TU Munich

Summary:
Mathematizing C++11 Concurrency

Salomon Sickert (sickert (at) in.tum.de)

April 29, 2013

1 Introduction

Nowadays Multiprocessors became the standard choice for nearly every computing device, ranging from a smart-phone up to a super-computer. To accommodate the increasing demand in computing performance several optimisations on the hardware as well as on the software side are taken, e.g multi-level caches, out-of-order execution and multi-threading. Unfortunately the specification of the widely used language C++ left out concurrent programming and let third-parties, such as `pthreads` and `OpenMP`, provide the necessary tools. This approach is nowadays considered problematic, as the compiler may produce inefficient or even incorrect code. For example an analysis of the `pthreads` library can be found in [5].

To solve this issue the new C++11 standard introduces several concurrency idioms, most notably a threading class and a memory model clearly specifying, how access to shared variables is performed. As concurrency especially on relaxed memory is sometimes counter-intuitive and extremely complex, Batty et al. formalised parts of the concurrency specification and proved the correctness of a prototype implementation for atomics on the x86 platform. During their work they also discovered and fixed some ambiguities and flaws in the concurrency drafts. From this point onwards the language has built-in concurrency primitives, similar to Java.

2 Memory Access on Modern x86 Multiprocessors

As already mentioned modern multiprocessors use a variety of techniques to increase performance, for example write buffers to enable non-blocking write access to the memory. While these optimisations are invisible for a single-threaded program, the effects can be observed in multi-threaded code. For instance write buffering can have surprising effects. Assume the shared variables `x` and `y` are initialised with 0 and the following instructions are executed in parallel on two hardware threads.

Hardware Thread 1	Hardware Thread 2
`MOV [x] ← 1`	`MOV [y] ← 1`
`MOV r1 ← [y]`	`MOV r2 ← [x]`

Write buffering allows the processor to end up in the final state (T1:r1 = 0) ∧ (T2:r2 = 0). This is called *relaxed memory model*, meaning different threads may observe a *subtly* different memory. As many algorithms are designed with a stronger memory model in mind, which forbids this final state, the programmer needs to proceed with caution. For example the classic Dekker's Mutex Algorithm [3] allows both threads to enter the critical section at the same time if a relaxed memory model is used.

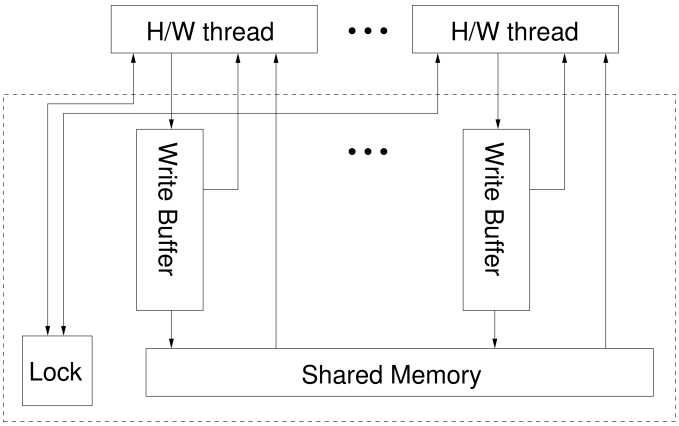


Figure 1: x86-TSO block diagram, taken from [8]

In order to restore sequential consistency **LOCK**'ed instructions or fencing (memory barriers) can be used. However, flushing caches is costly in terms of cycles and the overall performance decreases. Additionally one should keep in mind, that this problem is not limited to the x86 architecture, as some non-x86 architectures exhibit even weaker models, e.g ARM.

x86-TSO. Unfortunately Intel and AMD only publish a prose specification of their processor architectures, which are missing a precise definition of the memory model according to [8] and [6]. Hence Sewell et al. created the x86-TSO (Total Store Order) machine model, which is a "A Rigorous and Usable Programmers's Model for x86 Multiprocessors" [8].

The abstract machine - depicted in figure 1 - is composed of n hardware threads, each of them executing a separate instruction stream, and a storage system, which contains a shared memory, a global lock and for each thread a private write buffer. For every instruction, such as `ADD` and `INC`, there exists a locked version, such as `LOCK ADD` and `LOCK INC`, which limit read operations to the executing H/W thread.

The write buffers are organized as FIFO's and can propagate buffered writes to the shared memory at any time except another thread holds the global lock. Furthermore a thread accessing a memory address reads the most recent value from its write buffer, if there is one, and otherwise from the shared memory. By issuing a `MFENCE` the thread flushes the write buffer.

In order to execute LOCK'd instructions, the thread has to obtain the global lock, execute the instruction, flush its write buffer and finally relinquish the lock. While a thread holds the lock, all other threads are unable to perform read operations.

```cpp
#include <vector>
#include <iostream>
#include <thread>
#include <atomic>

std::atomic<int> cnt = ATOMIC_VAR_INIT(0);

void f()
{
    for (int n = 0; n < 1000; ++n) {
        /* Do something */
        cnt.fetch_add(1, std::memory_order_relaxed);
    }
}

int main()
{
    std::vector<std::thread> v;
    for (int n = 0; n < 10; ++n) {
        v.emplace_back(f);
    }
    for (auto& t : v) {
        t.join();
    }
    std::cout << "Final counter value is " << cnt;
}
```

Listing 1: Atomics and Threads, based on [1]

3 C++11 Concurrency

To address the mentioned issues the C++11 language extension introduces several concurrency idioms to the standard library, such as atomics, mutexes and threads. The atomic type [2] is the most interesting of these new additions, as one can specify the memory order, which should be used for this particular atomic memory operation. The short example in listing 1 illustrates the use of these new types. Two prominent memory orders are the relaxed and the sequentially consistent ordering.

Relaxed Ordering. This ordering does not provide any synchronisation, the only guarantees are the atomicity of the operation and that the modification order is respected. This is the weakest available ordering. In the context of the x86-TSO model a relaxed load and store can be implemented[1] using `MOV`. (See also [4])

Sequentially-Consistent Ordering. All threads using this ordering are synchronized and see the same order of memory accesses. Furthermore no write operation of a thread using a write operation to the atomic, can be reordered after the write. Accordingly no read operation of another thread can be reordered before the atomic read. For the x86-TSO model one can implement a sequentially consistent load with `MFENCE, MOV` and a store with `MOV, MFENCE`.

4 C++11 Memory Model

In the context of this summary only a small part of the C++ memory model, introduced in [4], will be presented. The central questions is: Given a program p, what are all possible executions? As different threads can observe a *subtly* different memory, "the semantic cannot be expressed in terms of changes to a monolithic memory" [4]. Hence the model is expressed in terms of memory actions and relations. In the first step a set of all candidate executions, called *pre-executions*, is computed. A candidate execution consists of X_{opsem} and $X_{witness}$, which will be introduced as we go along with the example from listing 2. In the second step the existence of undefined behaviour in at least one execution is tested. In the presence NONE is returned and in the absence the set of all pre-executions. Formally:

cpp_memory_model *opsem* (p : program) =
  let pre_executions = $\{(X_{opsem}, X_{witness}) \mid opsem\ p\ X_{opsem} \wedge$ consistent_execution $X_{opsem}\ X_{witness}\}$ **in**
  if $\exists X \in$ pre_executions. undefined_behaviour X
  then NONE
  else SOME pre_executions

4.1 X_{opsem}

X_{opsem} is defined by the "syntactic structure of the source code and the path of the control flow"[4] and hence is independent from the memory model. Loosely speaking it describes the structure of the program and is composed of a set of actions, a set of thread ids and a simple location typing function. Furthermore it contains the following binary relations:

- $\xrightarrow{sequenced-before}$ (sb)
- $\xrightarrow{additional-synchronized-with}$ (asw)
- $\xrightarrow{data-depenency}$ (dd)

sequenced-before is defined by the evaluation order of the statements in the source code and the *additional-synchronized-with* models synchronisation, such as thread creation and joins. As *data-depenency* is related to release and consume atomics, it will not be discussed in the context of this summary.

Actions are different memory operations and can be reads, writes, read-modify-writes, locks, unlocks and fences. Moreover there are qualified by an *aid* - action id - and a *tid*. Furthermore some of them have a location l and a value v. Additionally there are different variants, such as non-atomics (na) and atomics with different memory order (mo). In the example only non-atomic read and write operations are used.

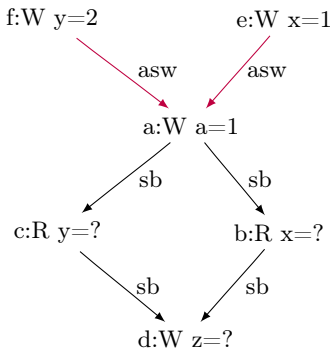
action =
  aid, *tid*: $\mathrm{R}_{na}\ l = v$
| *aid*, *tid*: $\mathrm{W}_{na}\ l = v$
| *aid*, *tid*: $\mathrm{R}_{mo}\ l = v$
| *aid*, *tid*: $\mathrm{W}_{mo}\ l = v$
| *aid*, *tid*: $\mathrm{RMW}_{mo}\ l = v_1/v_2$
| *aid*, *tid*: L l
| *aid*, *tid*: U l
| *aid*, *tid*: F_{mo}

Consider the short code example in listing 2, which is a fragment of C/C++. To simplify the example parallel composition is used instead of the regular thread creation and joining, which causes additional memory operations and is expressed by: $\{\{\{ \ldots \mid\mid\mid \ldots \}\}\}$.

As the statements $x = 1$ and $y = 2$ are executed in two separate threads, the short example has in total three threads. The statement $x = 1$ gives rise to the action $e, 3 : W_{na}\ x = 1$ and similar the comparison $x == y$ is related to the two non-atomic read actions: $b, 1 : R_{na} x =?$. (Value replaced by ? as the value of x is unknown without a consistent reads-from relation). For readability from now on the *thread ids* and the

[1] Provided the type fits into a single register.

```
1  int main() {
2    int x, y, z, a;
3    {{{  x = 1;
4    |||  y = 2;
5    }}};
6    a = 1;
7    z = (x == y);
8    return 0;
9  }
```



Listing 2: CPPMEM Input

na subscript are elided, as they are clear from the context. By further inspecting the code one can infer the two edges of the *additional-synchronized-with* caused by the implicit join of the parallel composition and the five edges of the *sequenced-before* (the transitive edge is omitted in the picture). Note that the read operations in line 7 are not ordered by sb, as C++ does not specify the evaluation order, in fact the values could also be computed in parallel.

4.2 $X_{witness}$

While there exists exactly one X_{opsem} for every program p, the different ways of executing a program is modelled by different $X_{witness}$'es. It is composed of three binary relations:

- $\xrightarrow{reads-from}$ (rf)
- $\xrightarrow{sequential-consistency}$ (sc)
- $\xrightarrow{modification-order}$ (mo)

The *reads-from* relation contains edges from writes to reads. While the *sequential-consistency* relation totally orders all mutex and sequentially consistent actions, the *modification-order* relation total orders all atomic writes at a specific memory location. As the example only contains non-atomic variable access, the second and the third relation are empty.

In order to add $X = (X_{opsem}, (rf, sc, mo))$ to the set of *pre-executions*, one has to find a consistent *reads-from* relation.

consistent_execution =
  well_formed_threads (+)
∧ consistent_locks (*)
∧ consistent_inter_thread_happens_before (§)
∧ consistent_sc_order (*)
∧ consistent_modification_order (*)
∧ well_formed_reads_from_mapping
∧ consistent_reads_from_mapping

Predicates marked with a (*) are irrelevant to the example, as they refer to atomic actions. Furthermore predicates with a (+) ensure sanity properties and are uninteresting. Additionally the third predicate can be discarded, as by theorem 1 from [4], §2.10 it is redundant, because no consume operations are contained in the example.

well_formed_reads_from_mapping =
  $\forall a \xrightarrow{rf} b.$ same_location $a\ b \wedge$
  is_write $a \wedge$ is_read $b \wedge$
  value_read_by $b =$ value_written_by $a \wedge$
  $\forall a' \xrightarrow{rf} b.(a = a')$

As the well-formedness predicate constrains *reads-from* to relate only a write and read action with the same location, rf is upper-bounded by $\{(e,b),(f,c)\}$ in this example. Additionally the relation has to respect visible side effects, which are defined using the *happens-before* relation, which partially orders all actions. In this example the relation can be simplified to:

$$\xrightarrow{happens-before} = (\xrightarrow{sb} \cup \xrightarrow{asw})^+$$

$a \xrightarrow{visible-side-effect} b =$
  $a \xrightarrow{happens-before} b \wedge$
  is_write $a \wedge$ is_read $b \wedge$ same_location $a\ b \wedge$
  $\neg(\exists c.(c \neq a) \wedge (c \neq b) \wedge$ is_write $c \wedge$ same_location $c\ b \wedge$
    $a \xrightarrow{happens-before} c \xrightarrow{happens-before} b)$

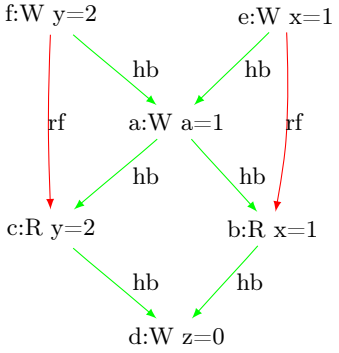
In the context of the example: $f \xrightarrow{vse} c$ and $e \xrightarrow{vse} b$.

consistent_reads_from_mapping =
  $(\forall b.$(is_read $b \wedge$ is_at_non_atomic_location $b) \implies$)
    (**if** $(\exists a_{vse}.a_{vse} \xrightarrow{visible-side-effect} b)$
    then $(\exists a_{vse}.a_{vse} \xrightarrow{visible-side-effect} b \wedge a_{vse} \xrightarrow{rf} b)$
    else $\neg(\exists a.a \xrightarrow{rf} b)) \wedge \ldots$

As all reads in the program are non-atomic and every read is covered by *vse*, all consistent *reads-from* are lower bounded by *vse*.

$$\{(e,b),(f,c)\} = vse \subseteq rf \subseteq \{(e,b),(f,c)\}$$

Hence there exists only one *reads-from* relation and there is only one candidate execution.



4.3 Undefined Behaviour

Finally *pre-executions* is checked for undefined behaviour. In the context of this example there are three different types of undefined behaviour relevant.

Indeterminate Reads. By inspecting the graph of the candidate execution, one can see that both read actions are satisfied by a previous write action. Thus the program is free of indeterminate reads.

indeterminate_reads = $\{b.$is_read $b \wedge \neg(\exists a.a \xrightarrow{rf} b)\}$

Unsequenced Races. An unsequenced race occurs, if there are two actions in the same thread unrelated by *sb* accessing the same location and at least one is a write, e.g. $x == (x = 2)$. While there are two unsequenced actions b and c, both of them are reads and thus race-free.

unsequenced_races = $\{(a,b).(a \neq b) \wedge$ same_thread $a\ b \wedge$
  same_location $a\ b \wedge$ (is_write $a \vee$ is_write $b) \wedge$
  $\neg(a \xrightarrow{sequenced-before} b \vee b \xrightarrow{sequenced-before} a)\}$

Data Races. Similar to unsequenced races data races occur, if two actions from different threads unrelated by hb are accessing the same location and at least one is a write. Again, in our example there are two concurrent writes (e and f), although two different locations. Hence there is no data-race.

$$\begin{aligned}&\text{data\_races} = \{(a,b).(a \neq b) \wedge \neg\text{same\_thread } a\ b \wedge\\ &\quad \text{same\_location } a\ b\ \wedge (\text{is\_write } a \vee \text{is\_write } b) \wedge\\ &\quad \neg(\text{is\_atomic\_action } a \wedge \text{is\_atomic\_action } b) \wedge\\ &\quad \neg(a \xrightarrow{happens-before} b \vee b \xrightarrow{happens-before} a)\}\end{aligned}$$

As the candidate executions are free of undefined behaviour the *pre-executions* set represents all possible executions of the program p.

5 Applications of the Model

Batty et al. used their formal model to explore different aspects of the C++ concurrency. During this research they discovered besides other issues, that memory_order_seq_cst was in fact not sequentially consistent in one of the drafts. Furthermore they proved the correctness of the compilation strategy for the x86-TSO model. In another study they extend the research to other architectures such as the POWER in [7]. These efforts can be seen as an important step towards a verified C++ compiler.

References

[1] std::memory_order - cppreference.com, . URL `http://en.cppreference.com/w/cpp/atomic/memory\_order`. Accessed 9.4.2013.

[2] C++ atomic types and operations, . URL `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2427.html`. Accessed 9.4.2013.

[3] Dekker's algorithm. URL `http://www.cs.utexas.edu/users/EWD/transcriptions/EWD01xx/EWD123.html`. Accessed 9.4.2013.

[4] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing c++ concurrency. *SIGPLAN Not.*, 46(1):55–66, January 2011. ISSN 0362-1340. doi: 10.1145/1925844.1926394. URL `http://doi.acm.org/10.1145/1925844.1926394`.

[5] Hans-J. Boehm. Threads cannot be implemented as a library. *SIGPLAN Not.*, 40(6):261–268, June 2005. ISSN 0362-1340. doi: 10.1145/1064978.1065042. URL `http://doi.acm.org/10.1145/1064978.1065042`.

[6] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO (extended version). Technical Report UCAM-CL-TR-745, University of Cambridge, Computer Laboratory, March 2009. URL `http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-745.pdf`.

[7] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. Synchronising c/c++ and power. *SIGPLAN Not.*, 47(6):311–322, June 2012. ISSN 0362-1340. doi: 10.1145/2345156.2254102. URL `http://doi.acm.org/10.1145/2345156.2254102`.

[8] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, July 2010. ISSN 0001-0782. doi: 10.1145/1785414.1785443. URL `http://doi.acm.org/10.1145/1785414.1785443`.