Introduction: What is a weak memory model ?

Luc Maranget Luc.Maranget@inria.fr

Happy Mondays

Our course is on Mondays, 16h15, (in room 1004. . .).

December		January		February	
		3	A. Guatto	7	A. Guatto
6	L. Maranget	10	A. Guatto	14	Lab class
13	L. Maranget	17	L. Maranget	21	Free slot
20		24	L. Maranget		
27		31	A. Guatto		

Exam will take place, on February 28, March 7 or March 14.

Weather permitting. . .

The business model of washing machines

I buy a new washing machine

⇒

when the old one is broken.

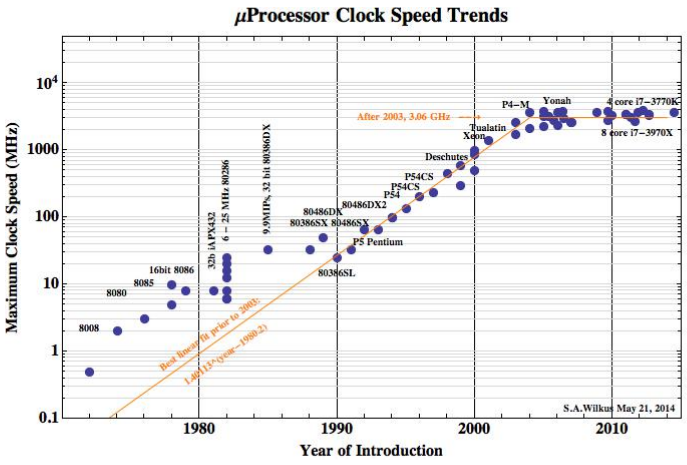
The business model of computers

The old one is still working, but. . .

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

Avoid the washing machine business model, at any price

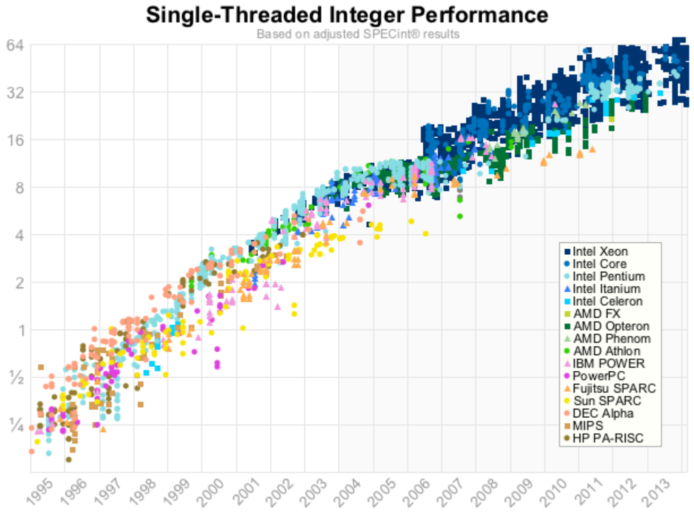
However, processors do not get faster anymore.



More precisely, clock speed does not increase anymore.

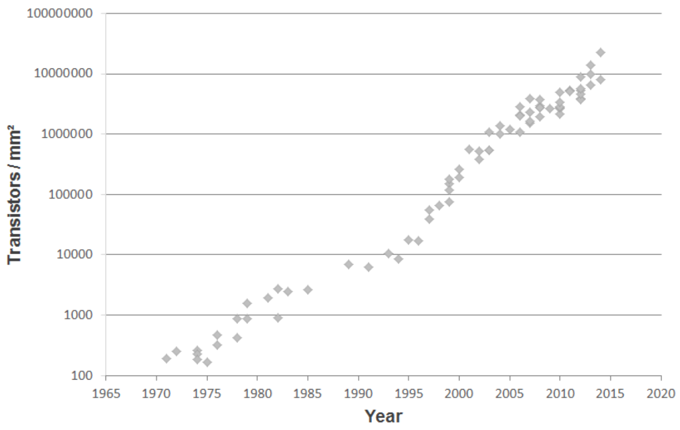
Performance sill increases!

Spec Benchmark results:



How long before it stabilises? Can we trust benchmarks?

And though, more and more transistors

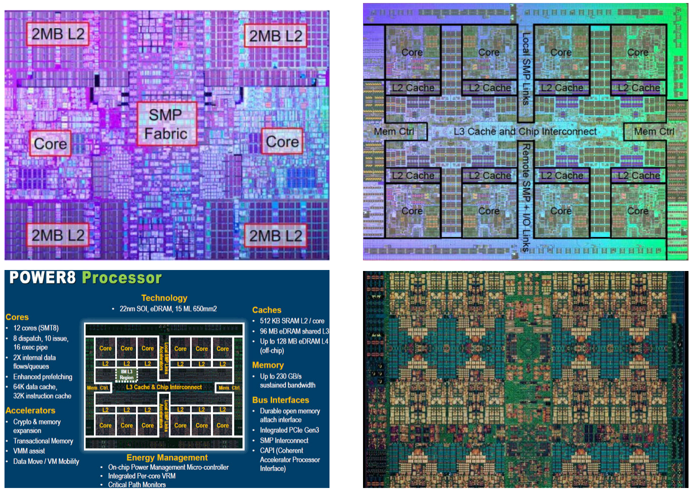


What to do with all these transistors (and how to sell them) ?

Change your phone

New one looks nicer? But it also (often) has more cores.

More and more cores, also for high-end computers



POWER8 Processor

Technology
- 22nm SOI, eDRAM, 15 ML 650mm2

Cores
- 12 cores (SMT8)
- 8 dispatch, 10 issue, 16 exec pipe
- 2X internal data flows/queues
- Enhanced prefetching
- 64K data cache, 32K instruction cache

Accelerators
- Crypto & memory expansion
- Transactional Memory
- VMM assist
- Data Move / VM Mobility

Energy Management
- On-chip Power Management Micro-controller
- Integrated Per-core VRM
- Critical Path Monitors

Caches
- 512 KB SRAM L2 / core
- 96 MB eDRAM shared L3
- Up to 128 MB eDRAM L4 (off-chip)

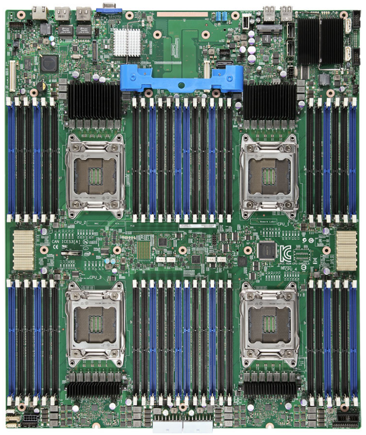
Memory
- Up to 230 GB/s sustained bandwidth

Bus Interfaces
- Durable open memory attach interface
- Integrated PCIe Gen3
- SMP Interconnect
- CAPI (Coherent Accelerator Processor Interface)

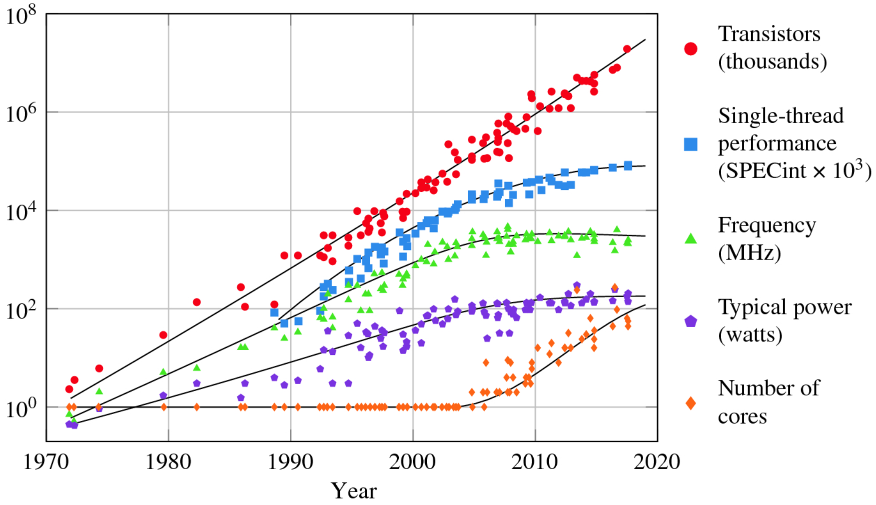
Power 6, 2 cores per chip Power 7, 8 cores per chip Power 8, 12 cores per chip Power 9, 24 cores per chip

---

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?

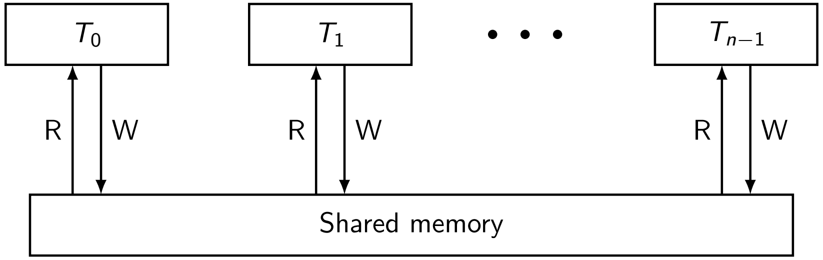
We shall limit ourselves to second second sub-question of second question.

What is a weak memory model ?

Hardware

A simple model for shared memory

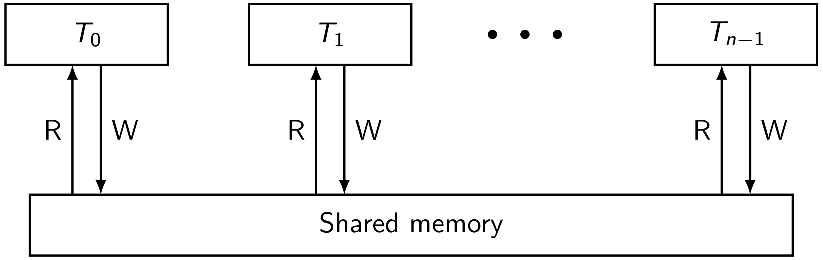
N threads (cores) write to and read from a shared memory.



“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.*

Another, intuitive?, view of SC



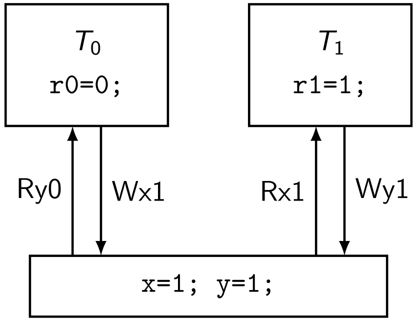
The *program order* is the execution order specified by the program which a thread executes. This ordering extends to “operations” or *events*.

- 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.

Example

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

T_0	T_1
$(a)\,\mathtt{x} \leftarrow 1$	$(c)\,\mathtt{y} \leftarrow 1$
$(b)\,\mathtt{r0} \leftarrow \mathtt{y}$	$(d)\,\mathtt{r1} \leftarrow \mathtt{x}$



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

Simple question on SC execution

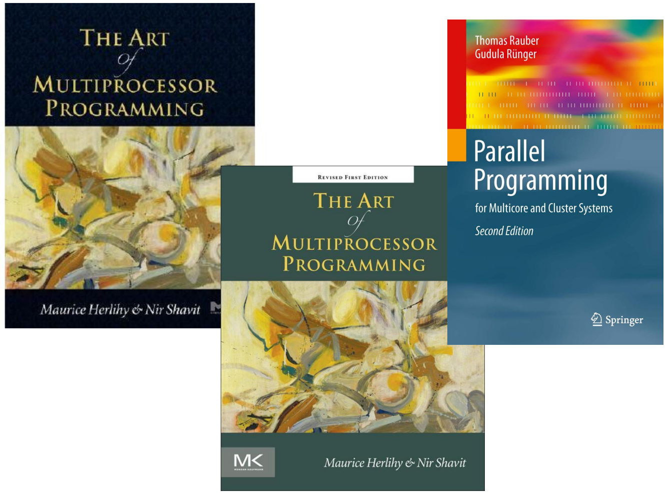
Is final observation `r0=0; r1=0;` possible?

T_0	T_1
(a) `x` $\leftarrow$ `1`	(c) `y` $\leftarrow$ `1`
(b) `r0` $\leftarrow$ `y`	(d) `r1` $\leftarrow$ `x`

No.

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

ProgrammersExperts often assume SC!



A typical concurrent program

```
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*256 → 512 times, `x` final value is 2*256 → 512.

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

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

```
            T0            ||       T1
...                       || ...
  int tmp = x ;           ||   int tmp = x ;
  x = tmp+1 ;             ||   x = tmp+1 ;
...                       || ...
```

For instance,

$$\cdots\ T_0{:}\mathrm{Rx}(v)\ \ T_1{:}\mathrm{Rx}(v)\ \ T_1{:}\mathrm{Wx}(v+1)\ \ T_0{:}\mathrm{Rx}(v+1)\cdots$$

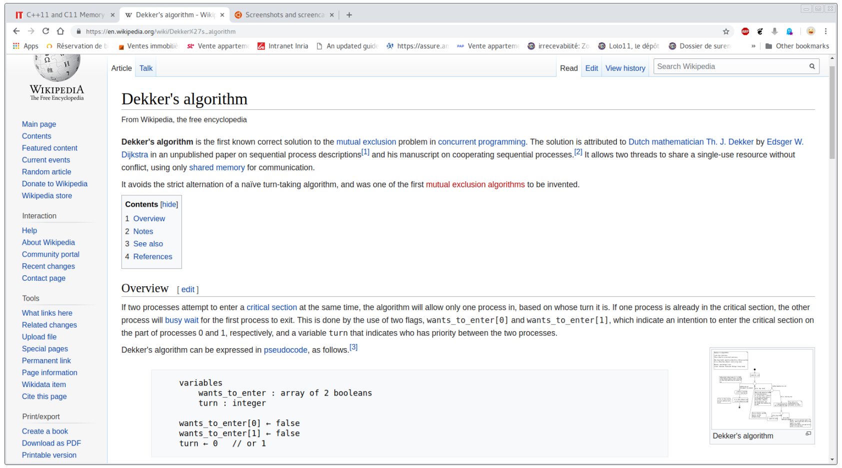
Solution: RW become scheduling atoms,

$$\cdots[T_0{:}\mathrm{Rx}(v)\ \ T_0{:}\mathrm{Wx}(v+1)]\quad[T_1{:}\mathrm{Rx}(v+1)\ \ T_1{:}\mathrm{Wx}(v+2)]\cdots$$

Mutual exclusion

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

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



Dekker's algorithm

From Wikipedia, the free encyclopedia

Dekker's algorithm is the first known correct solution to the mutual exclusion problem in concurrent programming. The solution is attributed to Dutch mathematician Th. J. Dekker by Edsger W. Dijkstra in an unpublished paper on sequential process descriptions[1] and his manuscript on cooperating sequential processes.[2] It allows two threads to share a single-use resource without conflict, using only shared memory for communication.

It avoids the strict alternation of a naïve turn-taking algorithm, and was one of the first mutual exclusion algorithms to be invented.

Contents [hide]

1 Overview
2 Notes
3 See also
4 References

Overview [edit]

If two processes attempt to enter a critical section at the same time, the algorithm will allow only one process in, based on whose turn it is. If one process is already in the critical section, the other process will busy wait for the first process to exit. This is done by the use of two flags, `wants_to_enter[0]` and `wants_to_enter[1]`, which indicate an intention to enter the critical section on the part of processes 0 and 1, respectively, and a variable `turn` that indicates who has priority between the two processes.

Dekker's algorithm can be expressed in pseudocode, as follows.[3]

```
variables
    wants_to_enter : array of 2 booleans
    turn : integer

wants_to_enter[0] ← false
wants_to_enter[1] ← false
turn ← 0   // or 1
```

Dekker's algorithm

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)`.

T_0

```
  int id = 0;
...
  lock(id) ;
  int tmp = x ;
  x = tmp+1 ;
  unlock(id) ;
...
```

T_1

```
  int id = 1;
...
  lock(id) ;
  int tmp = x ;
  x = tmp+1 ;
  unlock(id) ;
...
```

Code from a reliable source (Wikipedia)

```
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:

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

Let us simplify even more:

T_0	T_1
(a) `x` $\leftarrow$ 1 (b) `r0` $\leftarrow$ `y`	(c) `y` $\leftarrow$ 1 (d) `r1` $\leftarrow$ `x`

Can we observe `r0=0; r1=0;` ? If so, Dekker's locking code does not guarantee mutual exclusion.

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

Demo: `tst/Machine/Dekker.litmus`

To avoid compiler interference, we run assembly code:

```
X86_64 Dekker
{ want0=0; want1=0; }
 P0                 | P1                 ;
 movl $1,(want0)    | movl $1,(want1)    ;
 movl (want1),%eax  | movl (want0),%eax  ;
exists (0:rax=0 /\ 1:rax=0)
```

We run the test several times with the `litmus` tool:

```
% litmus7 -mach x86_64 Dekker.litmus
..
Test Dekker Allowed
Histogram (4 states)
178    *>0:rax=0; 1:rax=0;
1999870:>0:rax=1; 1:rax=0;
1999881:>0:rax=0; 1:rax=1;
71     :>0:rax=1; 1:rax=1;
...
```

We observe the non-SC outcome 178 times out of 4 millions attempts.

The horrible truth

Modern processors perform many optimisations:

- out of order execution;
- speculative execution;
- in-core store buffers;
- cache hierarchies. . .

These are

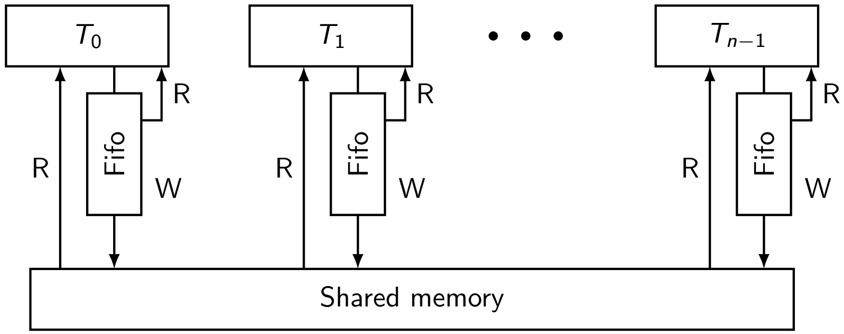
- unobservable by single-thread programs;
- sometime observable by concurrent programs;

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*.



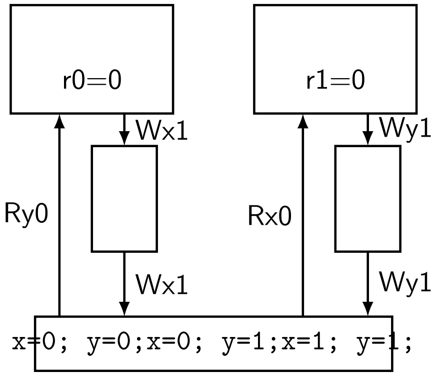
Cores write into their store buffer.

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)

T_0	T_1
(*a*) x ← 1 (*b*) r0 ← y	(*c*) y ← 1 (*d*) r1 ← x



Final state: `r0=0; r1=0;`.

Message passing test

MP

T_0	T_1
(*a*) x ← 1 (*b*) y ← 1	(*c*) r0 ← y (*d*) r1 ← x

Observed? `r0=1; r1=0`

All TSO executions:

F_x F_y (c) (d)	r0=1 r1=1
F_x (c) F_y (d)	r0=0 r1=1
F_x (c) (d) F_y	r0=0 r1=1
(c) F_x F_y (d)	r0=0 r1=1
(c) F_x (d) F_y	r0=0 r1=0
(c) (d) F_x F_y	r0=0 r1=0

Outcome `r0=1 r1=0` is forbidden.

As T_1 must see writes in order, T_1 must see flushes in order.

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 /\ 1:rbx=0)
```

Let us run the test:

```
% litmus7 -mach x86_64 MP.litmus
...
Test MP Allowed
Histogram (3 states)
1999919:>1:rax=0; 1:rbx=0;
3062   :>1:rax=0; 1:rbx=1;
1997019:>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 /\ 1:X2=0)
```

Let us compile and upload on my phone

```
% litmus7 -mach phone -o R MP.litmus
% make -C R
/opt/android-ndk/bin/aarch64-linux-android-gcc -Wall -O2 -pthread M
...
% scp -C -P 2222 R/run.exe 128.93.84.97:MP.exe
```

Run MP on my phone

```
%  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]  |              ;
exists (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.

Restoring SC

Why ? Using all those clever algorithms:

How ? By using specific instructions.

Strong fence

All architectures (I know of) provide a "strong" fence, whose purpose is restoring SC.

Demo: `tst/machine/Dekker+Fences.litmus`

```
% cat Dekker+Fences.litmus
X86_64 Dekker+Fences
{ }
 P0              | P1              ;
 movl $1,(x)     | movl $1,(y)     ;
 mfence          | mfence          ;
 movl (y),%eax   | movl (x),%eax   ;
exists (0:rax=0 /\ 1:rax=0)

% litmus7 -mach x86_64 Dekker+Fences.litmus
...
Test Dekker+Fences Allowed
Histogram (3 states)
1957077:>0:rax=1; 1:rax=0;
1930882:>0:rax=0; 1:rax=1;
112041:>0:rax=1; 1:rax=1;
...
```

Notice: Fences are inserted in-between memory accesses.

Specific store and load instructions

ARMv8 provides store release and load acquire.

Demo: `tst/ARMv8/MP+Rel+Acq.litmus`

```
% 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]  |               ;
exists (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

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 that are disallowed by the untransformed program.

Correctness, half-informal

Whole program approach: one program execution yields a *behaviour* (*e.g.* final state of some variables).

- ▶ Compiler correctness
  - ▷ Given any behaviour of the *compiled* program,
  - ▷ the source program can legitimately produce this behaviour.
- ▶ Compiler non-correcteness:
  - ▷ There exists a behaviour of the compiled program,
  - ▷ which the source program cannot legitimately produce.

A simple optimisation

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

```
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.

Constant propagation is invalid (SC model)

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

Print “0” or nothing

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

Print “1” or nothing

Another optimisation

Re-ordering “independant reads” does not harm (in sequential code).
Compile time:

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

⇓

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

Runtime:

Rxv_1; Ryv_2; ⇒ Ryv_2; Rxv_1;

However, output v_1 , v_2 does not change.

Read reordering is invalid on SC

```
int rx = x ;                     ||  y = 1 ;
int ry = y ;                     ||  x = 1 ;
printf("%i,␣%i\n", rx, ry) ;     ||
```

schedule	output
Wy1; Wx1; Rx1; Ry1	1, 1
Wy1; Rx0; Wx1; Ry1	0, 1
Wy1; Rx0; Ry0; Wx1	0, 0
Rx0; Wy1; Wx1; Ry1	0, 1
Rx0; Wy1; Ry1; Wx1	0, 1
Rx0; Ry0; Wy1; Wx1	0, 0

Read reordering is invalid on SC

```
int ry = y ;                     ||  y = 1 ;
int rx = x ;                     ||  x = 1 ;
printf("%i,␣%i\n", rx, ry) ;     ||
```

schedule	output
Wy1; Wx1; Ry1; Rx1	1, 1
Wy1; Ry1; Wx1; Rx1	1, 1
Wy1; Ry1; Rx0; Wx1	0, 1
Ry0; Wy1; Wx1; Rx1	1, 0
Ry0; Wy1; Rx0; Wx1	0, 0
Ry0; Rx0; Wy1; Wx1	0, 0

Additional output: 1, 0

Does it happen?

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

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

⇓

```
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
```

Reality check

Compile and run:

```
% litmus7 -mach ../tst.cfg -o R MP-LOOP.litmus
% cd R
% make
...
% sh run.sh
...
Test MP-LOOP Allowed
10000137:>0:rx=0; 0:ry=0;
129   *>0:rx=1; 0:ry=0;
281   :>0:rx=0; 0:ry=1;
9999453:>0:rx=1; 0:ry=1;
...
```

Bingo!

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) {
  pait_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.

Even worse

Let z be a pointer to shared memory.

```
pair_t p = f(z,z,1) ;              ||
// p.r0 is read first, then p.r1   ||  *z = 1 ;
printf("%i,␣%i\n",p.r0, p.r1);     ||  *z = 2 ;
```

One expects output:

schedule	output
Wz1; Wz2; Rz2; Rz2	2, 2
Wz1; Rz1; Wz2; Rz2	1, 2
Wz1; Rz1; Rz1; Wz2	1, 1
Rz0; Wz1; Wz2; Rz2	0, 2
Rz0; Wz1; Rz1; Wz2	0, 1
Rz0; Rz0; Wz1; Wz2	0, 0

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

Even worse

Let z be a pointer to shared memory.

```
pair_t p = f(z,z,1) ;              ||
// p.r1 is read first, then p.r0   ||  *z = 1 ;
printf("%i,␣%i\n",p.r0, p.r1);     ||  *z = 2 ;
```

One gets output:

schedule	output
Wz1; Wz2; Rz2; Rz2	2, 2
Wz1; Rz1; Wz2; Rz2	2, 1
Wz1; Rz1; Rz1; Wz2	1, 1
Rz0; Wz1; Wz2; Rz2	2, 0
Rz0; Wz1; Rz1; Wz2	1, 0
Rz0; Rz0; Wz1; Wz2	0, 0

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

Really even worse

Consider the simple CoRR program

```
int r0 = *z ;                ||
int r1 = *z ;                ||  *z = 1 ;
printf("%i,␣%i\n",r0, r1);   ||  *z = 2 ;
```

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

schedule	output
Wz1; Wz2; Rz2; Rz2	2, 2
Wz1; Rz1; Wz2; Rz2	1, 2 or 2, 1
Wz1; Rz1; Rz1; Wz2	1, 1
Rz0; Wz1; Wz2; Rz2	0, 2 or 2, 0
Rz0; Wz1; Rz1; Wz2	0, 1 or 1, 0
Rz0; Rz0; Wz1; Wz2	0, 0

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 canot guarantee anything.

What to do?

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

Languages options, accepting undefined behaviours or not.

1. C11/C++11, POSIX threads, ADA 83
2. 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 “simultaneaously”.

Disallowing conflicting accesses looks too drastic.

Disallowing races hence means avoiding concurrency. This looks plausible.

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

A racy program

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

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

schedule	race?
Wy1; Wx1; Rx1; Ry1	Ok
Wy1; Rx0; Wx1;	Ok
Rx0; Wy1; Wx1;	No

Important: We quantify over SC executions.

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

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:

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

schedule	race?
Wy1; L(ℓ); Wx1; U(ℓ); L(ℓ); Rx1; U(ℓ); Ry1	No
Wy1; L(ℓ); Rx0; U(ℓ); L(ℓ); Wx1; U(ℓ);	No
L(ℓ); Wy1; Rx0; U(ℓ); L(ℓ); Wx1; U(ℓ);	No
L(ℓ); Rx0; Wy1; U(ℓ); L(ℓ); Wx1; U(ℓ);	No
L(ℓ); Rx0; U(ℓ); Wy1; L(ℓ); Wx1; U(ℓ);	No

Another well synchronised program

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

schedule	race?
L(ℓ); Wy1; Wx1; U(ℓ); L(ℓ); Rx1; Ry1; U(ℓ)	No
L(ℓ); Rx0; U(ℓ); L(ℓ); Wy1Wx1; U(ℓ)	No

Races can be worse than being non-SC

Let x be a non-aligned pointer to some `int` in shared memory.

```
*x = 0x01010202; || 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    :>1:r1=0x202;
15      :>1:r1=0x1010000;
9998369:>1:r1=0x1010202;
...
```

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 wen there is none.
- Synchronising calls contain "sufficient fences" to prevent hardware reordering.

Semantics of programming languages

1. No concurrency at all (OCaml). Well, not very fashionable.
2. No shared memory (Erlang, MPI). Possible, but not a "natural" generalisation of sequential programming.
3. Enforce data-race freedom statically. Not general-purpose.
4. Leave it to the hardware (Aligned C,ML-toon). Not portable.
5. Complete solutions, DRF, plus
   1. DRF as a definition: racy-programs can behave in any way (catch fire semantics).
   2. 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, their semantics can be complex.

Some references

Introduction

"Memory Models: A Case for Rethinking Parallel Languages and Hardware" Sarita V. Adve and Hans-J. Boehm. Commun. ACM 53(8): (2010) pp. 90-101.

"Shared Memory Consistency Models: a Tutorial" Sarita V. Adve and Kourosh Gharacorloo, IEEE Computer 29, 12 (1996) pp. 66–76.

Some references

On Hardware models:

"A Rigorous and Usable Programmers Model for x86 Multiprocessors" Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. Commun. ACM 53(7): 89-97 (2010).

"Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory" Jade Alglave, Luc Maranget, Michael Tautschnig: ACM Trans. Program. Lang. Syst. 36(2): 7:1-7:74 (2014)

On languages:

"Foundations of the C++ concurrency memory model"' Hans-Juergen Boehm, Sarita V. Adve: PLDI 2008: 68-78

"Repairing Sequential Consistency in C/C++11" Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur and Derek Dreyer: PLDI 2017.