

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:(happen-before关系)

$e < e'$ hold if any of the following condition holds:

- Program order(程序顺序约束): e and e' are events in same process and e precedes e'
- Send/Recv(发送接收约束): e send message and e' receive
- Locking(锁的约束): The two operations acquire or release the same lock.
- Fork-Join(生成/合并线程的约束): One operation is $\text{fork}(t, u)$ or $\text{join}(t, u)$ and the other operation is by thread u .
- Transitive(happen-before的传递性): exist e'' such that $e < e''$ and $e'' < e'$

Time diagram and Poset-diagram(时间简图和偏序图)

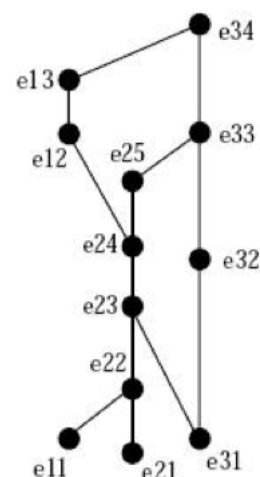
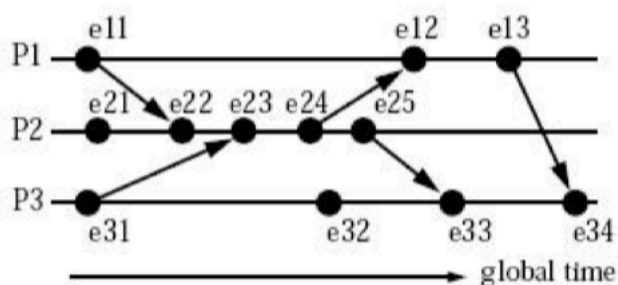


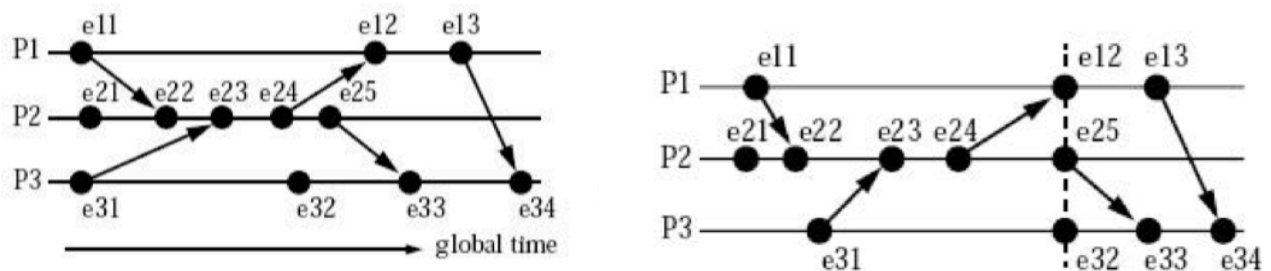
Fig1: Actual computation relationship with horizontal direction represents real time.

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

Equivalence of Time diagram(时间简图的等价性)

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 $e \in C \ \& \ e' <_I e \Rightarrow e' \in C$

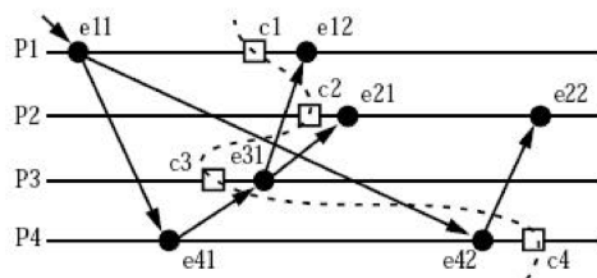


Fig. 4.
A cut

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

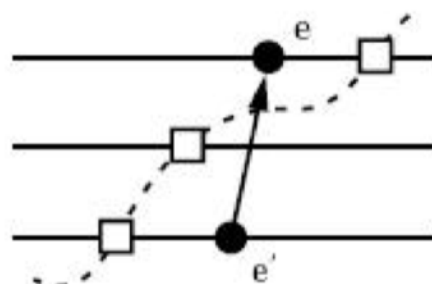


Fig. 7. An in-
consistent cut

There is an important theorem : For any time diagram with a consistent cut consisting of cut-events c_1, \dots, c_n , there is an equivalent time diagram where c_1, \dots, 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:

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

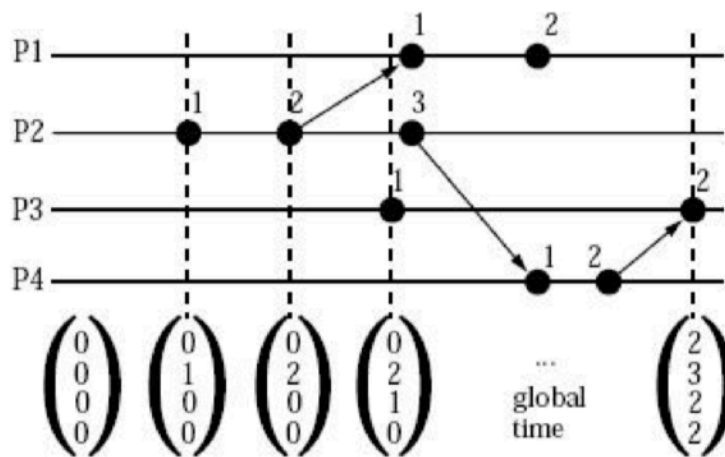


Fig. 12.
Global
vector
time

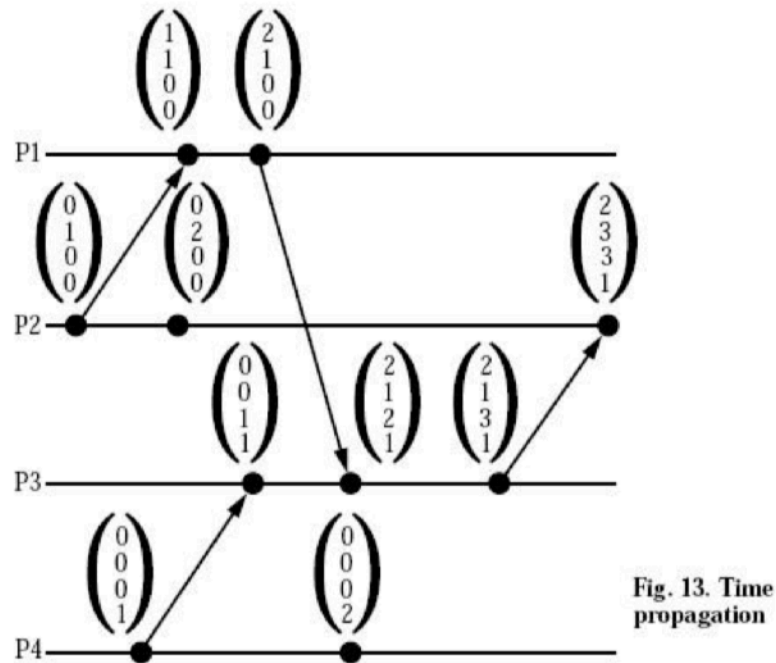
Let X be a cut and c_i denote the cut event of process P_i (or the maximal event of P_i belonging to X). Then $t_X = \sup(C_1, \dots, C_n)$ is called the global time of cut X .

Time Propagation(向量时间的传播)

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 $\forall i, j : C_i[i] \geq C_j[i]$



Virtual time and Real time(向量时间与现实时间的联系)

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

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 $t \in \text{Tid}$, variables $x \in \text{Var}$ and locks $m \in \text{Lock}$.

Vector Clocks:

- $C_i[.]$: the vector clock on thread i . $C_i[i]$ is thread i 's native time, $C_i[j](j \neq i)$ is the native time of thread j known by thread i .
- $R_v[.]$: $R_v[i]$ is the native time of thread i when i read variable v most recently.
- $W_v[.]$: $W_v[i]$ is the native time of thread i when i write variable v most recently.
- $L_S[.]$: $L_S[i]$ is the native time of thread i when i release lock S most recently.

Data race

CONCURRENT(并发):

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(数据冲突):

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.

Analysis Details

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 :

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

Upon a release of synchronization object S : (释放同步对象(锁))

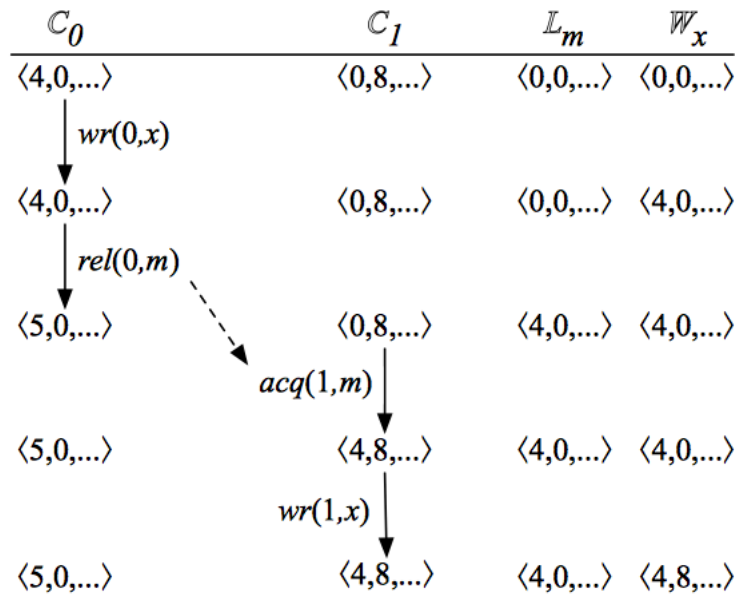
- The issuing thread t starts a new time frame. Therefore, it increments the entry corresponding to t in t 's vector— $C_t[t] \leftarrow C_t[t] + 1$.
- Each entry in S 's vector is updated to hold the maximum between the current value and that of t 's vector— $\forall i : L_S[i] \leftarrow \max(C_t[i], L_S[i])$

Upon an acquire of synchronization object S : (获取同步对象(锁))

- 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 an access to variable v :

- 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 $W_v[t] \leftarrow C_t[t]$
- If the access is a read, thread t checks whether there exists another thread u which also wrote to v , such that $W_v[u] \geq C_t[u]$. 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.
- If the access is a write, thread t checks whether there exists another thread u , such that $W_v[u] \geq C_t[u]$ or $R_v[u] \geq C_t[u]$



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 $\forall t, V_1(t) \leq V_2(t)$

join operator: $V_1 \sqcup V_2 = \lambda t. \max(V_1(t), V_2(t))$

init operator: $\perp_v = \lambda t. 0$

increment operator: $\lambda u . \text{if } u = t \text{ then } V(u) + 1 \text{ else } V(u)$

Here, $\lambda t . f(t)$ means for each t , $V[t] = f(t)$

Principle

Write-Write Race

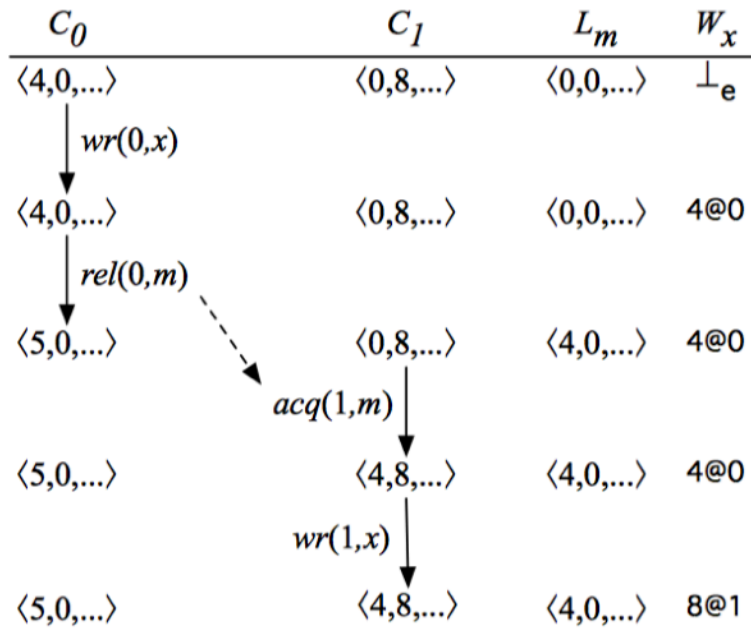
In DJIT+ algorithm, we have to confirm that $W_v \subseteq C_i$ and $R_v \subseteq C_i$ to make sure that written of v by P_i won't lead to data race. However, It is not necessary to record the entire vector clock $W_v[\cdot]$ from the first write to v . Assuming no races have been detected on v so far, 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.

(如果当前没有检测到冲突，那么之前所有的写入操作都是有序的，所以我们检验最后一次写入即可)

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 ($c @ t \leq V$) 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 write-read race is nearly the same as Write-Write. We only need to make the $O(1)$ -time comparison $W_x \leq C_t$ 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 .

(和写入不同的是，即使当前没有冲突，很有可能多个线程同时在读共享的变量。 R_x 如果只是一个epoch，该epoch记录的读操作不一定是现实时间中最近的一次读操作。所以，有必要记录一个向量时间。)

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.(几次读操作在同一线程上)
- *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.(几次读操作是有锁进行同步的)


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.

(如果多个线程共享读取一个变量，并且没有加锁的约束，那么写入该变量在大多数程序中是很少的)

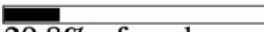
Analysis Details

Read Operation

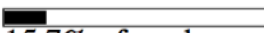
Reads:  **82.3% of all Operations**

[FT READ SAME EPOCH]  63.4% of reads


$$\frac{R_x = E(t)}{(C, L, R, W) \Rightarrow^{rd(t,x)} (C, L, R, W)}$$

[FT READ SHARED]  20.8% of reads

$$\frac{\begin{array}{l} R_x \in VC \\ W_x \preceq C_t \\ R' = R[x := R_x[t := C_t(t)]] \end{array}}{(C, L, R, W) \Rightarrow^{rd(t,x)} (C, L, R', W)}$$

[FT READ EXCLUSIVE]  15.7% of reads

$$\frac{\begin{array}{l} R_x \in Epoch \\ R_x \preceq C_t \\ W_x \preceq C_t \\ R' = R[x := E(t)] \end{array}}{(C, L, R, W) \Rightarrow^{rd(t,x)} (C, L, R', W)}$$

[FT READ SHARE]  0.1% of reads

$$\frac{\begin{array}{l} R_x = c@u \\ W_x \preceq C_t \\ V = \perp_V[t := C_t(t), u := c] \\ R' = R[x := V] \end{array}}{(C, L, R, W) \Rightarrow^{rd(t,x)} (C, L, R', W)}$$

FT Read Same Epoch:

上一次读该变量的线程正是本线程，即一个线程独占地连续读，无影响

FT Read Shared:

读取变量时， R_x 是一个向量时间，花 $O(1)$ 时间更新即可

FT Read Exclusive:

读取变量时， R_x 是一个Epoch，但本次读是有序的，所以更新 R_x 即可

FT Read Shared:


读取变量时， R_x 是一个Epoch，但本次读是无序的，因此我们将 R_x 更新为一向量时间，记录下这两次读，花费 $O(n)$ 时间。

Write Operation

Writes:  **14.5% of all Operations**


[FT WRITE SAME EPOCH]

$$\frac{W_x = E(t)}{(C, L, R, W) \Rightarrow^{wr(t,x)} (C, L, R, W)}$$

 71.0% of writes


[FT WRITE EXCLUSIVE]

$$\frac{\begin{array}{l} R_x \in Epoch \\ R_x \preceq C_t \\ W_x \preceq C_t \\ W' = W[x := E(t)] \end{array}}{(C, L, R, W) \Rightarrow^{wr(t,x)} (C, L, R, W')}$$

 28.9% of writes

[FT WRITE SHARED]

$$\frac{\begin{array}{l} R_x \in VC \\ R_x \sqsubseteq C_t \\ W_x \preceq C_t \\ W' = W[x := E(t)] \\ R' = R[x := \perp_e] \end{array}}{(C, L, R, W) \Rightarrow^{wr(t,x)} (C, L, R', W')}$$

 0.1% of writes

FT Write Same Epoch:

上一次写入该变量的线程正是本线程，即一个线程独占地连续写，无影响

FT Write Exclusive:

R_x 是一个Epoch，本次写入有序，更新 W_x 即可，花费 $O(1)$ 时间

FT Write Shared:

R_x 是一个向量时间，本次写入有序，为了提升之后的检测性能，将 R_x 复位为一个Epoch，花费 $O(n)$ 时间

Synchronization an Threading Operation

Other: **3.3% of all Operations**

[FT ACQUIRE]

$$\frac{C' = C[t := (C_t \sqcup L_m)]}{(C, L, R, W) \Rightarrow^{acq(t,m)} (C', L, R, W)}$$

[FT RELEASE]

$$\frac{\begin{array}{l} L' = L[m := C_t] \\ C' = C[t := inc_t(C_t)] \end{array}}{(C, L, R, W) \Rightarrow^{rel(t,m)} (C', L', R, W)}$$

[FT FORK]

$$\frac{C' = C[u := C_u \sqcup C_t, t := inc_t(C_t)]}{(C, L, R, W) \Rightarrow^{fork(t,u)} (C', L, R, W)}$$

[FT JOIN]

$$\frac{C' = C[t := C_t \sqcup C_u, u := inc_u(C_u)]}{(C, L, R, W) \Rightarrow^{join(t,u)} (C', L, R, W)}$$

Reference

FastTrack: Efficient and Precise Dynamic Race Detection. Cormac Flanagan, Computer Science Department University of California at Santa Cruz Santa Cruz, CA 95064; Stephen N. Freund, Computer Science Department Williams College Williamstown, MA 01267

E. Pozniansky and A. Schuster. MultiRace: Efficient on-the-fly data race detection in multithreaded C++ programs. *Concurrency and Computation: Practice and Experience*, 19(3):327–340, 2007.

F. Mattern. Virtual time and global states of distributed systems. In *Workshop on Parallel and Distributed Algorithms*, 1988.

L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), pages 558–565, July 1978.