Introduction: What is a weak memory model?

Luc Maranget  
Luc.Maranget@inria.fr

The business model of washing machines

I buy a new washing machine

when the old one is broken.

The business model of computers

The old one is still working, but...

The new one runs so faster... It looks nicer too?

Happy Thursdays

Our course is on Thursdays, 16h15, in room 1014.

Exam will take place on Feb. 27 or March 5.

Weather permitting...
Avoid the washing machine business model, at any price

However, processors do not get faster anymore.

More precisely, clock speed does not increase anymore.

Performance sill increases!

Spec Benchmark results:

How long before it stabilises? Can we trust benchmarks?

And though, more and more transistors

What to do with all these transistors (and how to sell them) ?

Change your phone

New one looks nicer? But it also (often) has more cores.
More and more cores, also for high-end computers

Power 6, 2 cores per chip
Power 7, 8 cores per chip
Power 8, 12 cores per chip
Power 9, 24 cores per chip

Multiprocessors exist too

Summary of processor evolution

Current trends: integration is still increasing, performance and clock speed are stabilising, number of cores is increasing.

Programming multi-(processor/core) machines

- Expected question:
  How to program, correctly, efficiently?
  This is difficult, because of “state explosion”.

- Another, less expected question?
  How do they function?
  Or, rather, what do they do?

We shall limit ourselves to second second sub-question of second question.
What is a weak memory model?

Hardware

Another, intuitive?, view of SC

The program order is the execution order specified by the program which a thread executes. This is a given ordering of “operations” or “events”.

- The “sequential order”, or schedule results from interleaving the program orders of all threads.
- Reads from location $x$ read the value written to $x$ by the most recent write.

Or: a read event from location $x$ reads the value written to $x$ by the maximal among writes to $x$ that precede the read in the schedule.

Sequential consistency (SC, L. Lamport, 1979):
The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

Example

Schedule: (a) (b) (c) (d)

Final state: $r0=0$; $r1=1$.

A simple model for shared memory

$N$ threads (cores) write to and read from a shared memory.
Simple question on SC execution

Is final observation $r_0=0; r_1=0$ possible?

<table>
<thead>
<tr>
<th>$T_0$</th>
<th>$T_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) $x \leftarrow 1$</td>
<td>(c) $y \leftarrow 1$</td>
</tr>
<tr>
<td>(b) $r_0 \leftarrow y$</td>
<td>(d) $r_1 \leftarrow x$</td>
</tr>
</tbody>
</table>

No.

Because schedule must start either by instruction (a) or by instruction (c).

A typical concurrent program

```c
int x; // Shared variable

void *P(void *p) {
    for (int k = 0; k < 256; k++) {
        int tmp = x;
        x = tmp+1;
    }
}
```

Let us run two instances of P concurrently.

As $x$ is incremented $2 \times 256 \rightarrow 512$ times, $x$ final value is $2 \times 256 \rightarrow 512$.

**Demo:** (tst/dekker/unprotected.out)

```bash
% ./unprotected.out 256
x=512
...
x=510
```

Programmers

Experts often assume SC!

A typical concurrent program

```c
int x; // Shared variable

void *P(void *p) {
    for (int k = 0; k < 256; k++) {
        int tmp = x;
        x = tmp+1;
    }
}
```

Let us run two instances of P concurrently.

As $x$ is incremented $2 \times 256 \rightarrow 512$ times, $x$ final value is $2 \times 256 \rightarrow 512$.

**Demo:** (tst/dekker/unprotected.out)

```bash
% ./unprotected.out 256
x=512
...
x=510
```

What happened?

R and W by two threads interleave as $T_0$:R $T_1$:R $T_1$:W $T_0$:W

<table>
<thead>
<tr>
<th>$T_0$</th>
<th>$T_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>int tmp = x ;</td>
<td>int tmp = x ;</td>
</tr>
<tr>
<td>x = tmp+1 ;</td>
<td>x = tmp+1 ;</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

For instance,

```bash
... T_0:Rx(v) T_1:Rx(v) T_1:Wx(v + 1) T_0:Rx(v + 1) ...
```

Solution: RW become scheduling atoms,

```bash
... [T_0:Rx(v) T_0:Wx(v + 1)] [T_1:Rx(v + 1) T_1:Wx(v + 2)] ...
```
Mutual exclusion

Sequence "read then write plus one" must be exclusive: only one thread at a time can execute it.

Dekker’s algorithm solves the issue (for two threads).

Dekker’s locking and unlocking

Critical section: a code sequence to be executed by at most one thread at a time.

The critical section of thread whose identity is id starts by calling lock(id) and ends by calling unlock(id).

Code from a reliable source (Wikipedia)

```c
volatile int want[2], turn;

void lock(int id) {
    want[id] = 1;  // I want to enter
    while (want[1-id]) {
        /* Other also wants to enter,
        let us arbitrate,
        depending on turn */
        if (turn != id) want[id] = 0;
        while (turn != id);
        want[id] = 1;
    }
}

void unlock(int id) {
    turn = 1-id;
    want[id] = 0;
}
```

Ok, let’s go

Demo: (tst/dekker/dekker.out)

```
% ./dekker.out
x=512
x=512
x=512
x=512
x=512
x=510
```

What happened? Wikipedia cannot be wrong!
What happened?

Let us simplify Dekker’s locking code:

```c
void lock(int id) {
    want[id] = 1;  // I write 1
    while (want[1-id]) {  
        ...  // I have read 0
    }
}
```

Let us simplify even more:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_0$</td>
<td></td>
</tr>
<tr>
<td>$(a) x \leftarrow 1$</td>
<td>$(c) y \leftarrow 1$</td>
</tr>
<tr>
<td>$(b) r_0 \leftarrow y$</td>
<td>$(d) r_1 \leftarrow x$</td>
</tr>
</tbody>
</table>

Can we observe $r_0=0; r_1=0$? If so, Dekker’s locking code does not guarantee mutual exclusion.

Remember: the observation is not possible on top of SC.

The horrible truth

Modern processors perform many optimisations:
- out of order execution;
- speculative execution;
- in-core store buffers;
- cache hierarchies…

These are
- unobservable by single-thread programs;
- sometime observable by concurrent programs;

As a result, modern multiprocessors are not sequentially consistent

As a result, concurrent programming is even more difficult than you thought.

Tell me more, oh tell me more

The x86-tso model features visible (Fifo) store buffers.

Cores write into their store buffer.

Then, writes are flushed asynchronously to shared memory.
The complete truth about store buffers

Schedule: (a) (c) (b) Flush($T_1$) (d) Flush($T_0$)

\[
\begin{array}{c|c}
T_0 & T_1 \\
\hline
(a) x \leftarrow 1 & (c) y \leftarrow 1 \\
(b) r_0 \leftarrow y & (d) r_1 \leftarrow x
\end{array}
\]

\[
\begin{array}{c}
T_0 = \emptyset \\
T_1 = \emptyset \\
(a) x \leftarrow 1 \\
(c) y \leftarrow 1 \\
(b) r_0 \leftarrow y \\
(d) r_1 \leftarrow x
\end{array}
\]

Final state: $r_0=0$; $r_1=0$.

Reality check: tst/Machine/MP.litmus

X86_64 MP

```
P0             | P1             ;
MOVL $1,[x]   | MOVL [y],EAX  ;
MOVL $1,[y]   | MOVL [x],EBX  ;
exists (1:RAX=1 \land 1:RBX=0)
```

Let us run the test:

```
% litmus7 -mach corei7 MP.litmus ...
```

Test MP Allowed

Histogram (3 states)

999992:1:RAX=0; 1:RBX=0;
73    :1:RAX=0; 1:RBX=1;
999935:1:RAX=1; 1:RBX=1;
...

The non-SC behaviour is not observed.

Reality check II

Demo: test/ARMv8/MP.litmus

```
% cat MP.litmus
AArch64 MP
{ 0:X1=x; 0:X3=y; 1:X1=y; 1:X3=x; }
P0             | P1             ;
MOV W0,#1     | LDR W0,[X1]   ;
STR W0,[X1]   | LDR W2,[X3]   ;
MOV W2,#1     |               ;
STR W2,[X3]   |               ;
exists (1:X0=1 \land 1:X2=0)
```

Let us compile and upload on my phone

```
% litmus7 -mach phone -o R MP.litmus
% make -C R
/opit/android-ndk/bin/aaarch64-linux-android-gcc -Wall -O2 - pthread MP.c
...
% scp -C -P 2222 R/run.exe 128.93.84.97:MP.exe
```

Message passing test

\[
\begin{array}{c|c}
T_0 & T_1 \\
\hline
(a) x \leftarrow 1 & (c) r_0 \leftarrow y \\
(b) y \leftarrow 1 & (d) r_1 \leftarrow x
\end{array}
\]

All TSO executions:

\[
\begin{array}{c|c}
F_x & F_y \\
\hline
(c) (d) & r_0=1 r_1=1 \\
(c) F_y & r_0=0 r_1=1 \\
F_x (c) (d) F_y & r_0=0 r_1=1 \\
(c) F_x F_y (d) & r_0=0 r_1=0 \\
(c) F_x (d) F_y & r_0=0 r_1=0 \\
(c) (d) F_x F_y & r_0=0 r_1=0
\end{array}
\]

Outcome $r_0=1 r_1=0$ is forbidden.

As $T_1$ must see writes in order, $T_1$ must see flushes in order.
Run MP on my phone

```bash
% ssh -p 2222 128.93.84.97 ./MP.exe
...
AArch64 MP
{0:X1=x; 0:X3=y; 1:X1=y; 1:X3=x;}
P0 | P1;
MOV W0,#1 | LDR W0,[X1];
STR W0,[X1] | LDR W2,[X3];
MOV W2,#1 | ;
STR W2,[X3] | ;
extists (1:X0=1 \ 1:X2=0)
... Test MP Allowed
Histogram (4 states)
1770774:>1:X0=0; 1:X2=0;
3909 =>1:X0=1; 1:X2=0;
7670 :>1:X0=0; 1:X2=1;
217647:>1:X0=1; 1:X2=1;
...
Bingo.
```

Strong fence

All architectures (I know of) provide a “strong” fence, whose purpose is restoring SC.
**Demo:** tst/machine/Dekker+Fences.litmus

```bash
% cat Dekker+Fences.litmus
X86_64 Dekker+Fences
P0 | P1;
MOVL $1,[want0] | MOVL $1,[want1];
MFENCE | MFENCE;
MOVL [want1],EAX | MOVL [want0],EAX;
extists (0:EAX=0 \ 1:EAX=0)
% litmus7 -mach corei7 Dekker+Fences.litmus
...
Test Dekker+Fences Allowed
Histogram (3 states)
973587:>0:EAX=1; 1:EAX=0;
976681:>0:EAX=0; 1:EAX=1;
49732 :>0:EAX=1; 1:EAX=1;
...
**Notice:** Fences are inserted in-between memory accesses.
```

Restoring SC

Why ? Using all those clever algorithms:

![Image](https://via.placeholder.com/150)

How ? By using specific instructions.

Specific store and load instructions

ARMv8 provides store release and load acquire.
**Demo:** tst/ARMv8/MP+Rel+Acq.litmus

```bash
% cat MP+Rel+Acq.litmus
AArch64 MP+Rel+Acq
{ 0:X1=x; 0:X3=y; 1:X1=y; 1:X3=x; }
P0 | P1;
MOV W0,#1 | LDAR W0,[X1];
STR W0,[X1] | LDR W2,[X3];
MOV W2,#1 | ;
STLR W2,[X3] | ;
extists (1:X0=1 \ 1:X2=0)
...
% Test MP+Rel+Acq Allowed
922885:>1:X0=0; 1:X2=0;
27630 :>1:X0=0; 1:X2=1;
1049485:>1:X0=1; 1:X2=1;
...
**Store-Release/Load-Acquire communication restores SC.**
```
What is a weak memory model?

High-Level language

A simple optimisation

Let x and y be two shared variables of type int (with initial value 0).

```c
void P0(void) {
    x = 1;
    if (y == 1) {
        printf("%i\n",x);
    }
}

⇓

void P0(void) {
    x = 1;
    if (y == 1) {
        printf("%i\n",1);
    }
}
```

This is constant propagation, a very innocent optimisation.

Semantics and efficiency

- Programmers:
  - Want to understand the code they write.
  - Code meaning.
- Compilers (and hardware):
  - Optimise code as much as they can.
  - Must not betray.

Betraying is transforming the program so that it produces additional behaviours.

Additional behaviours are behaviours disallowed by the untransformed program.

Constant propagation is invalid (SC model)

```c
x = 1;
if (y == 1) {  
    printf("%i\n",x);
}
```  

```c
x = 1;
if (y == 1) {  
    printf("%i\n",1);
}
```

Print “0” or nothing

```c
if (x == 1) {
    x = 0;
    y = 1;  // NB: y==1 → x == 0
    printf("%i\n",x);
}
```

Print “1” or nothing
Another optimisation

Re-ordering "independant reads" does not harm (in sequential code).

Compile time:

```c
int rx = x;
int ry = y;
printf("%i, %i\n", rx, ry);
```

⇓

```c
int ry = y;
int rx = x;
printf("%i, %i\n", rx, ry);
```

Runtime:

Rxv1; Ryv2; ⇒ Ryv2; Rxv1;

However, output v1, v2 does not change.

---

Read reordering is invalid on SC

```c
int ry = y;
int rx = x;
printf("%i, %i\n", rx, ry);
```

⇓

```c
int ry = y;
int rx = x;
printf("%i, %i\n", rx, ry);
```

Additional output: 1, 0

---

Does it happen?

Let x, y and n be pointers to shared memory.

```c
int rx = 0; int ry = 0;
for (int k=0 ; k < *n ; k++) {
    rx += x[k];
    ry += *y;
}
printf("%i, %i\n", rx, ry);
```

⇓

```c
int rx = 0; int ry = 0;
int tmp = *y;
for (int k=0 ; k < *n ; k++) {
    rx += x[k];
    ry += tmp;
}
printf("%i, %i\n", rx, ry);
```

Now assume *n to be 1.

Source program performs one read of *x, followed by one read of *y.
Optimised program performs one read of *y, followed by one read of *x.
Reality check

Demo: tst/C/MP-LOOP.litmus

% cat MP-LOOP.litmus
C MP-LOOP
{
  int n=1; }

void P0(int *x, int *y, int *n) {
  int rx = 0; int ry = 0;
  for (int k=0 ; k < *n ; k++) {
    rx += x[k];
    ry += *y;
  }
}

void P1(int *x, int *y) {
  *y = 1;
  *x = 1;
}

exists 0:rx=1 \ 0:ry=0

Even worse

Let consider our loop example again, as a (library) function:

typedef struct { int r0, r1; } pair_t;

pair_t f(int *x, int *y, int n) {
pair_t p;
p.r0 = p.r1 = 0;
for (int k=0 ; k < n ; k++) {
p.r0 += x[k];
p.r1 += *y;
}
return p;
}

Again, assuming \( n \) to be one. Optimised code will read \(*y\) first and then \(*x\) once.

Bingo!

Even worse

Let \( z \) be a pointer to shared memory.

\[
pair_t p = f(z, z, 1); \quad \text{\( p.r0 \) is read first, then \( p.r1 \)}
\]

\[
\text{printf(\"%i, \%i\n\", p.r0, p.r1);} \quad \text{\( *z = 1 \)}
\]

\[
\text{\( *z = 2 \)}
\]

One expects output:

<table>
<thead>
<tr>
<th>schedule</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wz1; Wz2; Rz2; Rz2</td>
<td>2, 2</td>
</tr>
<tr>
<td>Wz1; Rz1; Wz2; Rz2</td>
<td>1, 2</td>
</tr>
<tr>
<td>Wz1; Rz1; Rz1; Wz2</td>
<td>1, 1</td>
</tr>
<tr>
<td>Rz0; Wz1; Wz2; Rz2</td>
<td>0, 2</td>
</tr>
<tr>
<td>Rz0; Wz1; Rz1; Wz2</td>
<td>0, 1</td>
</tr>
<tr>
<td>Rz0; Rz0; Wz1; Wz2</td>
<td>0, 0</td>
</tr>
</tbody>
</table>

Demo: tst/C/CoRR-LOOP.litmus
Even worse

Let $z$ be a pointer to shared memory.

\begin{verbatim}
pair_t p = f(z, z, 1); // p.r1 is read first, then p.r0 printf("%i, %i\n", p.r0, p.r1);
\end{verbatim}

One gets output:

<table>
<thead>
<tr>
<th>schedule</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wz1; Wz2; Rz2; Rz2</td>
<td>2, 2</td>
</tr>
<tr>
<td>Wz1; Rz1; Wz2; Rz2</td>
<td>2, 1</td>
</tr>
<tr>
<td>Wz1; Rz1; Rz1; Wz2</td>
<td>1, 1</td>
</tr>
<tr>
<td>Rz0; Wz1; Wz2; Rz2</td>
<td>2, 0</td>
</tr>
<tr>
<td>Rz0; Wz1; Rz1; Wz2</td>
<td>1, 0</td>
</tr>
<tr>
<td>Rz0; Rz0; Wz1; Wz2</td>
<td>0, 0</td>
</tr>
</tbody>
</table>

**Demo:** tst/C/CoRR-LOOP.litmus

Really even worse

Consider the simple CoRR program

\begin{verbatim}
int r0 = *z;
int r1 = *z;
printf("%i, %i\n", r0, r1);
\end{verbatim}

\begin{verbatim}
*z = 1; *z = 2;
\end{verbatim}

Notice that CoRR and CoRR-LOOP have the same traces.

<table>
<thead>
<tr>
<th>schedule</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wz1; Wz2; Rz2; Rz2</td>
<td>2, 2</td>
</tr>
<tr>
<td>Wz1; Rz1; Wz2; Rz2</td>
<td>1, 2 or 2, 1</td>
</tr>
<tr>
<td>Wz1; Rz1; Rz1; Wz2</td>
<td>1, 1</td>
</tr>
<tr>
<td>Rz0; Wz1; Wz2; Rz2</td>
<td>0, 2 or 2, 0</td>
</tr>
<tr>
<td>Rz0; Wz1; Rz1; Wz2</td>
<td>0, 1 or 1, 0</td>
</tr>
<tr>
<td>Rz0; Rz0; Wz1; Wz2</td>
<td>0, 0</td>
</tr>
</tbody>
</table>

Hence, considering a trace-based semantics, allowing output 2, 1 for CoRR-LOOP, means allowing it for CoRR.

Let sum it up

SC is simple, let us choose SC as our model, but:
- Machines have relaxed memory model for speed.
- Many useful compiler transformation are invalid on SC.

So having SC as a model would be inefficient.

So let us adopt a weaker model, but
- When the model is too weak...
- One cannot guarantee anything.

What to do?

- Provide programmers with “reordering” or "synchronising" constructs. With simple and precise semantics.
- As to “non-synchronised” programs
  - Either forbid them, i.e. leave their meaning undefined.
  - Or provide weak semantics.

Languages options, accepting undefined behaviours or not.
- C11/C++11, POSIX threads, ADA 83
- Java, OCAML multicore.
**Data races**

Problematic (non-SC) executions exhibit races:
- Memory accesses conflict when:
  - they are by different threads,
  - they access the same memory location,
  - at least one is a write.
- Conflicting accesses form a data race when:
  - they occur “concurrently” or “simultaneously”.

Disallowing conflicting accesses looks too drastic.
Disallowing races hence means avoiding concurrency. This looks plausible.

Define “concurrent accesses” in SC traces: adjacent accesses.

**Avoiding data races**

High level languages provide “synchronising” constructs

**Mutexes** Critical sections lock(ℓ)...unlock(ℓ) do not overlap.

**Atomic** Concurrent accesses are not racy.

Example:

```
*y = 1;
llock(ℓ);
*x = 1;
unlock(ℓ);
if (rx == 1)
    printf("%i\n", y);
```

**A racy program**

```
*y = 1;
*x = 1;
if (rx == 1)
    printf("%i\n", y);
```

A program is racy, when one of its execution is.

<table>
<thead>
<tr>
<th>schedule</th>
<th>race?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wy1; Wx1; Rx1; Ry1</td>
<td>Ok</td>
</tr>
<tr>
<td>Wy1; Rx0; Wx1;</td>
<td>Ok</td>
</tr>
<tr>
<td>Rx0; Wy1; Wx1;</td>
<td>No</td>
</tr>
</tbody>
</table>

**Important:** We quantify over SC executions.

Non-SC behaviour “print 0” is observed on the weak model (of course).

**Another well synchronised program**

```
lock(ℓ);
*y = 1;
*x = 1;
if (rx == 1)
    printf("%i\n", y);
unlock(ℓ);
```

<table>
<thead>
<tr>
<th>schedule</th>
<th>race?</th>
</tr>
</thead>
<tbody>
<tr>
<td>L(ℓ); Wy1; Wx1; U(ℓ); L(ℓ); Rx1; Ry1; U(ℓ)</td>
<td>No</td>
</tr>
<tr>
<td>L(ℓ); Rx0; U(ℓ); L(ℓ); Wy1Wx1; U(ℓ)</td>
<td>No</td>
</tr>
</tbody>
</table>
Races can be worse than being non-SC

Let \( x \) be a non-aligned pointer to some \texttt{int} in shared memory.

\[
\*x = 0x01010202; \quad \| \text{printf(“0x%x\n”,x);}
\]

**Demo:** tst/C/NoAlign.litmus

Can (and does) output:

\%

litmus7 -mach ../tst -hexa -noalign x NoAlign.litmus

... 

Test NoAlign

10000228:>1:r1=0x0;
1388 :
15 :>1:r1=0x101000;
9998369:>1:r1=0x101002;
... 

DRF Guarantee

A model (any model) provides the DRF guarantee, when:

Race-free programs have SC semantics.

So what?

- Race-free is defined by quantifying over SC execution.
- In reality programs run on weak hardware, after optimisation by compiler.

This means that DRF is a property of the system "compiler + hardware".

- Synchronising calls are opaque to the compiler: potentially modifying any location, memory operation cannot be moved past them.
- Compiler must not introduce race where there is none.
- Synchronising calls contain “sufficient fences” to prevent hardware reordering.

Semantics of programming languages

- No concurrency at all (OCaml). Well, not very fashionable.
- No shared memory (Erlang, MPI). Possible, but not a “natural” generalisation of sequential programming.
- Leave it to the hardware (Aligned C, ML-toon). Not portable.
- Complete solutions, DRF, plus
  - DRF as a definition: racy-programs can behave in any way (catch fire semantics).
  - Give semantics to racy programs.

DRF is not 100% satisfactory:

- Race-freedom is hard to verify (undecidable), even test.
- Debugging gets harder: a wrong program may result from a pure bug or from a data-race.
- Useful racy programs exist (5-1), their semantics can be complex (5-2).

Some references

On Hardware models:


On languages:

“Foundations of the C++ concurrency memory model”‘