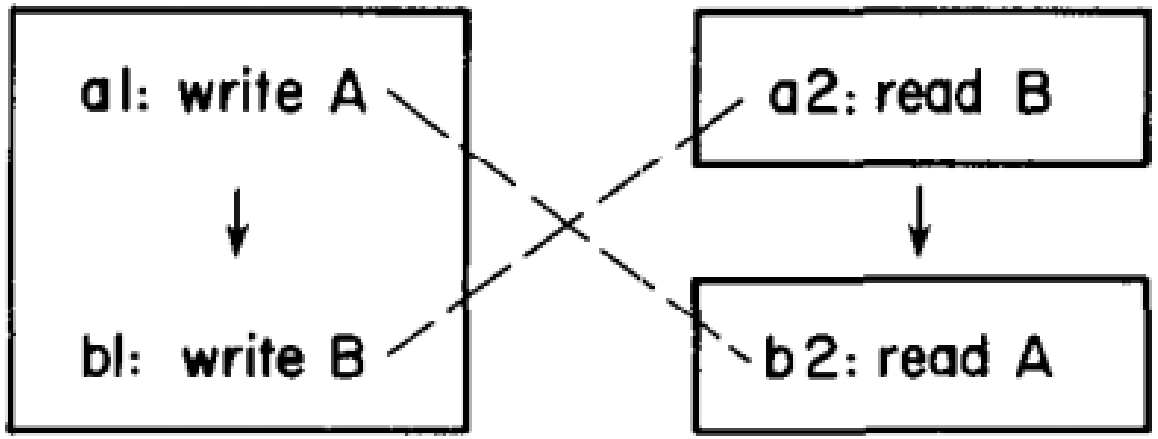
Note that now **C** edges may be present both inside atomic operations, and between them.

Wrong execution order within an atomic operation can be handled with delays in the same way as in the past.



Possible Issues with our Program

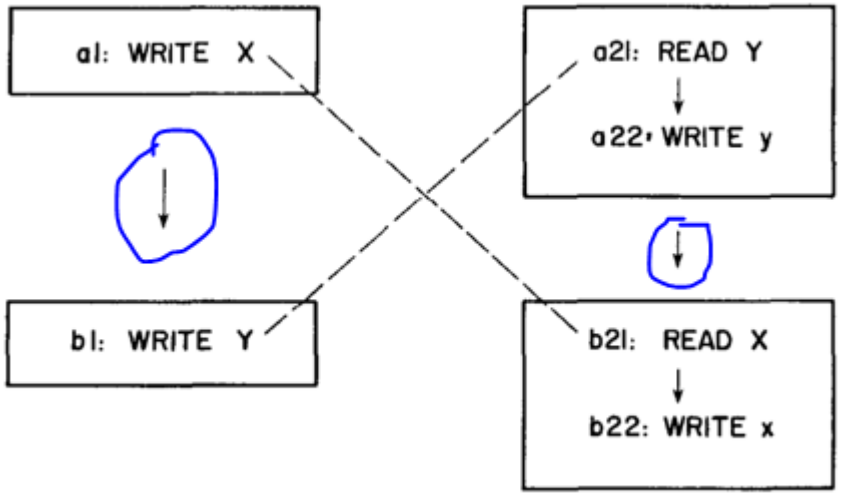
In order to handle the second type of inconsistencies (**wrong execution order across operations**), we consider how **critical cycles of operations** are related to **critical cycles of accesses**.



P/A – Definition

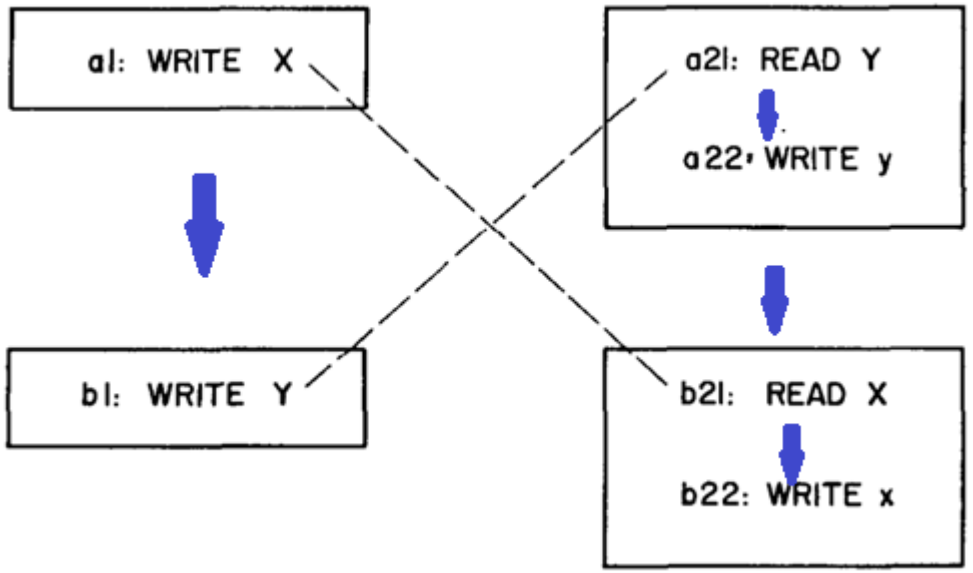
Let **P** be an irreflexive relation and **A** be an equivalence relation on the same set U. Then [u]**P/A**[v] if u≠v and there exists u, v such that u**P**v.

Visually, in our context, **P/A** edges would be **P** edges between operations:

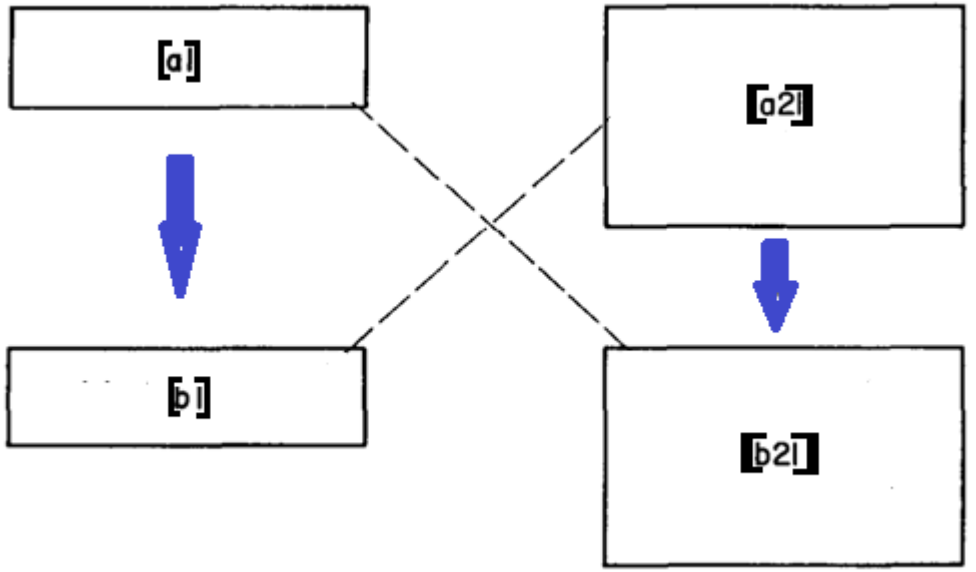


P/A – Same with a cycle

For example, a cycle under (**P/A, C/A)**:

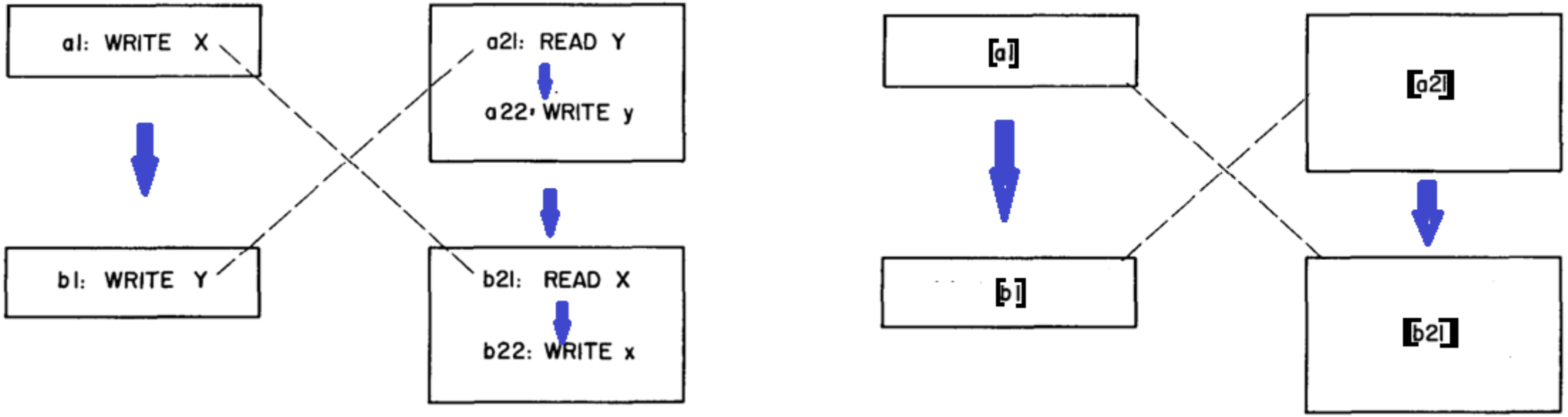


A cycle in **P**∪**C**, under relations **P**, **C**.



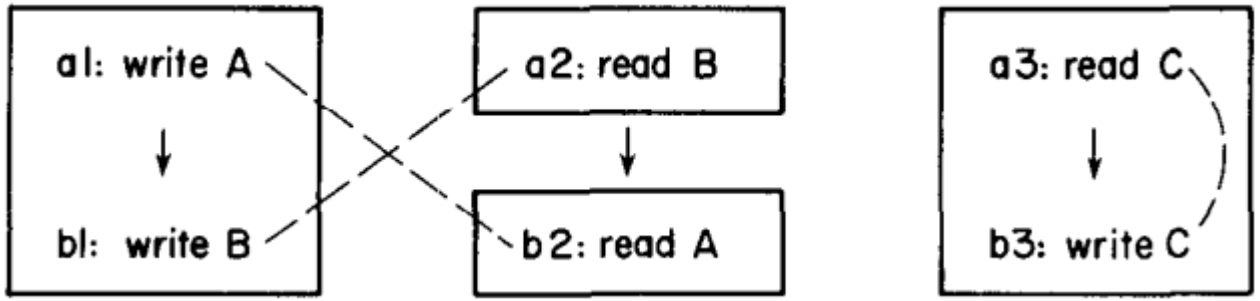
A cycle under relations **P/A**, **P/C**.

P/A – Same with a cycle



Also called the ***projection*** of the cycle under **A.**

Example



(**P** ∩ **A**, **C** ∩ **A**) has the critical cycle (a3, b3, a3).

(**P** ∪ **A**, **C** – **A**) has the cycle (a1, b1, a2, b2, a1).

Lemma 4.1

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 $(\mathbf{P}/\mathbf{A}, \mathbf{C}/\mathbf{A})$. Conversely, every critical cycle σ of $(\mathbf{P}/\mathbf{A}, \mathbf{C}/\mathbf{A})$ 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 $C - \mathbf{P} \cup \mathbf{A}$.*

What does it mean? That every ***critical cycle*** on the "big" graph (**P**∪**A**, **C – A**), which includes all operations and their sub-accesses, can be "minimized" to a ***critical cycle*** on the equivalent graph under **A**, (**P/A**, **C/A**), and vice versa.

"minimized" – has a projection

Theorem 4.2

Let $\mathrm{D_a}$ be the set of all critical pairs in (**P** ∩ **A**, **C** ∩ **A**), and D^a the set of all critical pairs in (**P** ∪ **A**, **C** – **A**). And define $\mathrm{D_0} = \mathrm{D_a} \cup \mathrm{D^a}$. Then $\mathrm{D_0}$ ∪ S contains every minimal inconsistent execution S, and $\mathrm{D_0}$ enforces correctness.

Proof Idea:

(**P** ∩ **A**, **C** ∩ **A**) takes care of all inconsistencies inside operations (constructed like in the previous section).

(**P** ∪ **A**, **C** – **A**) takes care of all inconsistencies across operations (constructed using Lemma 4.1).

Example

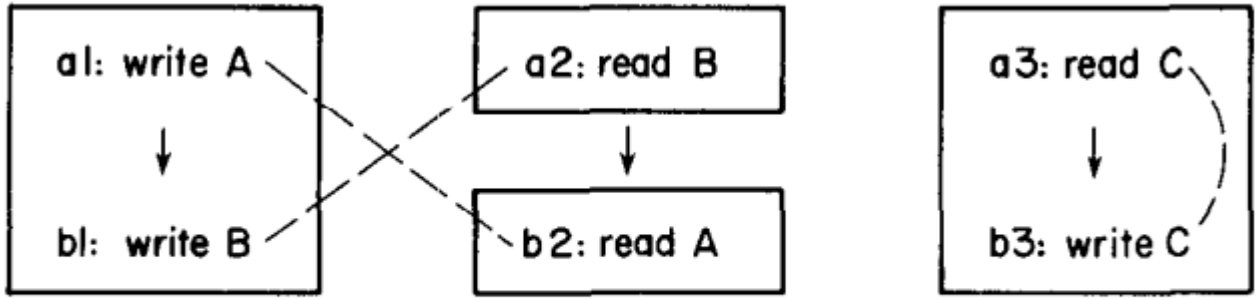


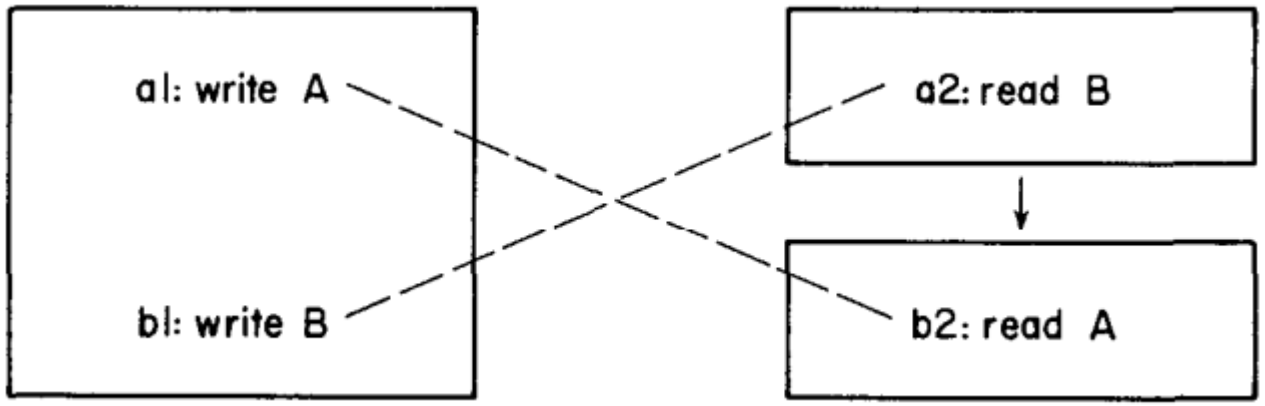
Fig. 16. Example code for nontrivial atomicity constraints.

(**P** ∩ **A**, **C** ∩ **A**) has the critical cycle (a3, b3, a3), D_a = {(a3, b3)}.

(**P** ∪ **A**, **C** – **A**) has the cycle (a1, b1, a2, b2, a1), D^a = {(a1, b1), (a2, b2)}.

D_0 = {(a3, b3), (a1, b1), (a2, b2)}. These delays assure atomicity and program order. **D**⊆ **P**.

Example

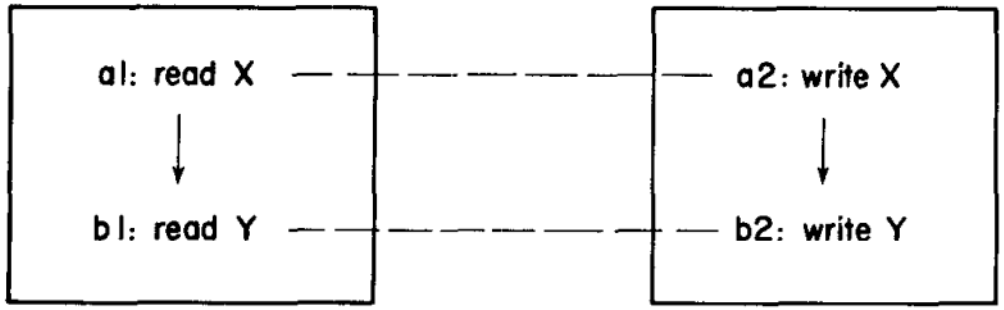


(**P** ∩ **A**, **C** ∩ **A**) has no critical cycles.

(**P** ∪ **A**, **C** – **A**) has the same cycle (a1, b1, a2, b2, a1), D^a = {(a1, b1), (a2, b2)} = D_0.

Note that there is no program order between a1 and b1, yet atomicity requires a delay between them.

Delays Have their Limit

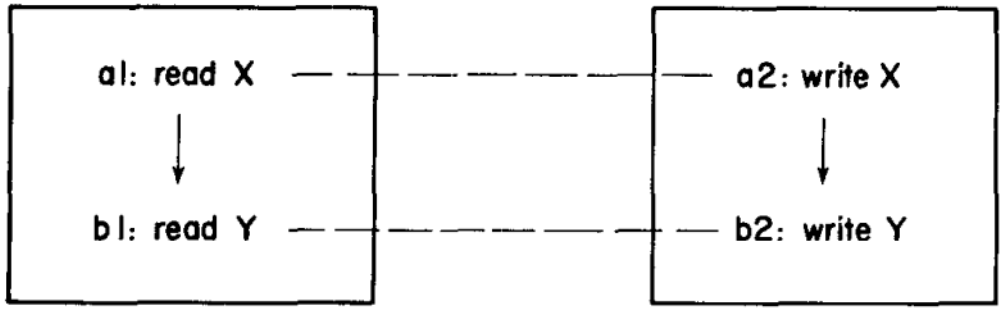


(**P** ∪ **A**, **C** – **A**) has two critical cycles: (a1, b1, b2, a2, a1), (a1, a2, b2, b1, a1).

One contradicts each other, D_0 has a cycle. This is impossible to enforce.

When D_0 has cycles delays are not sufficient to enforce correctness (locks or other mechanisms are required).

Can We Avoid the Failure?



D_0 in this case is a subset of **P** ∪ **A**, and not only **P**.

This is the only case where it can have cycles (edges exclusively in **A**).

We'll look for a sufficient condition which prevents it.

Systems where Delays Enforce Correctness

Definition: An access u is ***external*** if there exists another access v, such that:

¬u**A**v, u**C**v, but neither u**P**v nor u**P**u.

That is, an access is external iff it conflicts with an access of another operation, and there is no program order between them.

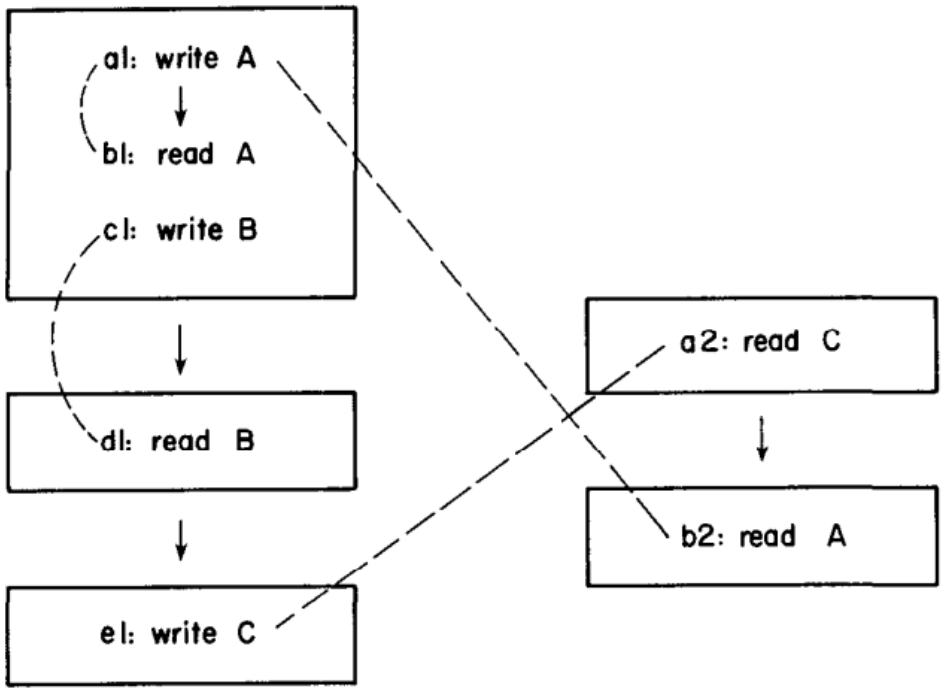
Represents a nondeterministic interaction with another operation (e.g. from a parallel segment of code).

The Sufficient Property

Definition: A has the ***single external access*** property if each operation contains at most one external access.

Code with this property can use delays to enforce correctness, since D_0 has no edges in **A**, so $\mathrm{D}_0 \subseteq \mathrm{P}$ (and acyclic).

Example



Operations “interfere non-deterministically ” with each other only “once”. No cycles in D_0 = {(a1, b1), (a1, e1), (c1, d1), a2, b2)}.

Summary + A Word about Minimality

We showed the cases in which a delay relation **D** (actually D_0) that enforces correctness exists.

It can be shown that **D** is indeed minimal.

Proof is very similar to what we saw in the previous section, so we skip it.

Large Atomic Operations & Locks

As we saw there are cases where delays do not suffice to enforce correct executions.

We add to our arsenal a locking protocol :

A partition of the accesses in the code into disjoint ***locking sets***.

The protocol protects the execution of accesses in each set.

The partition is represented by an equivalence relation **L**.

Locking sets are the equivalence classes of **L**.

Locks

The locking protocol can be obtained by using ***locks***.

We distinguish between two kinds:

A ***read*** lock – multiple of which can be set simultaneously on the same variable.

A ***write*** lock – exclusive of any other lock.

The Locking Protocol

Let **u** be a locking set.

A lock by **u** is set on all variables accessed by **u** prior to the actual access' execution.

A ***read lock*** is secured for each read access. A ***write lock*** is secured only for update accesses.

If we cannot secure locks for all the accesses in **u**, we release those we secured, and try again (so no deadlock).

Locks and Delays

We can combine the use of delays with locks.

Let **D** and **L** be a delay relation and a locking relation respectively.

If **uD\Lv**, then all locks on behalf of **v** are secured after locks on behalf of **u** are released.

We assume **D** and **D\L** are acyclic.

Delays and Locking Lemma

Using our locking protocol, every execution **E** is consistent with **D** and **L.**

Our Formal Problem Revisited

As before, every program is represented by the tuple $\langle \mathbf{V, A, P, C} \rangle$. $\mathbf{E}$ is correct if it is consistent with $\mathbf{P}$ and $\mathbf{A}$.

We can trivially enforce such consistency, by using a locking set for every atomic set, and delaying every two accesses related by program order: $(\mathbf{D, L}) = (\mathbf{P, A})$.

We'll try to do better.

Our Formal Problem Revisited

Following the spirit of the previous sections, we look for $(\mathbf{D}, \mathbf{L})$ such that $\mathbf{L} \sqsubseteq \mathbf{A}$, (i) $\mathbf{D} \subseteq \mathbf{P}$, or (ii) $\mathbf{D} \sqsubseteq \mathbf{P} \cup \mathbf{A}$.

In (i) we do not lose any legal executions, but might not always be feasible.

In (ii) we might lose some legal executions, but we enforce correctness.

Reminder

$D_0 = D_a \cup D^a \subseteq (\mathbf{P} \cap \mathbf{A}, \mathbf{C} \cap \mathbf{A}) \cup (\mathbf{P} \cup \mathbf{A}, \mathbf{C} - \mathbf{A})$, the set of critical pairs.

First Case

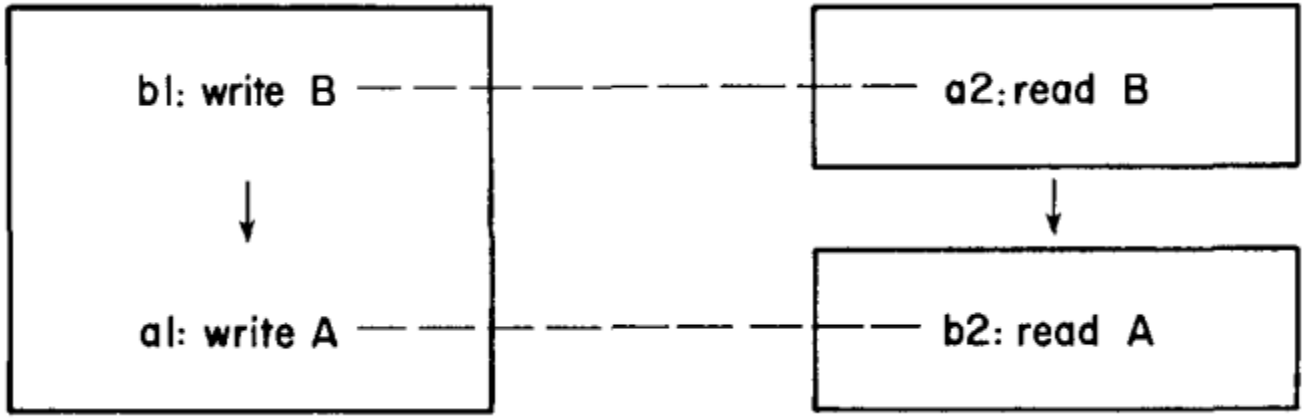
Suppose it is possible to achieve non-trivial **L** and **D** such that $\mathbf{L} \sqsubseteq \mathbf{A}$ and $\mathbf{D} \subseteq \mathbf{P}$.

Given D_0 (can always be computed regardless of correctness), we define:

(5.1) Let $\mathbf{D^L} = \mathbf{D}_0 - \mathbf{L}$,
(5.2) let $\mathbf{D_L}$ be the set of **P** edges on critical cycles of $(\mathbf{P} \cap \mathbf{L}, \mathbf{C} \cap \mathbf{L})$, and
(5.3) let $\mathbf{D(L)} = \mathbf{D^L} \cup \mathbf{D_L}$.

Then we take **L** to be the **symmetric, transitive closure** of D_0 **- P**.

First Case - Example



$D_0 = ((a1, b1), (a2, b2))$.

The locking sets are, $L = (D_0 - P)^+ = \{\{a1, b1\}, \{a2\}, \{b2\}\}$.

And D(L) is (a2, b2).

Second Case

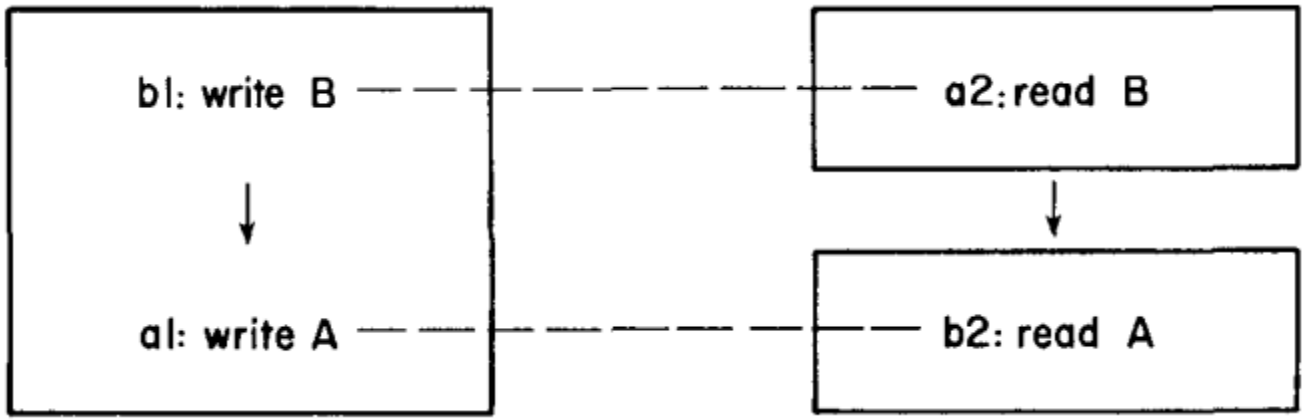
Suppose it is possible to achieve non-trivial **L** and **D** such that $\mathbf{L} \sqsubseteq \mathbf{A}$ and $\mathbf{D} \subseteq \mathbf{P} \cup A$.

Given D_0 (can always be computed regardless of correctness), we define:

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

And we take **L** to be the strongly connected components of D_0.

Second Case - Example



$D_0 = ((a1, b1), (a2, b2))$.

The locking sets are, $L = \{\{a1\}, \{a2\}, \{a3\}, \{a4\}\}$. ($D_0$ **is acyclic**)

And $D(L) = \{(a1, b1), (a2, b2)\}$. Note that we lose a legal execution: a2, b2, b1, a1.

The Mechanism of Delays

How do we decide which delays will actual take part in the code?

Useful to notice: For a relation D that enforces correctness, if u**D**v, v**D**w and u**D**w, it is not necessary to explicitly enforce the delay uDw, since it is implicit in the first two.

Using this logic, we conclude that the ***transitive reduction*** of D is enough: the smallest R relation with the property that R⊆**D**⊆ R^+.

R consists of all pairs uv∈**D** such that the longest path from u to v in the graph of **D** is of length 1.

The Mechanism of Delays

How do we actually impose delay in the hardware?

In some machines, the hardware itself is capable of delaying some of its “atomic” operations. In this case we can directly enforce the delays necessary by the R relation.

In others cases we use “fences”.

Fences

A ***fence*** instruction delays the execution of the next memory access until all preceding accesses have been executed.

We wish, of course, to minimize the number of fences.

Fences divide the code to S_0, S_1, .. S_i such that all operations in a given processor in S_i are executed before all operations in S_j, for i<j. Say level(u) is the longest path in D to u. Then set S_i to be the set of all nodes in level i.

Fences - Example

$u_i \in \mathbf{D}$

Level(0) = $\{u_0, u_2, u_4\}$. Level(1) = $\{u_1, u_3, u_5\}$.

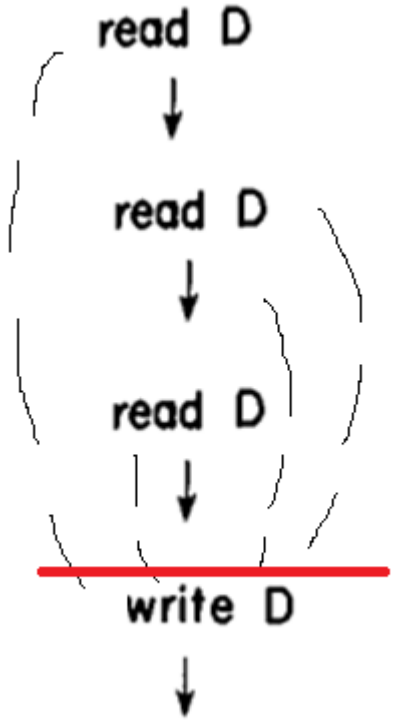
$u_0 \longrightarrow u_1 \qquad u_2 \longrightarrow u_3 \qquad u_4 \longrightarrow u_5$

Level(0) = $\{u_0, u_3\}$, level(1) = $\{u_1, u_4\}$, level(2) = $\{u_2, u_5\}$.

$u_0 \longrightarrow u_1 \longrightarrow u_2 \qquad u_3 \longrightarrow u_4 \longrightarrow u_5$

And so on…

Only 1 fence, multiple Delays- Example



Introduction

Preliminaries

Systems With No Atomic Constraints

Delays in General Systems

Large Atomic Operations & Locks

From Abstract Code to Real Programs

Summary

Summary

We presented a method of enforcing efficient and sequentially consisted execution of concurrent process on a shared-memory multiprocessor.

Note that we haven’t seen how to find data dependencies (conflicts), this problem can be hard, and the methods of discovering them are out of the scope of this paper.

Similarly, we did not present a way of finding the various minimal cycles.

THE END

Thank You