Program logics, sixth lecture

Program logics for weakly-consistent memory

Xavier Leroy

2021-04-08

Collège de France, chair of software sciences
`xavier.leroy@college-de-france.fr`

Sequential consistency: an idealized model for concurrent programming

Sequential consistency

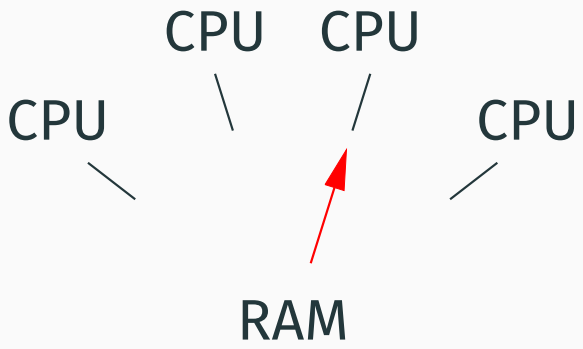
> *A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program*
>
> *(L. Lamport, 1978)*

For a shared-memory system: the state of the shared memory is the result of an interleaving of the memory operations of the processes.

Two sequentially-consistent (SC) implementations

1. Time-sharing on a monoprocessor.

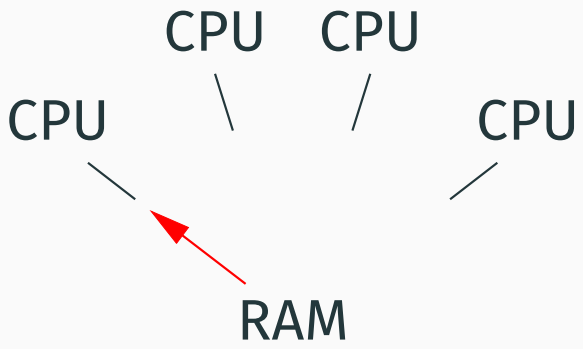
2. Multiprocessor system with a single “port” to access memory.



Two sequentially-consistent (SC) implementations

1. Time-sharing on a monoprocessor.

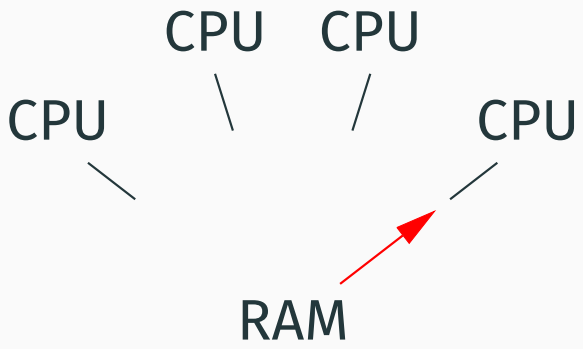
2. Multiprocessor system with a single “port” to access memory.



Two sequentially-consistent (SC) implementations

1. Time-sharing on a monoprocessor.

2. Multiprocessor system with a single “port” to access memory.



A straightforward formal semantics

Reminder from lecture #4:

Semantics for the $c_1 \parallel c_2$ construct:
an interleaving of the reductions of c_1 and c_2.

$$(a_1 \parallel a_2)/h \to 0/h \qquad \text{(or any combination of } a_1 \text{ and } a_2\text{)}$$

$$(c_1 \parallel c_2)/h \to (c_1' \parallel c_2)/h' \qquad \text{if } c_1/h \to c_1'/h'$$

$$(c_1 \parallel c_2)/h \to (c_1 \parallel c_2')/h' \qquad \text{if } c_2/h \to c_2'/h'$$

$$(c_1 \parallel c_2)/h \to \texttt{err} \qquad \text{if } c_1/h \to \texttt{err} \text{ or } c_2/h \to \texttt{err}$$

Our semantics for parallelism in PTR was already SC!

Programming in an SC model

The SC model defines precisely the semantics of concurrent programs even when they contain race conditions.

This opens the way to concurrent algorithms that use race conditions in a well-controlled manner.

These algorithms are useful when the atomic instructions of the processor or the critical sections of the language are lacking or too expensive.

Peterson's mutual exclusion algorithm

$$flag : bool[2] = \{\texttt{false}, \texttt{false}\}; turn : \{0, 1\};$$

$flag[0] := \texttt{true};$	$flag[1] := \texttt{true};$
$turn := 1;$	$turn := 0;$
$\texttt{while}\ flag[1] \wedge turn = 1$	$\texttt{while}\ flag[0] \wedge turn = 0$
$\texttt{do skip done};$	$\texttt{do skip done};$
// enter critical section	// enter critical section
...	...
// leave critical section	// leave critical section
$flag[0] := \texttt{false}$	$flag[1] := \texttt{false}$

Peterson's mutual exclusion algorithm

$$flag : bool[2] = \{\texttt{false}, \texttt{false}\}; turn : \{0, 1\};$$

$flag[0] := \texttt{true};$	$flag[1] := \texttt{true};$
$turn := 1;$	$turn := 0;$
$\texttt{while}\ flag[1] \wedge turn = 1$	$\texttt{while}\ flag[0] \wedge turn = 0$
$\texttt{do skip done};$	$\texttt{do skip done};$
// enter critical section	// enter critical section
...	...
// leave critical section	// leave critical section
$flag[0] := \texttt{false}$	$flag[1] := \texttt{false}$

With an enumeration of SC interleavings, we can show that both processes are simultaneously in their critical sections only if $flag[0] = flag[1] = \texttt{true} \wedge turn = 0 \wedge turn = 1$,
which is impossible.

The ticket lock algorithm

A process that tries to enter the critical section

- takes the next ticket (atomic increment)
- waits for the number on its ticket to be displayed.

When it leaves the critical section, it ensures the next number is displayed.

The ticket lock algorithm

Two global variables: `next` and `now_serving`, initially 0.

```
lock() {
    int t = fetch_and_add(&next, 1);     // atomic increment
    while (now_serving != t) pause();    // nonatomic read
}

unlock() {
    now_serving = now_serving + 1;       // nonatomic increment
}
```

The increment of `next` must be atomic
(otherwise two processes could get the same ticket).

Accesses to `now_serving` need not be atomic.

The real world: weakly-consistent memory models

A litmus test

A fragment of Dekker's algorithm for mutual exclusion:

$$\begin{array}{l} \texttt{set}(X, 1); \\ \texttt{let}\ a = \texttt{get}(Y) \end{array} \quad \Big\| \quad \begin{array}{l} \texttt{set}(Y, 1); \\ \texttt{let}\ b = \texttt{get}(X) \end{array}$$

Initially, $X = Y = 0$.

In the SC model:

- either $\texttt{set}(X, 1)$ executes first, and then $b = 1$ at the end;
- either $\texttt{set}(Y, 1)$ executes first, and then $a = 1$ at the end.

Therefore, the final state $a = b = 0$ is impossible.

Experimental refutation

We write the test in x86 assembly (to get full control on the 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 execute the test with the `litmus7` tool:

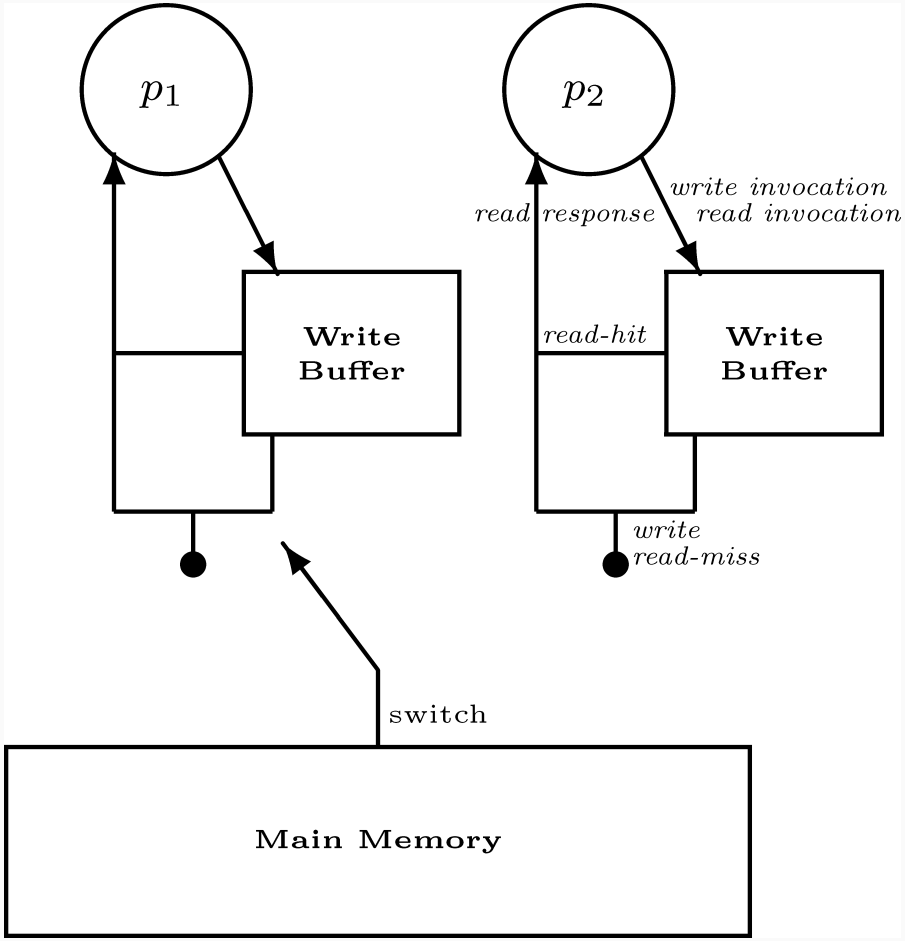
```
Test Dekker Allowed
Histogram (4 states)
178     *>0:rax=0; 1:rax=0;
1999870:>0:rax=1; 1:rax=0;
1999881:>0:rax=0; 1:rax=1;
71      :>0:rax=1; 1:rax=1;
```

Harsh reality

The sequential consistency model is respected

- neither by the modern hardware architectures
  (in order to provide faster memory subsystems).
- nor by optimizing compilers
  (in order to increase performance of generated code).

Hardware: write buffers, store buffers



(Higham, Jackson, Kawash, 2007)

Each processor puts its writes in a buffer while they are transmitted to the shared main memory.

Writes performed by a processor are immediately visible by this processor, but not immediately by the other processors.

A non-SC execution

$$\begin{array}{l} \texttt{set}(X, 1); \\ \texttt{let } a = \texttt{get}(Y) \end{array} \quad \Big\| \quad \begin{array}{l} \texttt{set}(Y, 1); \\ \texttt{let } b = \texttt{get}(X) \end{array}$$

	Processor 1	Processor 2
Time $t = 0$	puts $X \leftarrow 1$ in its buffer	puts $Y \leftarrow 1$ in its buffer
Time $t = 1$	reads $Y = 0$ from main memory	
Time $t = 2$		reads $X = 0$ from main memory
Time $t = 3$	sends $X \leftarrow 1$ to main memory	
Time $t = 4$		sends $Y \leftarrow 1$ to main memory

Hardware: out-of-order, speculative execution

The processor can reorder instructions on the fly, so as to start long-running instructions earlier (e.g. memory reads).

$$\text{write } X; \ldots; \text{read } Y \rightarrow \text{read } Y; \text{write } X; \ldots$$

This out-of-order execution is often speculative: if the processor realizes that $X = Y$, it cancels the anticipated read from Y, or satisfies it with the value written to X (forwarding).

Another non-SC execution

Machine code:

$$\begin{array}{l} \texttt{set}(X, 1); \\ \texttt{let } a = \texttt{get}(Y) \end{array} \quad \Big\| \quad \begin{array}{l} \texttt{set}(Y, 1); \\ \texttt{let } b = \texttt{get}(X) \end{array}$$

Code actually executed by the processor after on-the-fly reordering:

$$\begin{array}{l} \texttt{let } a = \texttt{get}(Y) \texttt{ in} \\ \texttt{set}(X, 1) \end{array} \quad \Big\| \quad \begin{array}{l} \texttt{let } b = \texttt{get}(X) \texttt{ in} \\ \texttt{set}(Y, 1) \end{array}$$

The reordered code can obviously terminate with $a = b = 0$.

Hardware: split memory accesses

A misaligned datum can span two cache lines, requiring two memory accesses per access to the datum.

$$\texttt{set}(X, 0x12345678) \;\Big\|\; \texttt{let}\ a = \texttt{get}(X)$$

can be executed like

$$\begin{array}{l} \texttt{set}(X_1, 0x1234); \\ \texttt{set}(X_2, 0x5678); \end{array} \;\Big\|\; \begin{array}{l} \texttt{let}\ a_1 = \texttt{get}(X_1)\ \texttt{in} \\ \texttt{let}\ a_2 = \texttt{get}(X_2)\ \texttt{in} \\ \texttt{let}\ a = a_1 << 16 \mid a_2 \end{array}$$

A non-SC with values “out of thin air”

$$\texttt{set}(X, 0x12345678) \;\Big\|\; \texttt{let}\ a = \texttt{get}(X)$$

Starting from $X = 0$, we have two SC executions:
$a = 0$ or $a = 0x12345678$.

$$\begin{array}{l} \texttt{set}(X_1, 0x1234); \\ \texttt{set}(X_2, 0x5678); \end{array} \;\Big\|\; \begin{array}{l} \texttt{let}\ a_1 = \texttt{get}(X_1)\ \texttt{in} \\ \texttt{let}\ a_2 = \texttt{get}(X_2)\ \texttt{in} \\ \texttt{let}\ a = a_1 << 16 \mid a_2 \end{array}$$

After splitting the memory accesses, a third result is possible:
$a = 0x12340000$, coming from $a_1 = 0x1234$ and $a_2 = 0$.

Note: the value $0x12340000$ appears nowhere in the initial code.
It appears out of thin air!

Possible remedies

Use barrier instructions that prevent the processor from reordering certain memory accesses:

- “Strong” barrier: preserves ordering between accesses before the barrier and accesses after the barrier.
- “Weak” barrier: preserves ordering between reads before the barrier and accesses after the barrier.

Other instructions with special memory behaviors:

- “locked” instructions (the x86 `lock` prefix);
- *load-acquire* and *store-release* (Itanium, ARM);
- etc.

Compiler optimizations: instruction reordering

The compiler can reorder independent reads and writes (at addresses X, Y that are guaranteed to be different).

Typically, reads are anticipated while writes are delayed.

$$\text{write } X; \ldots; \text{read } Y \rightarrow \text{read } Y; \text{write } X; \ldots$$

$\rightarrow$ same non-SC behaviors as dynamic reordering by the processor.

Compiler optimization: factoring redundant reads

A special case of common subexpression elimination (CSE).

$$
\begin{array}{l}
\texttt{let } a = \texttt{get}(X) \texttt{ in} \\
\ldots \\
\text{(no writes to } X\text{)} \\
\ldots \\
\texttt{let } b = \texttt{get}(X) \texttt{ in} \\
\ldots
\end{array}
\quad \rightsquigarrow \quad
\begin{array}{l}
\texttt{let } a = \texttt{get}(X) \texttt{ in} \\
\ldots \\
\ldots \\
\ldots \\
\texttt{let } b = a \texttt{ in} \\
\ldots
\end{array}
$$

A non-SC execution

$$\begin{array}{l} \texttt{let } a = \texttt{get}(X) \texttt{ in} \\ \texttt{let } b = \texttt{get}(Y) \texttt{ in} \\ \texttt{let } c = \texttt{get}(X) \end{array} \;\Big\|\; \begin{array}{l} \texttt{set}(X, 1); \\ \texttt{set}(Y, 1); \end{array}$$

With $X = Y = 0$ initially, no SC execution terminates with $(a, b, c) = (0, 1, 0)$.

After factoring of $\texttt{get}(X)$, we have $\texttt{let } c = a$ and the result $(a, b, c) = (0, 1, 0)$ is possible.

Compiler optimizations: loop-invariant code motion

A computation performed repeatedly at each loop iteration can be performed once before the loop:

$$\begin{array}{l} \texttt{for } i = 0 \texttt{ to } 99 \texttt{ do} \\ \quad A[i] := i + j \times 10 \\ \texttt{done} \end{array} \quad \rightsquigarrow \quad \begin{array}{l} t := j \times 10; \\ \texttt{for } i = 0 \texttt{ to } 99 \texttt{ do} \\ \quad A[i] := i + t \\ \texttt{done} \end{array}$$

This can break codes based on busy waiting:

$$\begin{array}{l} \texttt{do} \\ \quad t := \texttt{get}(X) \\ \texttt{while } t = 0 \end{array} \quad \rightsquigarrow \quad \begin{array}{l} t := \texttt{get}(X); \\ \texttt{do} \\ \quad \texttt{skip} \\ \texttt{while } t = 0 \end{array}$$

Possible remedies

Turn optimizations off? Never!

Inform the compiler of which memory accesses implement inter-process communications, so as to compile them specially:

- the `volatile` modifier (C,C++, Java)
- a library of low-level atomic operations (C/C++ 2011)
- etc.

The *low-level atomics* in C/C++ 2011

Various atomic operations: read, write, fetch-and-add, compare-and-swap, ...

Each operation is annotated with the consistency model expected by the programmer:

`memory_order_seq_cst`	sequential consistency
`memory_order_acq_rel`	just enough for message passing
`memory_order_acquire`	
`memory_order_release`	
`memory_order_consume`	
`memory_order_relaxed`	no guarantees beyond atomicity

DRF + CSL = ♡

Concurrent separation logic and the DRF guarantee

The Data Race Free (DRF) guarantee

A property of a relaxed memory model:

> *If a program executes in the SC model without race conditions,*
>
> *then it executes in the relaxed model exactly like in the SC model.*

In other words: for a program free of race conditions, the relaxations of the memory model do not add more behaviors beyond those permitted by SC.

This “DRF guarantee” seems to hold / is claimed to hold for all the known memory models (hardware models + language models).

The extended DRF guarantee

> *If a program comprising critical sections, atomic operations, and other synchronization devices executes in the SC model without race conditions,*
>
> *then it executes in the relaxed model exactly like in the SC model,*
>
> *provided the synchronization devices are correctly implemented.*

"Correctly implemented" = with enough memory barriers and special instructions to rule out non-SC behaviors.

Examples: implementing locks

x86: lock

```
     movl  $1, %edx
.L2: movl  %edx, %eax
     xchgb (%rdi), %al
     testb %al, %al
     jne   .L2
```

x86: unlock

```
     movb  $0, (%rdi)
```

Power: lock

```
.L2: lbarx  9,0,3
     stbcx. 10,0,3
     bne    0,.L2
     isync
     andi.  9,9,0xff
     bne    0,.L2
```

Power: unlock

```
     lwsync
     li  9,0
     stb 9,0(3)
```

A subtle point of the x86 implementation

Before 1999, the Linux kernel implemented `unlock` with an atomic instruction

```
lock; btr $0, (...)
```

instead of a nonatomic write

```
movb $0, (...)
```

A long discussion concluded that a nonatomic write is enough, because the x86 memory model is TSO.

The write of 0 in the lock is not immediately visible by other processes waiting for the lock. But when it becomes visible, all preceding writes are already visible, and all preceding reads have been performed.

Compatibility with compiler optimizations

A great many compiler optimizations are sound for programs that contain no race conditions.

(Sound = all behaviors of the optimized program are possible behaviors of the original program. Optimization did not introduce additional behaviors.)

Compatibility with compiler optimizations

Transformation	SC	DRF guarantee	JMM
Trace-preserving transformations	✓	✓	✓
Reordering normal memory accesses	×	✓	×
Redundant read after read elimination	✓	✓	×
Redundant read after write elimination	✓	✓	✓
Irrelevant read elimination	✓	✓	✓
Irrelevant read introduction	✓	?	×
Redundant write before write elimination	✓	✓	✓
Redundant write after read elimination	✓	✓	×
Roach-motel reordering	×(✓for locks)	✓	×
External action reordering	×	✓	×

J. Ševčik, *Program Transformations in Weak Memory Models*, PhD, 2008.

How to prove the absence of race conditions?

The assumption “the program has no race conditions in the SC model” is strong! How to establish it?

- *Ad-hoc* proof.
- Type system ($\approx$ Rust).
- Static analysis (Infer, etc).
- Deductive verification in concurrent separation logic!

Reminder (lecture #4): if $J \vdash \{ P \}\ c\ \{ Q \}$, then c executes without race conditions in an interleaving semantics that is equivalent to the SC model.

The CSL guarantee (Concurrent Separation Logic)

If a program is provable in concurrent separation logic (including critical sections, atomic sections, etc),

then it executes correctly in a relaxed memory model that respects the DRF guarantee,

provided that critical sections and atomic sections are correctly implemented.

A direct proof of the CSL guarantee for a TSO model

We now show semantic soundness for concurrent separation logic in a variant of our PTR language extended with write buffers (TSO model):

Write buffers: $s ::= \varepsilon \mid (\ell, v) \cdot s$

We consider whole-program configurations

$$((c_1/s_1) \parallel \cdots \parallel (c_n/s_n)) \ / \ h$$

composed of n processes $c_1 \ldots c_n$, each with its own buffer s_i, plus a global heap h.

We also consider local, per-process configurations

$$c/s/h$$

Local reductions

At any time a process can perform the oldest write from its buffer:

$$c/s \cdot (\ell, v)/h \to c/s/h[\ell \leftarrow v]$$

The base language constructs have their usual semantics:

$$\begin{aligned}
(\texttt{let}\ x = a\ \texttt{in}\ c)/s/h &\to c[x \leftarrow [\![a]\!]]/s/h \\
(\texttt{let}\ x = c_1\ \texttt{in}\ c_2)/s/h &\to (\texttt{let}\ x = c_1'\ \texttt{in}\ c_2)/s'/h' \\
&\qquad \text{if } c_1/s/h \to c_1'/s'/h' \\
(\texttt{let}\ x = c_1\ \texttt{in}\ c_2)/s/h &\to \texttt{err} \quad \text{if } c_1/s/h \to \texttt{err}
\end{aligned}$$

Local reductions: imperative constructs

Imperative constructs write to s (the buffer) and read from $s \rhd h$, the heap h updated as described by s:

$$\varepsilon \rhd h = h \qquad ((\ell, v) \cdot s) \rhd h = (s \rhd h)[\ell \leftarrow v]$$

$$\mathtt{get}(a)/s/h \to (s \rhd h)([\![a]\!])/s/h \quad \text{if } [\![a]\!] \in \mathit{Dom}(s \rhd h)$$

$$\mathtt{set}(a, a')/s/h \to 0/([\![a]\!], [\![a']\!]) \cdot s/h \quad \text{if } [\![a]\!] \in \mathit{Dom}(s \rhd h)$$

$$\mathtt{get}(a)/s/h \to \mathtt{err} \quad \text{if } [\![a]\!] \notin \mathit{Dom}(s \rhd h)$$

$$\mathtt{set}(a, a')/s/h \to \mathtt{err} \quad \text{if } [\![a]\!] \notin \mathit{Dom}(s \rhd h)$$

Local reductions: atomic sections

As in PTR, atomic sections execute in a single "big step":

$$\texttt{atomic}(c)/s/h \to a/\varepsilon/h' \qquad \text{if} \quad c/s/h \xrightarrow{*} a/\varepsilon/h'$$

$$\texttt{atomic}(c)/s/h \to \texttt{err} \qquad \text{if} \quad c/s/h \xrightarrow{*} \texttt{err}$$

However, the buffer must be empty at the end of the atomic section ($\approx$ there is a write barrier at the end).

When we create a resource invariant, the buffer must also be empty, hence the $\texttt{mkinv}(c)$ construct:

$$\texttt{mkinv}(c)/\varepsilon/h \to c/\varepsilon/h$$

Global reductions

At each step we locally reduce one of the processes c_i/s_i from the parallel composition; the other processes are unchanged.

$$\frac{c_i/s_i/h \to c'/s'/h'}{(\cdots \parallel (c_i/s_i) \parallel \cdots) \,/\, h \to (\cdots \parallel (c'/s') \parallel \cdots) \,/\, h'}$$

$$\frac{c_i/s_i/h \to \texttt{err}}{(\cdots \parallel (c_i/s_i) \parallel \cdots) \,/\, h \to \texttt{err}}$$

Semantic soundness of concurrent separation logic

For PTR in the SC model, we decomposed the current heap h in three disjoint parts:

$$h = h_1 \uplus h_j \uplus h_f$$

h_1 is the private memory for c.
h_j is the shared memory accessible to atomic sections.
h_f is the "frame" memory, including the private memories of the processes that execute in parallel with c.

Semantic soundness of concurrent separation logic

For PTR in the TSO model, we decompose the main heap h and the buffer s for the current process c as follows:

$$h = h_u \uplus h_j \qquad s \rhd h_u = h_1 \uplus h_f$$

The main heap h decomposes into shared memory h_j and unshared memory h_u.

The unshared memory h_u, updated according to the buffer s, decomposes into h_1, the private memory for c, and h_f, the frame.

Equivalent presentation:

$$s \rhd h = h_1 \uplus h_j \uplus h_f \quad \text{with} \quad Dom(s) \cap Dom(h_j) = \emptyset$$

The semantic triple

We define the semantic triple $J \models \{\!\{ P \}\!\}\ c\ \{\!\{ Q \}\!\}$ as follows:

$$J \models \{\!\{ P \}\!\}\ c\ \{\!\{ Q \}\!\} \quad \stackrel{def}{=} \quad \forall n, h,\ P\ h \Rightarrow \texttt{Safe}^n\ c\ h\ Q\ J$$

As in lecture #4, we have:

$$\texttt{Safe}^0\ c\ h\ Q\ J \qquad \frac{Q\ [\![a]\!]\ h}{\texttt{Safe}^{n+1}\ a\ h\ Q\ J} \qquad \frac{(\forall a, c \neq a) \qquad \cdots}{\texttt{Safe}^{n+1}\ c\ h\ Q\ J}$$

The semantic triple

The recursive case: one reduction step for $c/s/h$.

$$\forall a, c \neq a$$

$$\forall s, h, h_j, h_f, s \rhd h = h_1 \uplus h_j \uplus h_f \wedge Dom(s) \cap Dom(h_j) = \emptyset \wedge J\ h_j \Rightarrow c/s/h \not\to \texttt{err}$$

$$\forall s, h, h_j, h_f, c', s', h', s \rhd h = h_1 \uplus h_j \uplus h_f \wedge Dom(s) \cap Dom(h_j) = \emptyset \wedge J\ h_j \wedge c/s/h \to c'/s'/h' \Rightarrow \exists h_1', h_j',\ s' \rhd h' = h_1' \uplus h_j' \uplus h_f \wedge Dom(s') \cap Dom(h_j') = \emptyset \wedge J\ h_j' \wedge \texttt{Safe}^n\ c'\ h_1'\ Q$$

$$\texttt{Safe}^{n+1}\ c\ h_1\ Q$$

Validating the rules of the logic

It remains to show that this semantic triple $J \models \{\!\{ P \}\!\}\ c\ \{\!\{ Q \}\!\}$ validates the rules of concurrent separation logic. The two interesting cases deal with atomic sections.

$$\frac{\mathtt{emp} \vdash \{\, P \mathbin{\ast} J \,\}\ c\ \{\, \lambda v.\ Q\ v \mathbin{\ast} J \,\}}{J \vdash \{\, P \,\}\ \mathtt{atomic}\ c\ \{\, Q \,\}}$$

At the end of the execution of c we have a decomposition $\varepsilon \rhd h' = (h'_1 \uplus h'_j) \uplus \emptyset \uplus h_f$ that we rewrite as $\varepsilon \rhd h' = h'_1 \uplus h'_j \uplus h_f$. It is crucial that the final buffer s' is ε, otherwise the constraint $Dom(s') \cap Dom(h'_j) = \emptyset$ could not be satisfied.

Validating the rules of the logic

Consider adding an invariant J' to J:

$$\frac{J \mathbin{\ast} J' \vdash \{\, P \,\}\ c\ \{\, Q \,\}}{J \vdash \{\, P \mathbin{\ast} J' \,\}\ \texttt{mkinv}\ c\ \{\, \lambda v.\ Q\ v \mathbin{\ast} J' \,\}}$$

At the beginning of the execution, we have a decomposition $s \rhd h = (h_1 \uplus h'_j) \uplus h_j \uplus h_f$ that we rewrite as $s \rhd h = h_1 \uplus (h_j \uplus h'_j) \uplus h_f$.

Here too we must enforce $s = \varepsilon$ to satisfy $\mathit{Dom}(s) \cap \mathit{Dom}(h_j \uplus h'_j) = \emptyset$.

The `mkinv` construct forces the buffer to be empty.

A limitation and a strength

A limitation: our formalization fails to justify the implementation of $\texttt{unlock}(\ell) = \texttt{atomic}(\texttt{set}(\ell, 0))$ by a normal write, without flushing the store buffer.

This is an aspect of TSO that our formalization does not capture.

A strength: this makes our proof reusable for memory models that are more relaxed than TSO, in particular PSO (*Partial Store Ordering*), where this optimization of `unlock` is invalid.

From TSO to PSO

In the PSO model, writes to different locations can be reordered, and therefore "leave" the write buffer in a different order than execution order:

$$c/s_1 \cdot (\ell, v) \cdot s_2/h \to c/s_1 \cdot s_2/h[\ell \leftarrow v] \quad \text{if } \ell \notin Dom(s_2)$$

Here, the write to ℓ "overtakes" the writes in s_2.

This makes no difference for the soundness proof of the logic, since

$$(s_1 \cdot (\ell, v) \cdot s_2) \rhd h = (s_1 \cdot s_2) \rhd (h[\ell \leftarrow v]) \quad \text{if } \ell \notin Dom(s_2)$$

A logic for release-acquire

The release-acquire model

A write marked “release” guarantees that all preceding reads and writes have been performed.

(No reordering of $X; W_{rel}$ into $W_{rel}; X$.)

A read marked “acquire” guarantees that all following reads and writes have not started yet.

(No reordering of $R_{acq}; X$ into $X; R_{acq}$.)

Use for message passing

// preparing the message	// waiting for the message
// nonatomic (na) writes	// read *ready* until $\neq 0$
$\texttt{set}_{na}(msg, \ldots);$	$\texttt{while get}_{acq}(ready) = 0 \texttt{ do skip};$
$\texttt{set}_{na}(msg + 1, \ldots);$	// accessing the message
$\texttt{set}_{na}(msg + 2, \ldots);$	// nonatomic reads
// sending the message	$\texttt{let } x = \texttt{get}_{na}(msg) \texttt{ in}$
$\texttt{set}_{rel}(ready, 1)$	$\ldots$

A lightweight form of synchronization and resource transfer, without mutual exclusion.

Use for implementing locks

Unlocking a lock is just one release write:

$$\mathit{unlock}(\ell) = \mathtt{set}_{\mathit{rel}}(\ell, 0)$$

Locking a lock requires an atomic instruction such as Compare And Swap, marked “acquire”:

$$\mathit{lock}(\ell) = \mathtt{while}\ (\mathtt{CAS}_{\mathit{acq}}(\ell, 0, 1) \neq 0)\ \mathtt{do}\ \mathtt{skip}$$

We can improve performance with a busy-wait loop using relaxed reads:

$$\mathit{spin}(\ell) = \mathtt{while}\ (\mathtt{get}_{\mathit{rlx}}(\ell) \neq 0)\ \mathtt{do}\ \mathtt{skip}$$
$$\mathit{lock}(\ell) = \mathtt{do}\ \mathit{spin}(\ell)\ \mathtt{while}\ (\mathtt{CAS}_{\mathit{acq}}(\ell, 0, 1) \neq 0)$$

Implementing the release/acquire model

For TSO architectures like x86: nothing to do!

- Ordinary writes have release semantics.
- Ordinary loads have acquire semantics.

For more relaxed architectures like Power and ARM:

- Memory barriers that are less costly than the barriers needed to guarantee SC.

Relaxed Separation Logic (Vafeiadis & Narayan, 2013)

A concurrent separation logic for a fragment of the low-level atomics from C/C++ 2011.

c	$::= a$	pure expression
	$\mid \texttt{let}\ x = c\ \texttt{in}\ c'$	sequencing and binding
	$\mid \texttt{if}\ a\ \texttt{then}\ c_1\ \texttt{else}\ c_2$	conditional
	$\mid \texttt{repeat}\ c$	repeat until not 0
	$\mid c_1 \parallel c_2$	parallel execution
	$\mid \texttt{alloc}()$	allocation
	$\mid \texttt{get}_X(a)$	memory read
	$\mid \texttt{set}_Y(a, a')$	memory write
	$\mid \texttt{CAS}_{Z,X}(a, a', a'')$	Compare And Swap
X	$::= sc \mid acq \mid rlx \mid na$	type of read
Y	$::= sc \mid rel \mid rlx \mid na$	type of store
Z	$::= sc \mid rel\_acq \mid acq \mid rel$	type of CAS

Assertions in RSL

Assertions, preconditions:

$$
\begin{aligned}
P ::= {} & \langle A \rangle \mid \texttt{true} \mid P \mathbin{\ast} P' \\
& \mid \ell \overset{\pi}{\mapsto} v && \text{location } \ell \text{ contains } v \text{ with permission } \pi \\
& \mid \mathit{Acq}(\ell, \Phi) \mid \mathit{Rel}(\ell, \Phi) && \text{resource invariants} \\
& \mid \mathit{RMWAcq}(\ell, \Phi) && \text{resource invariant} \\
& \mid \mathit{Init}(\ell) && \text{invariant was established} \\
& \mid \mathit{Uninit}(\ell) && \text{location } \ell \text{ is freshly allocated}
\end{aligned}
$$

Postconditions, resource invariants:

$Q, \Phi ::= \lambda v.\ P$

Nonatomic accesses

Nonatomic reads and writes follow standard separation logic:

$$\begin{array}{rcl}
\{\,\mathtt{emp}\,\} & \mathtt{alloc}() & \{\,\lambda \ell.\ \mathit{Uninit}(\ell)\,\} \\
\{\,\ell \overset{\pi}{\mapsto} v\,\} & \mathtt{get}_{na}(\ell) & \{\,\lambda x.\ \langle x = v\rangle \mathbin{\ast} \ell \overset{\pi}{\mapsto} v\,\} \\
\{\,\mathit{Uninit}(\ell) \vee \ell \overset{1}{\mapsto} _\,\} & \mathtt{set}_{na}(\ell, v) & \{\,\lambda _.\ \ell \overset{1}{\mapsto} v\,\}
\end{array}$$

(The role of *Uninit* is to prevent us from reading from a freshly allocated memory location that has not been initialized yet.)

Release writes, acquire reads

$\mathit{Rel}(\ell, \Phi)$ grants permission to write to location ℓ a value v provided we have the resource $\Phi\ v$.

$\mathit{Acq}(\ell, \Phi)$, in conjunction with $\mathit{Init}(\ell)$, grants permission to read from location ℓ, obtaining a value v and the resource $\Phi\ v$.

$$\begin{array}{rcl}
\{\,\mathtt{emp}\,\} & \mathtt{alloc}() & \{\,\lambda \ell.\ \mathit{Rel}(\ell, \Phi) \mathbin{\ast} \mathit{Acq}(\ell, \Phi)\,\} \\
\{\,\mathit{Rel}(\ell, \Phi) \mathbin{\ast} \Phi\ v\,\} & \mathtt{set}_{\mathit{rel}}(\ell, v) & \{\,\mathit{Rel}(\ell, \Phi) \mathbin{\ast} \mathit{Init}(\ell)\,\} \\
\{\,\mathit{Acq}(\ell, \Phi) \mathbin{\ast} \mathit{Init}(\ell)\,\} & \mathtt{get}_{\mathit{acq}}(\ell) & \{\,\lambda v.\ \Phi\ v \mathbin{\ast} \mathit{Acq}(\ell, \Phi[v \leftarrow \mathtt{emp}])\,\}
\end{array}$$

We can read the same value multiple times, but the second and subsequent reads transfer no resources:

$$\Phi[v \leftarrow \mathtt{emp}] \stackrel{\mathit{def}}{=} \lambda v'.\ \mathtt{if}\ v' = v\ \mathtt{then}\ \mathtt{emp}\ \mathtt{else}\ \Phi\ v'.$$

Example of resource transfer

We have a (nonatomic) buffer b and an (atomic) flag x.
We take $\Phi = \lambda v.\ \texttt{if}\ v = 0\ \texttt{then}\ \texttt{emp}\ \texttt{else}\ b \overset{1}{\mapsto} 53$.

$$\texttt{let}\ x = \texttt{alloc}()\ \texttt{in}\ \texttt{let}\ b = \texttt{alloc}()\ \texttt{in}\ \texttt{set}_{rel}(x, 0);$$
$$\{\, \mathit{Uninit}(b) \ast \mathit{Rel}(x, \Phi) \ast \mathit{Init}(x) \ast \mathit{Acq}(x, \Phi) \,\}$$

$$\{\, \mathit{Uninit}(b) \ast \mathit{Rel}(x, \Phi) \,\}$$
$$\texttt{set}_{na}(b, 53);$$
$$\{\, b \overset{1}{\mapsto} 53 \ast \mathit{Rel}(x, \Phi) \,\}$$
$$\Rightarrow \{\, \Phi\ 1 \ast \mathit{Rel}(x, \Phi) \,\}$$
$$\texttt{set}_{rel}(x, 1)$$

$$\{\, \mathit{Init}(x) \ast \mathit{Acq}(x, \Phi) \,\}$$
$$\texttt{repeat}\ \texttt{get}_{acq}(x);$$
$$\{\, \exists v \neq 0,\ \Phi\ v \,\} \Rightarrow \{\, b \overset{1}{\mapsto} 53 \,\}$$
$$\texttt{let}\ n = \texttt{get}_{na}(b)\ \texttt{in}$$
$$\{\, b \overset{1}{\mapsto} 53 \ast \langle n = 53 \rangle \,\}$$

Multiple readers, multiple writers

Write permissions can be duplicated:

$$\mathit{Init}(\ell) = \mathit{Init}(\ell) \mathbin{\ast} \mathit{Init}(\ell) \qquad \mathit{Rel}(\ell, \Phi) = \mathit{Rel}(\ell, \Phi) \mathbin{\ast} \mathit{Rel}(\ell, \Phi)$$

Read permissions can be split:

$$\mathit{Acq}(\ell, \lambda v.\ \Phi_1\ v \mathbin{\ast} \Phi_2\ v) = \mathit{Acq}(\ell, \Phi_1) \mathbin{\ast} \mathit{Acq}(\ell, \Phi_2)$$

Example: one writer, two readers.

$$\begin{array}{l} \mathtt{set}_{na}(a, 13); \\ \mathtt{set}_{na}(b, 17); \\ \mathtt{set}_{rel}(x, 1); \end{array} \;\Big\|\; \begin{array}{l} \mathtt{repeat}\ \mathtt{get}_{acq}(x); \\ \{\, a \overset{1}{\mapsto} 13 \,\} \end{array} \;\Big\|\; \begin{array}{l} \mathtt{repeat}\ \mathtt{get}_{acq}(x); \\ \{\, b \overset{1}{\mapsto} 17 \,\} \end{array}$$

Compare And Swap

$$\{\,\texttt{emp}\,\}\ \texttt{alloc}()\ \{\,\lambda\ell.\ \mathit{Rel}(\ell,\Phi)\ast\mathit{RMWAcq}(\ell,\Phi)\,\}$$

RMWAcq permissions can be duplicated:

$$\mathit{RMWAcq}(\ell,\Phi)\ast\mathit{RMWAcq}(\ell,\Phi)=\mathit{RMWAcq}(\ell,\Phi)$$

The rule for $\mathit{CAS}_{X,rlx}$:

$$\frac{\begin{array}{c}P\Rightarrow \mathit{Init}(\ell)\ast\mathit{RMWAcq}(\ell,\Phi)\ast\texttt{true}\\ P\ast\Phi\ v\Rightarrow \mathit{Rel}(\ell,\Psi)\ast\Psi\ v'\ast R\ 1\\ P\Rightarrow R\ 0\\ X\in\{rel,rlx\}\Rightarrow\Phi\ v=\texttt{emp}\qquad X\in\{acq,rlx\}\Rightarrow\Psi\ v'=\texttt{emp}\end{array}}{\{\,P\,\}\ \texttt{CAS}_{X,rlx}(\ell,v,v')\ \{\,R\,\}}$$

A logic for relaxed accesses

Relaxed reads and writes

Intuition: like a release write or an acquire load, but without resource transfer.

$$\{\, \mathit{Acq}(\ell, \Phi) \,\}\ \mathtt{get}_{\mathit{rlx}}(\ell)\ \{\, \lambda v.\ \langle \Phi\ v \neq \mathtt{false} \rangle \,\}$$

$$\frac{\Phi\ v = \mathtt{emp}\ \text{(i.e. } \Phi\ v \text{ is pure and true)}}{\{\, \mathit{Rel}(\ell, \Phi) \,\}\ \mathtt{set}_{\mathit{rlx}}(\ell, v)\ \{\, \mathit{Rel}(\ell, \Phi) \,\}}$$

A modest application: control the set of all possible values for location ℓ, for instance $\Phi = \lambda v.\ \langle 0 \leq v \leq 10 \rangle$.

Relaxed accesses and values out of thin air

Vafeiadis & Narayan point out that these rules are incorrect for C/C++ 2011, since relaxed accesses are allowed to produce values out of thin air.

$$\begin{array}{l} \texttt{let } a = \texttt{get}_{rlx}(X) \texttt{ in} \\ \texttt{set}_{rlx}(Y, a) \end{array} \quad \Big\| \quad \begin{array}{l} \texttt{let } b = \texttt{get}_{rlx}(Y) \texttt{ in} \\ \texttt{set}_{rlx}(X, b) \end{array}$$

Starting with $X = Y = 0$, we can (according to the C/C++ 2011 standard) end with $X = Y = 1$.

According to Vafeiadis & Narayan's rules, $\Phi = \lambda v.\ \langle v = 0 \rangle$ is a correct invariant for X and for Y.

The problem with values out of thin air

A major risk: they break type safety and open security holes.

A theoretical risk: no known architecture or compiler exhibits “out of thin air” behaviors.

A specification problem: axiomatic definitions (using event structures) of C11-style memory models have a hard time distinguishing between

- behaviors involving values out of thin air;
- speculative behaviors (of the “load buffering” kind) that are correct.

A promising semantics

Kang et al (2017) describe an operational semantics for C/C++ 2011 atomics, of the (simplified) shape below.

1- Shared memory M = a set of write messages

A message is $\langle \ell : v \,@\, t \rangle$, representing the write of value v at location ℓ at timestamp t.
There is at most one message $\langle \ell : v \,@\, t \rangle$ for a given ℓ and a given t.

A promising semantics

Kang et al (2017) describe an operational semantics for C/C++ 2011 atomics, of the (simplified) shape below.

1- Shared memory M = a set of write messages

2- A process = a view V of the shared memory ...

A view = a function location $\to$ time at which the contents of this location was observed most recently.

Reading from location ℓ = observing a message $\langle \ell : v \mathbin{@} t \rangle$ with $t \geq V(\ell)$.

Writing v to location ℓ = sending a message $\langle \ell : v \mathbin{@} t \rangle$ with $t > V(\ell)$ a fresh timestamp.

In both cases, V is updated to $V[\ell \leftarrow t]$.

A promising semantics

Kang et al (2017) describe an operational semantics for C/C++ 2011 atomics, of the (simplified) shape below.

1- Shared memory M = a set of write messages

2- A process = a view V of the shared memory ...

3- ... plus a set P of promises.

A promise = a speculative write = a message already in the shared memory, but which still needs to be realized by an actual write later in the process execution.

A promising semantics

Kang et al (2017) describe an operational semantics for C/C++ 2011 atomics, of the (simplified) shape below.

1- Shared memory M = a set of write messages

2- A process = a view V of the shared memory …

3- …plus a set P of promises.

Invariant enforced at each reduction step: it is always possible to reduce so as to realize all promises. This prevents out-of-thin-air behaviors.

$$\begin{array}{ccccc} (c_1, V_1, P_1)/M_1 & \longrightarrow & (c_2, V_2, P_2)/M_2 & \longrightarrow & \cdots \\ \searrow^{*} & & \searrow^{*} & & \\ (\_, \_, \emptyset)/\_ & & (\_, \_, \emptyset)/\_ & & \end{array}$$

SLR: a separation logic for the promising semantics

(Svendsen *et al*, 2018.)

An extension of RSL with extra assertions:

$O(\ell, v, t)$ (generalizes $\mathit{Init}(\ell)$)

I observed value v in location ℓ at time t.

$W^{\pi}(\ell, X)$ (generalizes $\ell \overset{\pi}{\mapsto} v$)

I have permission π on location ℓ.

$X = \{(v_1, t_1), \ldots, (v_n, t_n)\}$ is a set of timestamped writes to this location.

If $\pi = 1$ (exclusive permission), X contains all the writes to ℓ ever performed.

If $\pi < 1$, X is a subset of these writes.

Some properties of assertion W

The assertion can be split:

$$W^{\pi_1+\pi_2}(\ell, X_1 \cup X_2) = W^{\pi_1}(\ell, X_1) \mathbin{\ast} W^{\pi_2}(\ell, X_2)$$

Writes are consistent (unique value for a given timestamp):

$$W^{\pi}(\ell, X) \mathbin{\ast} \langle (v, t) \in X \land (v', t') \in X \land v \neq v' \rangle \Rightarrow W^{\pi}(\ell, X) \mathbin{\ast} \langle t \neq t' \rangle$$

All writes are observed:

$$W^{\pi}(\ell, X) \mathbin{\ast} \langle (v, t) \in X \rangle \Rightarrow W^{\pi}(\ell, X) \mathbin{\ast} O(\ell, v, t)$$

The converse is true if the permission is exclusive:

$$W^{1}(\ell, X) \mathbin{\ast} O(\ell, v, t) \Rightarrow W^{1}(\ell, X) \mathbin{\ast} O(\ell, v, t) \mathbin{\ast} \langle (v, t) \in X \rangle$$

Reasoning about relaxed atomic accesses

$$\frac{\Phi\ v = \mathtt{emp}\ \text{(i.e. } \Phi\ v \text{ is pure and true)}}{\left\{\begin{array}{l} W^{\pi}(\ell, X) \\ \ast\ \mathit{Rel}(\ell, \Phi) \\ \ast\ O(\ell, \_, t) \end{array}\right\}\ \mathtt{set}_{rlx}(\ell, v)\ \left\{\begin{array}{l} \lambda\_.\ \exists t' > t, \\ \qquad W^{\pi}(\ell, \{(v, t')\} \cup X) \end{array}\right\}}$$

As in RSL, a relaxed write transfers no resources ($\Phi\ v = \mathtt{emp}$).

The write is reflected in assertion $W^{\pi}(\ell, X)$, which does not need to be exclusive. ($\pi < 1$ is allowed!)

$O(\ell, \_, t)$ proves that ℓ is initialized and gives a lower bound for the new timestamp t'.

Reasoning about relaxed atomic accesses

$$\left\{ \begin{array}{l} \mathit{Acq}(\ell, \Phi) \\ \ast\ \mathit{O}(\ell, \_, t) \end{array} \right\} \mathtt{get}_{\mathit{rlx}}(\ell) \left\{ \begin{array}{l} \lambda v.\ \exists t' \geq t, \\ \qquad \mathit{Acq}(\ell, \Phi) \ast \mathit{O}(\ell, v, t') \ast \nabla(\Phi\ v) \end{array} \right\}$$

A relaxed read of value v gives access to the pure part $\nabla(\Phi\ v)$ of the resource invariant $\Phi\ v$, and to a new observation $\mathit{O}(\ell, v, t')$.

If we own the full permission on ℓ, the value read is determined by the most recent write.

$$\left\{ \begin{array}{l} \mathit{Acq}(\ell, \Phi) \\ \ast\ \mathit{W}^1(\ell, X) \end{array} \right\} \mathtt{get}_{\mathit{rlx}}(\ell) \left\{ \begin{array}{l} \lambda v.\ \exists t, \langle (v, t) = \max(X) \rangle \\ \qquad \ast\ \mathit{Acq}(\ell, \Phi) \ast \mathit{W}^1(\ell, X) \ast \nabla(\Phi\ v) \end{array} \right\}$$

Summary

Summary

A realization: relaxed memory models such as those of Java or C/C++ are complicated and not fully understood yet.

> *There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies*
>
> (C. A. R. Hoare)

Summary

A realization: relaxed memory models such as those of Java or C/C++ are complicated and not fully understood yet.

A hope: it affects a handful of libraries only;
the bulk of parallel computation codes are still written using conventional synchronization primitives.

Summary

A realization: relaxed memory models such as those of Java or C/C++ are complicated and not fully understood yet.

A hope: it affects a handful of libraries only;
the bulk of parallel computation codes are still written using conventional synchronization primitives.

A most necessary tool: program logics!

- To abstract over some of the complexity of the memory model (cf. RSL, SLR).
- To combine reasoning in standard separation logic with reasoning specific to a given memory model.

References

References

An introduction to weakly-consistent memory models:

- S. V. Adve, H. J. Boehm, *Memory models: a case for rethinking parallel languages and hardware*, Comm. ACM, 2010.

The RSL and SLR program logics:

- V. Vafeiadis, C. Narayan, *Relaxed separation logic: a program logic for C11 concurrency*, OOPSLA 2013.
- K. Svendsen, J. Pichon-Pharabod, M. Doko, O. Lahav, V. Vafeiadis, *A Separation Logic for a Promising Semantics*, ESOP 2018.

The promising semantics for C/C++ 2011:

- J. Kang, C.-K. Hur, O. Lahav, V. Vafeiadis, D. Dreyer, *A promising semantics for relaxed-memory concurrency*, POPL 2017.