Global State, Vector Time and Dynamic Race Detection

Yongzhou Chen

Introduction:

Global State:

In an asynchronous distributed system, as no process has a consistent and immediate view of the all the process states. The following traditional control tasks of operating system are difficult:

I. mutual exclusion
II. deadlock detection
III. concurrency control

What's more, there are more new problems:
I. distributed agreement
II. distributed termination detection
III. symmetry breaking
IV. election problem.

Race Detection in multithreaded program

A race condition occurs when a program's execution contains two accesses to the same memory location that are not ordered by the happens-before relation, where at least one of the accesses is a write.

Static race detector:

Perform a compile-time analysis of the program's source code.

Advantage: Globally, and be able to warn about any data race that might occur in an execution of the program.

Disadvantage: Conservative, and always result in excessive false alarms.

Dynamic race detector:

Detect a particular execution of the program.

Advantage: Detect only the apparent data races that actually occurred during real executions.
Disadvantage: Only one specific execution path.

Global State and Vector Time

Event

Definition:

an atomic transition of the local state which happens in no time
common model:

- *internal event*: only cause a change of state
- *send event*: send a message
- *receive event*: receive a message and update local state

Causality relation / Happen-before relation:

e < e’ hold if any of the following condition holds:

- Program order: e and e' are events in same process and e precedes e'
- Send/Recv: e send message and e' receive
- Locking: The two operations acquire or release the same lock.
- Fork-join: One operation is fork(t, u) or join(t, u) and the other operation is by thread u.
- Transitive: exist e'' such that e < e'' and e'' < e’

Time diagram and Poset-diagram

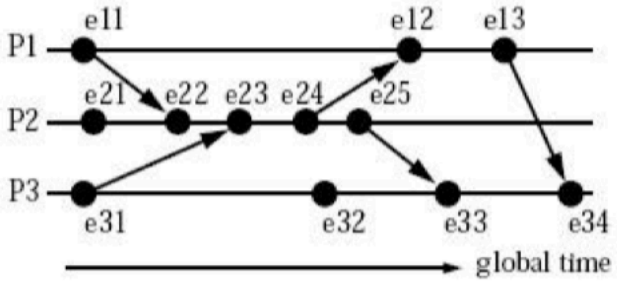


Fig. 1. An event diagram

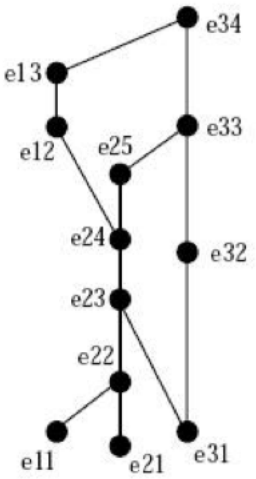


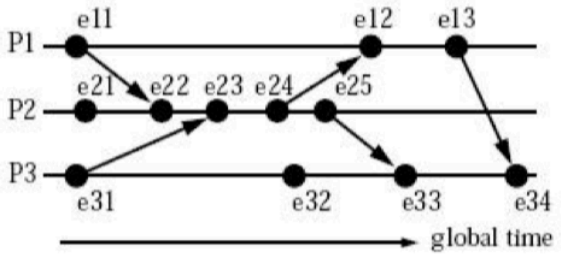
Fig. 2. The poset diagram

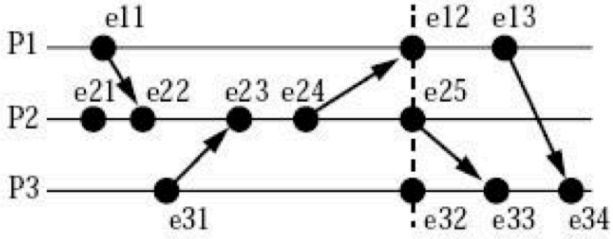
Fig1: Actual computation relationship with horizontal direction represents real time.

Fig2: logical relationship of events(causal structure of computation)

Equivalence of Time diagram

We can stretch or compress the horizontal process line of a time-diagram.
These two time-diagram are isomorphic(同构).





Cut and Consistent Cut

Graphically, a cut is a zigzag line cutting a time-diagram into two parts.

Def1: A cut C of an event set E is a finite subset such that $e \in C \ \& \ e' <_l e \Rightarrow e' \in C$

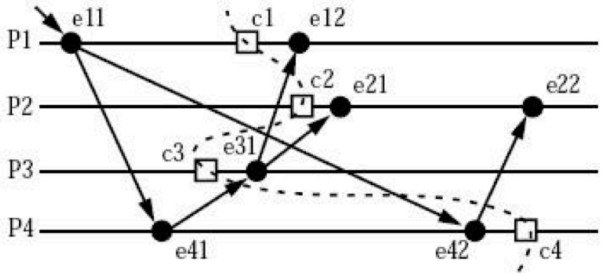


Fig. 4.
A cut

Def2: A consistent cut C of an event set E is a finite subset such that
$e \in C \ \& \ e' < e \Rightarrow e' \in C$

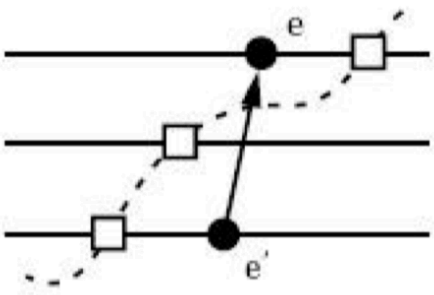


Fig. 7. An in-
consistent cut

There is an important theorem : For any time diagram with a consistent cut consisting of cut-events $c_1, \ldots c_n$, there is an equivalent time diagram where $c_1, \ldots c_n$ occur simultaneously (where the cut line forms a straight vertical line)

Virtual time and Vector time

Friedemann come up with the notion “Vector Time”. He argue that a linearly ordered structure of time is not always adequate for distributed systems and that a partially ordered system of vectors forming a lattice(格) structure is a natural representation of time in a distributed system.

Vector Time: each process P_i with a clock C_i consisting of a vector of length n, where n is the total number of processes. With each event the local clock “ticks”. Process P_i ticks by incrementing its own component of its clock:

$$C_i[i] = C_i[i] + 1$$

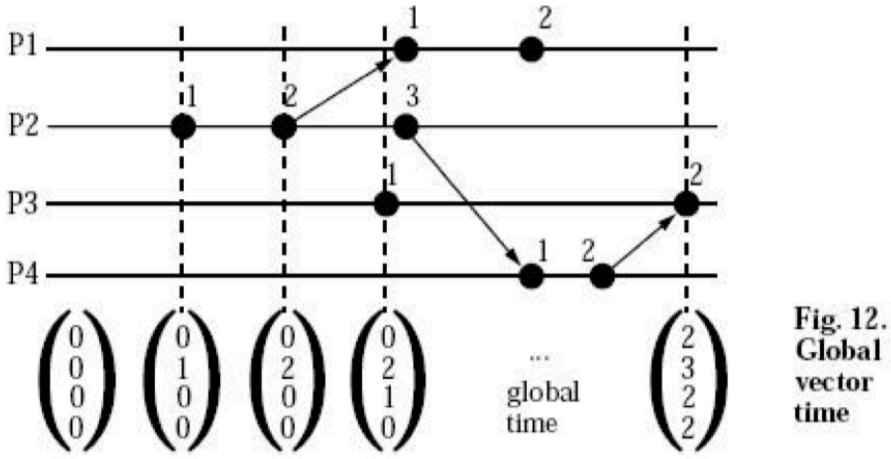


Fig. 12. Global vector time

Let *X* be a cut and c_i denote the cut event of process P_i(or the maximal event of P_i belonging to *X*). Then $t_X = sup(C_1, \ldots C_n)$ is called the global time of cut *X*.

Time Propagation

By receiving a timestamped message a process learn about the other processes' global time approximation. The receiver combines its own knowledge about global time (C_i) and the approximation it receives (t) simply by

$$C_i = sup(C_i, t)$$

At any instant of real time $\forall i, j : C_i[i] \geqslant C_j[i]$

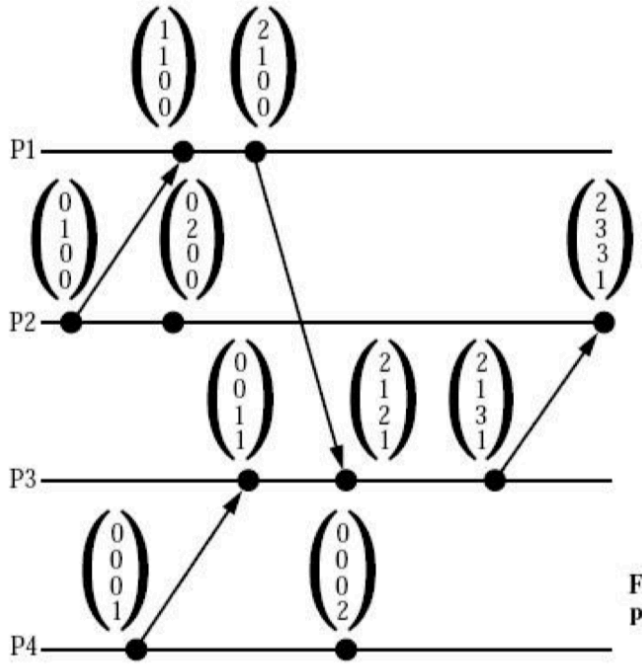


Fig. 13. Time propagation

Virtual time and Real time

A cut X is consistent if and only if $t_X = (C_1[1], C_2[2], \ldots C_n[n])$

If we take all the cut simultaneously, the cut is always consistent(as we state before), so we can get the global time by each $C_i[i]$.

Application

- *distributed debugging*
- *performance analysis*
- *data race detection*

DJIT+ Algorithm

Preliminary

Memory and Synchronization Model

The set of **operations** that a thread t can perform:

- $rd(t, x)$ and $wr(t, x)$, which read and write a value from x;
- $acq(t, m)$ and $rel(t, m)$, which acquire and release a lock m;
- $fork(t, u)$, which forks a new thread u;
- $join(t, u)$, which blocks until thread u terminates.

Here, thread identifier t ∈ Tid, variables x ∈ Var and locks m ∈ Lock.

Data race

CONCURRENT:
Two events A and B are synchronized if either $A < B$ or $B < A$. If A and B are not synchronized, A and B are concurrent events.

DATA RACE:
If two concurrent operations both access (read or write) the same variable, and at least one of the operations is a write. We say there exists an apparent data race.

The Detection Protocol

The execution of each thread is logically split into a sequence of *time frames*.
Each thread t maintain a vector of time frames, denoted $C_t[\,.\,]$
The $C_t[u]$ store the last time frame of thread u, known to thread t.

Upon initialization :

1. Each initializing thread t fills its vector of time frames with ones— $\forall i, C_t[i] \leftarrow 1$
2. The access history of each shared location v is filled with zeros (since no thread has accessed it yet) $\forall i : R_v[i] \leftarrow 0, W_v[i] \leftarrow 0$
3. The vector of each synchronization object S is filled with zeros— $L_S[i] \leftarrow 0$

Upon a release of synchronization object S :

1. The issuing thread t starts a new time frame. Therefore, it increments the entry corresponding to t in t's vector— $L_S[t] \leftarrow L_S[t] + 1.$

2. Each entry in S's vector is updated to hold the maximum between the current value and that of t's vector— $\forall i : L_S[i] \leftarrow max(C_t[i], L_S[i])$

Upon an acquire of synchronization object S :

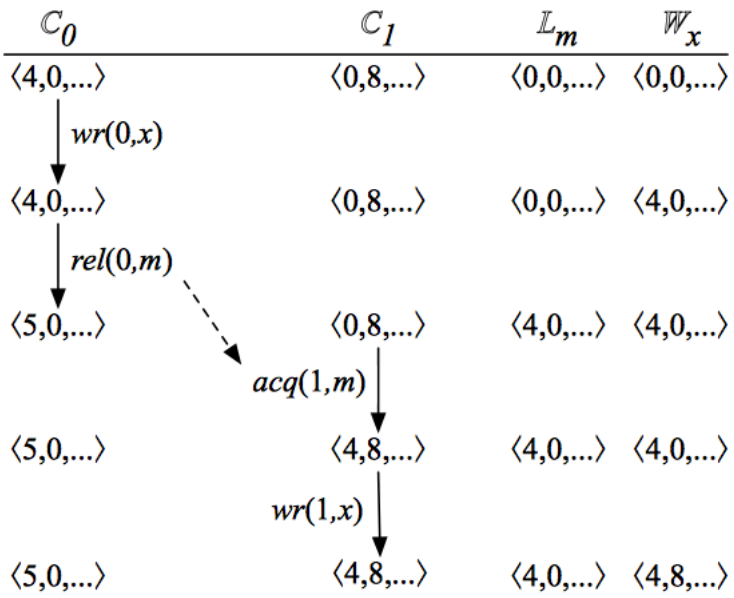
1. The issuing thread t updates each entry in its vector to hold the maximum between its current value and that of S's vector— $\forall i : C_t[i] \leftarrow max(C_t[i], L_S[i])$

Upon a first access to a shared location v in a time frame or a first write to v in a time frame :

1. The issuing thread t updates the relevant entry in the history of v. If the access is a read, it performs $R_v[t] \leftarrow C_t[t]$. Otherwise, it performs $W_v[t] \leftarrow C_t[t]$

2. If the access is a read, thread t checks whether there exists another thread u which also wrote to v, such that $W_v[u] \geqslant C_t[u]$. In other words, t checks whether it knows only about a release that preceded the write in u, and if so reports a data race.

3. If the access is a write, thread t checks whether there exists another thread u, such that $W_v[u] \geqslant C_t[u]$ or $R_v[u] \geqslant C_t[u]$



FASTTRACK Algorithm

Motivates

A limitation of VC-based race detectors such as DJIT + is their performance. If a target program has n threads, then each vector clock requires O(n) storage space and each vector clock operation (copying, comparing, joining, etc) requires O(n) time.

Preliminary

Here we formalize more notations:
partially-ordered operator: $V_1 \sqsubseteq V_2 \ iff \ \forall t, V_1(t) \leqslant V_2(t)$
join operator: $V_1 \sqcup V_2 = \lambda t \,.\, max(V_1(t), V_2(t))$

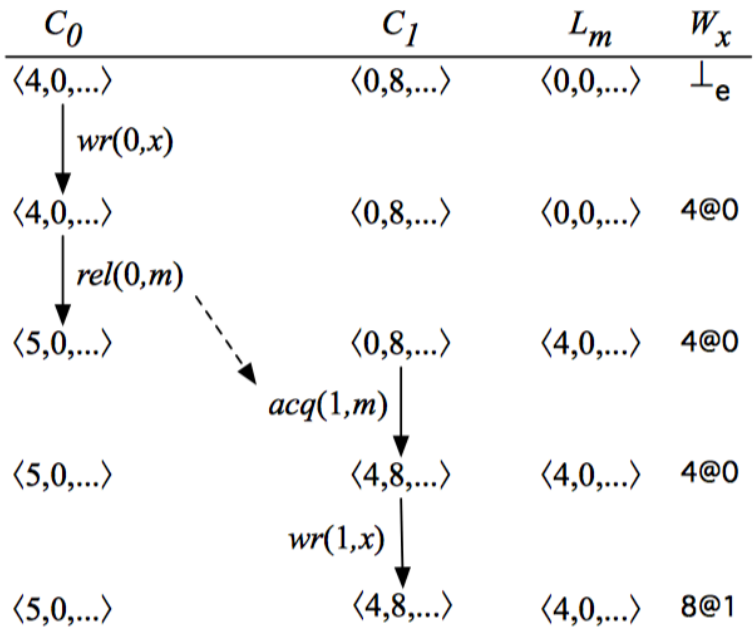
Principle

Write-Write Race

In DJIT+ algorithm, we have to confirm that $W_v[i] \leqslant C_i$ and $R_v[i] \leqslant C_i$ to make sure that written of v by P_i won't lead to data race. However, It is not necessary to record the entire vector clock $W_v[\,.\,]$ from the first write to v. Assuming no races have been detected on v so far, then all writes to v are totally ordered by the happens-before relation, and so the only critical information that needs to be recorded is the clock and identity of the thread performing the last write.

We define a pair of clock c and a thread t as an epoch, denoted as $c@t$.
An epoch $c@t$ happen before a vector clock V ($c@t \preceq V$) if and only if $c \leq V(t)$.

Using this optimized representation, FASTTRACK analyzes the above trace using a compact instrumentation state that records only a write epoch W_x for variable x, instead of the entire vector clock.



Write-Read Race

Detecting the read-write race is nearly the same as Write-Write. We only need to make the *O(1)-time* comparison $W_x \preceq C_t$ now.

Read-Write Race

Unlike write operations, which are totally ordered (assuming no race conditions detected so far), reads are not totally ordered even in race-free programs. Thus, a write to a

variable x could potentially conflict with the last read of x performed by any other thread, not just the last read in the entire trace seen so far. Hence, we need to record an entire vector clock $R_x[\ .\ ]$ for variable *x.*

However, there are usually two kind of *ordered read* in practice:
Thread-local data: only one thread accesses a variable, and hence these accesses are totally ordered by program-order.

Lock-protected data: a protecting lock is held on each access to a variable, and hence all access are totally ordered, either by program order or synchronization order.

In the less common case where reads are not totally ordered, FASTTRACK stores the entire vector clock. Since such data is typically read-shared, writes to that data are rare, so the overhead is negligible.

RoadRunner

See code directory.