CAUSAL MEMORY: DEFINITION, IMPLEMENTATION AND PROGRAMMING

Presented by Yoav Kaempfer

DEFINITIONS

DEFINITIONS

Memory – A finite set of locations

Operations on memory:

- Read Operation – $r_i(x)v$
- Write Operation - $w_i(x)v$

System – A finite set of processes $\mathcal{P} = \{p_1, \dots, p_n\}$ interacting via shared memory

DEFINITIONS – CONT.

Local history of a process p_i – A sequence of operations denoted L_i.

History – A collection of local histories denoted

$$H = < L_1, \dots, L_n >$$

If operation o_1 precedes operation o_2 in L_i, we write:

$$o_1 \underset{i}{\rightarrow} o_2$$

DEFINITIONS – CONT. 2

A serialization S of a set of operations A:

- A linear sequence
- Containing exactly the operations of A
- Each read operation returns the most recent value written to the location (initial value $\perp$)

S respects order $\rightarrow$ if $o_1 \rightarrow o_2$ implies o_1 precedes o_2 in S

A_{i+w}^{H} is the set of all operations in L_i and all write operations in H

DEFINITIONS – EXAMPLE

p_1	p_2
$w_1(x)0$	$w_2(x)1$
$r_1(x)0$	$r_2(x)0$

Formally:

$$\boldsymbol{L_1 = (w_1(x)0, r_1(x)0\ ),\ \ w_1(x)0 \underset{1}{\rightarrow} r_1(x)0,\ \ A^H_{1+w} = \{w_1(x)0, r_1(x)0, w_2(x)1\}}$$

$$\boldsymbol{L_2 = (w_2(x)1, r_2(x)0\ ),\ \ w_2(x)1 \underset{2}{\rightarrow} r_2(x)0,\ \ A^H_{2+w} = \{w_1(x)0, w_2(x)1, r_2(x)0\}}$$

A possible serialization of H:

$$\boldsymbol{S = w_2(x)1, w_1(x)0, r_1(x)0, r_2(x)0}$$

$$\begin{tikzcd} w_1(x)0 \arrow[d] & w_2(x)1 \arrow[d] \\ r_1(x)0 & r_2(x)0 \end{tikzcd}$$

CONSISTENT MEMORY

A memory is said to be X consistent if all histories permitted by it are X consistent

Thus, a program execution on an X consistent memory can always be described by some X consistent history H

SEQUENTIAL CONSISTENCY (SC)

Definition: There is a serialization S of the history H that respects all program orders $\underset{i}{\rightarrow}$

The processes cannot tell they are not using a single memory

Example:

$$S = r_2(x)\perp, w_2(x)1, \\ w_1(x)0, r_2(x)0, r_1(x)0$$

p_1	p_2
$w_1(x)0$	$r_2(x)\perp$
$r_1(x)0$	$w_2(x)1$
	$r_2(x)0$

Costly to implement

PIPELINED RAM (PRAM)

Definition: For each process p_i there is a serialization S_i of A^H_{i+w} that respects all program orders $\underset{j}{\rightarrow}$

Each process sees only the writes of other processes in program order

Example:

p_1	p_2
$w_1(x)0$	$w_2(x)1$
$r_1(x)1$	$r_2(x)0$
$S_1 = w_1(x)0, w_2(x)1, r_1(x)1$	$S_2 = w_2(x)1, w_1(x)0, r_2(x)0$

Hard to code

CAUSAL MEMORY

WRITE-INTO ORDER

Associates a write operation with each read operation (except reads of initial value $\perp$)

Formally, a write-into order $\mapsto$ on a history H is a relation such that:

- If $o_1 \mapsto o_2$, then there are x and v s.t. $o_1 = w(x)v$ and $o_2 = r(x)v$
- For any operation o_2, there is at most one o_1 s.t. $o_1 \mapsto o_2$
- If $o_2 = r(x)v$ and there is no o_1 s.t. $o_1 \mapsto o_2$, then $v = \perp$

A history H may have more than one write-into orders

CAUSALITY ORDER

The transitive closure of the union of all $\underset{i}{\rightarrow}$ and $\mapsto$

Alternatively, $o_1 \rightsquigarrow o_2$ if and only if at least one of the cases holds:

- $o_1 \underset{i}{\rightarrow} o_2$
- $o_1 \mapsto o_2$
- There exists o' s.t. $o_1 \rightsquigarrow o' \rightsquigarrow o_2$

If $\rightsquigarrow$ is cyclic, then it is not a causality order

A history H may have more than one causality orders

EXAMPLE

p_1	p_2
$w_1(x)1$	$w_2(x)1$
$r_1(x)1$	$r_2(y)\perp$

There are two possible write-into orders, each corresponding to a causality order:

$$\begin{tikzcd}
w_1(x)1 \arrow[d, shift right] \arrow[d, shift left] & w_2(x)1 \arrow[d] \\
r_1(x)1 & r_2(y)\perp
\end{tikzcd}$$

$$\begin{tikzcd}
w_1(x)1 \arrow[d] & w_2(x)1 \arrow[d] \arrow[dl] \\
r_1(x)1 & r_2(y)\perp
\end{tikzcd}$$

CAUSAL MEMORY (CM)

A history H is causal if there exists a causality order $\rightsquigarrow$ such that: For each process p_i, there is a serialization S_i of A_{i+w}^{H} that respects $\rightsquigarrow$

Each process sees the writes of the other processes in the same causality order

Strictly weaker than SC but strictly stronger than PRAM

CM < SC

p_1	p_2
$w_1(x)0$	$w_2(x)1$
$r_1(x)1$	$r_2(x)0$

The history is not SC, but it is CM:

Causality order:

$$\begin{tikzcd}
w_1(x)0 \arrow[d] \arrow[dr] & w_2(x)1 \arrow[d] \arrow[dl] \\
r_1(x)1 & r_2(x)0
\end{tikzcd}$$

Serializations:

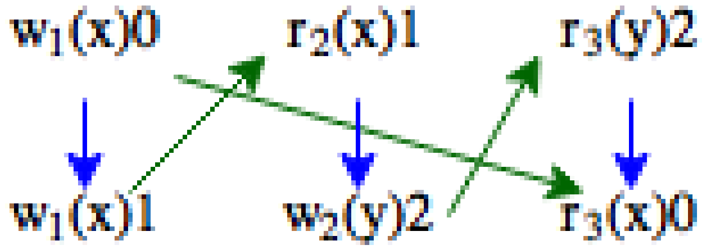
$$S_1 = w_1(x)0, w_2(x)1, r_1(x)1$$

$$S_2 = w_2(x)1, w_1(x)0, r_2(x)0$$

PRAM < CM

p_1	p_2	p_3
$w_1(x)0$	$r_2(x)1$	$r_3(y)2$
$w_1(x)1$	$w_2(y)2$	$r_3(x)0$

The history is PRAM, but it is not CM; There is only one possible causality order:



And there is no possible serialization for p_3

MORE CM EXAMPLES

p_1	p_2	p_3	p_4
$w_1(x)1$	$r_2(x)1$	$w_3(y)1$	$r_4(y)1$
	$r_2(y)\perp$		$r_4(x)\perp$

Causality order:

$$\begin{tikzcd}
w_1(x)1 \arrow[r] & r_2(x)1 \arrow[d] & w_3(y)1 \arrow[r] & r_4(y)1 \arrow[d] \\
 & r_2(y)\perp & & r_4(x)\perp
\end{tikzcd}$$

Serializations:

$S_1 = w_1(x)1, w_3(y)1$
$S_2 = w_1(x)1, r_2(x)1, r_2(y)\perp, w_3(y)1$
$S_3 = w_1(x)1, w_3(y)1$
$S_4 = w_3(y)1, r_4(y)1, r_4(x)\perp, w_1(x)1$

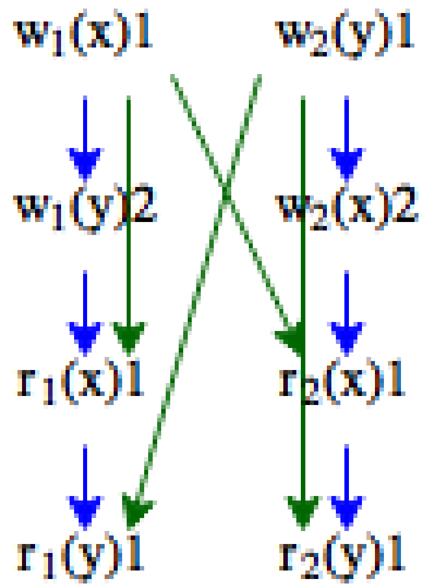
MORE CM EXAMPLES – CONT.

p_1	p_2
$w_1(x)1$	$w_2(y)1$
$w_1(y)2$	$w_2(x)2$
$r_1(x)1$	$r_2(x)1$
$r_1(y)1$	$r_2(y)1$

Serializations:

$$S_1 = w_1(x)1, w_1(y)2, r_1(x)1, w_2(y)1, r_1(y)1, w_2(x)2$$
$$S_2 = w_2(y)1, w_2(x)2, w_1(x)1, r_2(x)1, r_2(y)1, w_1(y)2$$

Causality order:



IMPLEMENTATION

DATA STRUCTURES

Each process holds:

- M – A private copy of the shared memory $\mathcal{M}$
- t – A vector clock (an integer array of size n)
- $OutQueue$ – a FIFO queue of outgoing messages
- $InQueue$ – a priority queue of incoming messages, ordered by “timestamp”

Two vector clocks (timestamps) can be compared element wise:

- If each element in t_1 is less or equal to its correspondence in t_2 then $t_1 \preccurlyeq t_2$
- If $t_1 \preccurlyeq t_2$ and $t_1 \neq t_2$ then $t_1 \prec t_2$

$\preccurlyeq$ is transitive

IMPLEMENTATION

```
/* Initialization: */
    foreach $x \in \mathcal{M}$ do
        $M[x] := \bot$
    for $j := 1$ to $n$ do
        $t[j] := 0$
    $OutQueue := \langle\rangle$
    $InQueue := \langle\rangle$

/* Read action: to read from $x$ */
    return($M[x]$)                          /* Add $r_i(x)*$ to $L_i$ and $S_i$ */

/* Write action: to write $v$ to $x$ */
    $t[i] := t[i] + 1$
    $M[x] := v$                             /* Add $w_i(x)v$ to $L_i$ and $S_i$ */
    enqueue $\langle i, x, v, t\rangle$ to $OutQueue$
```

IMPLEMENTATION – CONT.

```
/* Send action: executed infinitely often */

  if $OutQueue \neq \langle\rangle$ then
      let $A$ be some nonempty prefix of $OutQueue$
      remove $A$ from $OutQueue$
      send $A$ to all others

/* Receive action: upon receipt of $A$ from $p_j$ */

  foreach $\langle j, x, v, s\rangle \in A$
      enqueue $\langle j, x, v, s\rangle$ to $InQueue$

/* Apply action: executed infinitely often */

  if $InQueue \neq \langle\rangle$ then
      let $\langle j, x, v, s\rangle$ be head of $InQueue$
      if $s[k] \leq t[k]$ for all $k \neq j$ and $s[j] = t[j] + 1$ then
          remove $\langle j, x, v, s\rangle$ from $InQueue$
          $t[j] := s[j]$
          $M[x] := v$                                  /* Add $w_j(x)v$ to $S_i$ */
```

EXAMPLES

Positive example:

p_1	p_2
$w_1(x)0$	$w_2(x)1$
$r_1(x)1$	$r_2(x)0$

Negative example:

p_1	p_2	p_3
$w_1(x)0$	$r_2(x)1$	$r_3(y)2$
$w_1(x)1$	$w_2(y)2$	$r_3(x)0$

Reminder: Data structures are $M, t, InQueue, OutQueue$

Write-tuple is $< i, x, v, t >$

IMPLEMENTATION NOTES

Correctness is proved in the paper

Assuming that local computation is negligible with respect to message delays, d is the worst-case message delay, R is the worst-case time for a read and W is the worst-case time for a write:

- In SC: $R + W \geq d$
- In CM: $R = W = 0$

Implementation requires reliability, which can be dropped in exchange for inefficiency

PROGRAMMING

PROGRAMMING

Two classes of programs are shown, which can be written assuming SC and run correctly on CM

The above statement is proven rigorously in the paper, we will sketch a proof for a special case

This implies improved performance with little coding hassle

CONCURRENT-WRITE FREE PROGRAMS

If neither $o_1 \rightsquigarrow o_2$ nor $o_2 \rightsquigarrow o_1$ hold, o_1 and o_2 are said to be concurrent with respect to $\rightsquigarrow$

A program Π is concurrent-write free, if for all histories H of Π:

- For all causality orders $\rightsquigarrow$ of H:
  - If H has a serialization that respects $\rightsquigarrow$ (H is SC), then it has no two concurrent write operations with respect to $\rightsquigarrow$

"No two writes can occur interchangeably assuming SC"

CONCURRENT-WRITE FREE PROGRAM EXAMPLE

x, y, and z are shared variables, initially 0;
a, b, c, and d are local variables

```
process $p_1$:
    $x := 1$
    $y := 1$
```

```
process $p_2$:
    repeat
        $a := y$
    until $a = 1$
    $z := 1$
```

```
process $p_3$:
    $b := y$;
    repeat
        $c := z$
    until $c = 1$
    $d := x$
```

PROOF SKETCH

Theorem (4): If Π is concurrent-write free, then all histories of Π with causal memory are sequentially consistent

Proof sketch (finite case):

- Let H be a finite causal history of Π and $\rightsquigarrow$ its causality order
- Using structural induction, prove that H is concurrent-write free with respect to $\rightsquigarrow$ and has a serialization that respects $\rightsquigarrow$

PROOF SKETCH – CONT.

- Let H be a finite causal history of Π
- Assume towards contradiction concurrent writes w_1, w_2
- Let H' be the (proper) prefix of H excluding w_1, w_2
- H' has serialization S' because of induction assumption
- Let $\widehat{H}$ be H' where w_1 and w_2 are added to the right processes
- $S'w_1w_2$ is a serialization of $\widehat{H}$, but Π is concurrent-write free – **CONTRADICTION**

=> H is concurrent-write free with respect to $\rightsquigarrow$

PROOF SKETCH – CONT. 2

- Let H be a finite causal history of Π
- Let S_i be the serialization of A_{i+w}^{H} that respects $\rightsquigarrow$
- Let $\Longrightarrow$ be the transitive closure of $\rightsquigarrow$ union with:
  - $o_1 \Rightarrow o_2$ if o_1 is a read by p_i, o_2 is a write and o_1 precedes o_2 in S_i
- $\Rightarrow$ is acyclic, so choose o s.t. there is no o' s.t. $o \Rightarrow o'$
- $\bar{S}$ is a serialization of $H - o$ because of induction assumption
- **$\bar{S}; o$ is a serialization of H ■**

DATA-RACE FREE PROGRAMS

A program Π is data-race free, if for all histories H of Π:

- For all causality orders $\rightsquigarrow$ of H:
  - If H has a serialization that respects $\rightsquigarrow$ (H is SC), then it has no two operations that:
    - Both access the same location
    - At least one is a write
    - They are concurrent with respect to $\rightsquigarrow$

“No two operations (which are not both reads) can access the same location interchangeably assuming SC”

DATA-RACE FREE PROGRAM EXAMPLE

Data-Race Free but Not Concurrent-Write Free!

complete$[1..n]$ and *changed*$[1..n]$ are shared variables, initially 0;
done is a shared variable, initially *false*;
$x[1..n]$ are shared variables, initially 0;
$A[1..n, 1..n]$ and $b[1..n]$ are shared constants;
$t[1..n]$ are local variables, $t[i]$ local to p_i;
converged is an external routine that evaluates convergence

```
process $p_0$:
    while not $done$
        for $i := 1$ to $n$
            await($complete[i] = 1$)
        for $i := 1$ to $n$
            $complete[i] := 0$
        for $i := 1$ to $n$
            await($changed[i] = 1$)
        $done := converged(A, x, b)$
        for $i := 1$ to $n$
            $changed[i] := 0$
```

```
process $p_i$:
    while not $done$
        $t[i] := \left(b[i] - \sum_{j=1}^{i-1} A[i,j]\, x[j] -$
                $\sum_{j=i+1}^{n} A[i,j]\, x[j]\right) / A[i,i]$
        $complete[i] := 1$
        await($complete[i] = 0$)
        $x[i] := t[i]$
        $changed[i] := 1$
        await($changed[i]) = 0$
```

DATA-RACE FREE PROGRAM NEGATIVE EXAMPLE

x, y, and z are shared variables, initially 0;
a, b, c, and d are local variables

```
process $p_1$:
    $x := 1$
    $y := 1$
```

```
process $p_2$:
    repeat
        $a := y$
    until $a = 1$
    $z := 1$
```

```
process $p_3$:
    $b := y$;
    repeat
        $c := z$
    until $c = 1$
    $d := x$
```

Concurrent-Write Free But Not Data-Race Free!

CM SYNCHRONIZATION

CM can improve performance for DRF programs with synchronization, e.g. busy-wait **await** statements

Mutual exclusion cannot be realized with CM without cooperation, thus **semaphores** can be added

Although blocking, CM with synchronization primitives can be faster in practice than SC

SUMMARY

We have seen:

General consistency related definitions

The definition of Causal Memory

Two classes of programs which can be programmed for SC and run on CM

CM is faster than SC, but more easily programmable than PRAM

THE END

Thank You!