Concurrent Systems
Nebenläufige Systeme

V. Elementary Operations

Wolfgang Schröder-Preikschat

November 11, 2015

Outline

Preface

Primitive Instructions
  Atomic Operations
  Equivalence

Memory Models
  Properties

Summary

Subject Matter

- discussion on abstract concepts as to elementary operations at instruction structure set architecture level
  - atomic load/store of a naturally aligned machine word
  - atomic read-modify-write of complex machine instructions
- impartation of knowledge on memory models that are relevant to multi-threading on multi/many-core (multi-) processors
  - atomicity, visibility, and ordering of memory operations against the background of UMA, NUMA, and (partly) COMA architectures
  - ordering enforcing hardware such as memory barriers or fences, resp., allowing one to pattern sequential, relaxed, and weak data consistency
- excursion into practice of hardware features that are of importance for the implementation of any synchronisation algorithm
Memory-Operation Semantics

- of particular interest (at this point) are **shared-memory operations**
  - commonality is the opportunity, at least, for **indivisible execution**
- note, all memory operations are also divisible in the following respect:
  - **sub-operation**
    - processors are word-oriented, but memory is byte-oriented
    - with **word size** as a multiple of **byte size**, e.g. \(4 \times 8\) bits
    - thus, loads/stores will operate on a **sequence of bytes**
  - **sub-step**
    - processors perform a **fetch-execute-cycle** to run programs
    - \(n\)-address machines mean \(n\)-operand instructions, \(n \geq 2\)
    - thus, execution requires a **sequence of loads/stores**

\[^1\] In general \(n \geq 0\), but only for \(n \geq 2\) becomes the problem apparent.

Load/Store I

C—Level 5

```c
#include <stdint.h>

static int64_t label;

int64_t get_label () {
    return label;
}

void set_label(int64_t value) {
    label = value;
}
```

- in logical respect any of these single statements is indivisible, atomic
- lines 6 conceals a load and line 10 conceals a store operation
- each case forms an ELOP of the **abstract processor** "C"
- in physical respect these statements are **conditionally atomic**, only
  - a matter of optimisation options, the CPU, and alignment restrictions

Load/Store II

ASM—Level 4

```asm
gcc -m32...
get_label:
    movl label , %eax
    movl label +4 , %edx
    ret

set_label:
    movl 4(%esp) , %eax
    movl 8(%esp) , %ecx
    movl %ecx , label +4
    movl %eax , label
    ret
```

```asm
gcc -m64...
get_label:
    movq label (%rip) , %rax
    ret

set_label:
    movq %rdi , label (%rip)
    ret
```

- actions 2-3 and 9-10 are divisible
- any of these 8 **mov** instructions is **conditionally indivisible**

- beware of the processor architecture or the data alignment, resp.
  - usually, memory-word loads/stores are indivisible if "word" corresponds to the smallest addressable unit of main memory: **byte**, nowadays
  - on some architectures (e.g., x86) they are indivisible too if the address of the memory operand is **naturally aligned**
Execution cycle of a machine instruction that involves the ALU consists of the following individual operation steps:

1. load input operands (acc. operation code or addressing mode, resp.)
2. compute result (acc. operation code)
3. store output operand (acc. operation code or addressing mode, resp.)

Steps (i) and (iii) require the bus in case of memory-sensitive operations—reusable hardware resource, shareable, allocated per (load/store) step.

Typical compound action at instruction set architecture (ISA) level—memory-sensitive only for a complex instruction set computer (CISC). In a multiprocessor case, the whole cycle is divisible (non-atomic).

Indivisibility requires a bus lock for the duration of the whole cycle:

1. an atomic RMW instruction that implicitly performs the lock or
2. a lock prefix that makes the adjacent normal RMW instruction atomic.

Test & Set II

**Swap**

The original copy (IBM System/370) has swapping characteristic.

\[\text{swap}(x, y), \text{with } x = \ast ref_0 \text{ and } y = 11111111_2[0]\]

For a contemporary processor (x86), this translates into the following:

```c
int tas(any_t *ref) {
    return TAS(ref);
}
```

Whereby (using GCC atomic built-in functions):

```c
#define TAS(ref) __sync_lock_test_and_set(ref, 1)
```

Note that xchg interlocks against simultaneous main memory accesses.

Beware of the unconditional store carried out by both TS and xchg:

- This semantic has a deleterious effect for cache-coherent processors.
- The cache line holding the main memory operand is always invalidated

\[\leftrightarrow\text{ dedicated hardware implementation (p.12) or mapping to CAS (p.13)}\]

Test & Set III

**DPRAM**

A kind of random access memory (RAM) that supports simultaneous load and store operations from two directions.

The interlock is conducted by a “DPRAM monitor” that, e.g. [18]:

- Records the processor that issued the TAS and acquired access.
- Notifies processors that, at a time, issue a TAS simultaneously—signalling BUSY interrupt, forcing the receiving processor into busy waiting.
- Performs the test and then, if and only if the test succeeds:
  - Sets the memory location to the value given by the owning processor and
  - Releases access to that memory location.

This scheme translates into a conditional store as follows:

```c
word tas(word *ref) {
    word aux;
    atomic { if ((aux = *ref) == 0) *ref = 1; }
    return aux;
}
```

Test & Set I

**TAS**

Definition (TS, acc. IBM System/370)

The leftmost bit (bit position 0) of the byte located at the second-operand address is used to set the condition code, and then the entire addressed byte is set to all ones. [8, p. 144]

The operation effectively does an unconditional store in main memory.

- The byte in storage is set to all ones as it is fetched for the testing of bit position 0. [8, p. 144]

In terms of main memory significance, this translates into the following:

```c
bool tas(byte *ref) {
    atomic { bool aux = *ref & 0x1; *ref = 0x11111111; }
    return aux;
}
```

Note that TS interlocks against simultaneous main memory accesses.

A similar effect has ldstub of SPARC V9.
Compare & Swap I  

Definition (CS, acc. IBM System/370)

The first and second operands are compared. If they are equal, the third operand is stored in the second-operand location. If they are unequal, the second operand is loaded into the first-operand location. [8, p. 123]

- The operation effectively performs a conditional store in main memory
- The first and third operands [each are] occupying a general register. The second operand is a word in main storage. [8, p. 123]
- In terms of main memory significance, this translates into the following:

```c
atomic word cas(register old, word *ref, register new) {
    word aux;
    return aux = (*ref == old) ? (*ref = new) : (old = *ref);
}
```

- With the actual parameters `old` and `new` being kept in general registers
- Note that CS interlocks against simultaneous main memory accesses
- The dimension of the reservation depends on the hardware implementation. It may be exactly the effective address or a larger address range around.

Load-Linked/Store-Conditional I  

Definition

Paired instructions to form a flow of actions without any guarantee of indivisibility but that it succeeds only in case of indivisibility operation.

- Originated in the MIPS II or R6000, resp., RISC architecture [9]:
  - LL loads a word from the specified effective memory address
  - Makes a reservation on that very address (range)\(^5\)
  - SC checks for a reservation on the specified effective memory address\(^5\)
    - If the reservation persists, stores the specified word at that address
    - Delivers the result of the reservation check
- Reasons for cancellation of a persisting address (range) reservation:
  - Successful execution of SC—hoped for, normally
  - Execution of LL by another processor applying the same address (range)
  - An exception (trap/interrupt) on the processor holding the reservation
  - LL and SC interlock against simultaneous main memory accesses

\(^5\)The dimension of the reservation depends on the hardware implementation. It may be exactly the effective address or a larger address range around.

Compare & Swap II  

Definition (ABA, also A-B-A)

The ABA problem is a false positive execution of a CAS-based speculation on a shared location \(L_i\). [2, p. 186]

- When the successful execution of a CAS instruction indicates:
  - That the two operands subject to comparison are equal and, thus, purport the presence of a certain condition (positive)
  - But the condition is not in fact present (false)
- Assuming that processes \(P_1\) and \(P_2\) simultaneously access location \(L_i\)
  - Value \(A\) read by \(P_1\) from \(L_i\) be a sign of a dedicated global state \(S_1\), but \(P_1\) will be delayed before being able to commit a new value to \(L_i\)
  - Meanwhile \(P_2\) changes the value of \(L_i\) to \(B\) and then back to \(A\), defining a new global state \(S_2 \neq S_1\)
  - \(P_1\) resumes, observes that the value of \(L_i\) equals \(A\) and, thus, acts on the assumption that the global state must be \(S_1\)—which is no longer true
- Severity of false positive execution depends on the problem (cf. p. 36)

Load-Linked/Store-Conditional II  

Definition

Use of LL/SC to recreate TAS and CAS:

- In case of TAS, a boolean variable is conditionally set true
  ```c
  int tas(long *ref) {
    return (LL(ref) == 0) && SC(ref, 1);
  }
  ```
  - In case of CAS, a memory word is conditionally overwritten
  ```c
  int cas(long *ref, long old, long new) {
    return (LL(ref) == old) && SC(ref, new);
  }
  ```
- Note that this implementation of CAS is free from the ABA problem:
  - \(P_1\) shares location \(ref\) with \(P_2\), established reservation \(ref_p\) by LL
    - Gets delayed for some reason, thus has not yet executed SC
  - \(P_2\) overlaps \(P_1\), establishes reservation \(ref_p\) and, thus, cancels \(ref_p\)
    - Successfully executes SC ⇒ CAS succeeds
  - \(P_1\) resumes ⇒ SC will fail because reservation \(ref_p\) is invalid
    - Returns failure of CAS ⇒ rolls back, backs up, and retries CAS...
**Fetch & Add**

**Definition (acc. [6, p. 17])**

A value-returning instruction that operates on a global (i.e., shared) variable \( G \) and a local variable \( L \).

- an atomic RMW instruction, inspired by “Replace Add” [3, p. 6]
  - prefix (FAA) or postfix (AAF) form, as to when fetch becomes effective
    - prefix – save the old value of \( G \) for return, then add \( L \) to \( G \)
    - postfix – add \( L \) to \( G \), then return the new value of \( G \)
  - whereby (cf. p. 39):
    \[
    FAA(G, L) \equiv AAF(G, L) - L \quad \text{and} \quad AAF(G, L) \equiv FAA(G, L) + L
    \]

- transferable to any associative binary operation \( fetch-and-\Phi \)
  - but for noninvertible operations the prefix form is considered more general
  - be \( \Phi = \text{max} \) (i.e., \( X \)): only \( XAF(G, L) \equiv \text{max}(FAX(G, L), L) \) (cf. p. 40)

**Equality of Atomic Operations**

**Definition (Consensus Number)**

The consensus number for \( X \) is the largest \( n \) for which \( X \) solves \( n \)-process consensus. If no largest \( n \) exists, the consensus number is said to be infinite. [7, p. 130]

- \( n \) processes need to interact to achieve agreement on a single data value
  - note that only 1-process consensus requires no interaction
- consensus numbers of the elementary operations considered:
  - \( \infty \) ■ compare-and-swap, load-linked/store-conditional
  - 2 ■ test-and-set, swap, fetch-and-add
  - 1 ■ atomic read, atomic write
- key point is the progress guarantee a certain operation has to give
  - for wait-freedom [7], the operation must have consensus number \( n = \infty \)
  - in that case, every action has guarantee to complete in finite steps/time

**Properties Relevant to Multi-Threading**

- fundamental characteristics that are of particular importance for the implementation of any synchronisation algorithm:
  - atomicity ■ as to how certain machine instructions are executed
    - differentiates in RISC and CISC machines
  - visibility ■ as to when memory-cell changes are observable
    - concerns delays in sensing the most recent memory-word write
  - ordering ■ as to how memory operations appear to be performed
    - stands for a variant of out-of-order execution
  - these properties are linked with each other, are mutual prerequisites
  - atomicity applies to all other—and to a single machine instruction, only
  - visibility depends on the memory architecture, may cause “jitter”
  - ordering comprises multiple machine instructions, may cause “fencing”
  - as to the level of abstraction, they must all be considered together
  - this is especially true for the operating-system machine level (i.e., level 3)
Atomicity

common are two classes of memory-sensitive operations (cf. p. 25):

L/S  
- atomic load (L) or store (S), resp., as single action
- granularity is the **machine word**, i.e., a multiple of a byte
- with **word-alignment** constraint on the operand address, usually
  - only word-aligned accesses will be carried out indivisibly

RMW  
- atomic read (R), modify (M), and write (W) as single action
- common for CISC and, there, for **two-address machines**
- uncommon for RISC, which is characteristic of load/store principle
- single- or double-word cycles for 32- or 64-bit architectures, resp.
  - “double” means “physically consecutive” or “logically interrelated”
  - i.e.: CDS or cmpxchg8b/cmpxchg16b compared to DCAS or CAS2

processes cannot observe any intermediate steps and partial effects

here, only in matters of a single (L/S or RMW) machine instruction

that is to say, the ISA-level action appears **indefinable** and **irreducible**

as a consequence, the instruction will be performed entirely or not all

- with the latter meaning **failure indication** (TAS, CAS, SC)

Visibility

When other interacting processes will notice the changes made by the current process, and whether they will notice them at all.

- depends on the **memory architecture** and behaviour of read or write operations to the same memory location
  - UMA  **uniform memory architecture**  ~ the same access time
  - each address is assigned a fixed home in the global address space
  - no processor uses private (local) memory besides shared memory
  - NUMA  **non-uniform memory architecture**  ~ different access times
  - each address is assigned a fixed home in the global address space
  - each processor (“NUMA node”) uses private (local) memory, too
  - COMA  **cache-only memory architecture**  ~ different access times
  - each address is assigned a fixed home in the global address space
  - each processor uses private (local) memory, only

orthogonal with it is the **consistency** aspect as to shared information stored in multiple local **caches**

- **cache-coherent (cc)** v. **non-cache-coherent (ncc)** memory architecture

Ordering

What memory re-orderings are possible for a process, relatively to the order as specified by its program.

- to improve performance, memory-sensitive machine instructions are not executed in the order originally specified by the program
  - on the one hand, the compiler reorders (L3) instructions\(^6\) before run-time
  - on the other hand, the CPU reorders (L2) instructions\(^6\) at run-time
    - it is this aspect of **dynamic ordering** that is of relevance in the following
  - mainly, dynamic ordering is an issue of non-blocking synchronisation
    - as blocking synchronisation implicitly can take care of “fencing” proper
      - depending on the kind of critical section and type of data dependency
    - but, critical section **per se** is no guarantee for memory ordering (cf. p. 25)
  - ordering ensuring needs special instructions: **memory barrier/fence**

\(^6\)According to the actual level of abstraction: operating-system machine (L3) or instruction set architecture (L2) level. See also [10] or [17, p. 34].
assessing that the following function is executed by a single processor, but the global variables are then read by at least one more processor:

```c
int a = 1, b = 2;
void ab_set() {
  a = 3;
  b = 4;
}
```

what values of $a$ and $b$ do other processors see once line 6 has been reached by one processor?

assuming that the next function is executed directly afterwards to the former one just discussed, but by a different processor:

```c
void ab_get(int ab[2]) {
  ab[0] = b;
  ab[1] = a;
}
```

what values of $a$ and $b$ are delivered?

- line 8 may read the new value of $b$ while line 9 may read the old value of $a$

although the assignment to $a$ (line 4) was instructed previous to the one of $b$

Memory Barriers

- `ld_a LoadLoad st_b StoreStore` ensures that $a$ is read before $b$ is accessed
- `st_a StoreStore st_b StoreLoad` ensures that $a$ is visible before $b$ is flushed
- `ld_a LoadStore st_b StoreLoad` ensures that $a$ is read before $b$ is flushed
- `st_a StoreLoad st_b StoreLoad` ensures that $a$ is visible before $b$ is accessed
- write to same location by another processor

CAS and LL/SC typically include a StoreLoad barrier on the target

i.e., not only a general-purpose but also the most expensive fence

out-of-order processors that can bypass loads

write to same location by another processor

Consistency Models

- data consistency as close as possible to sequential processes or with optimisation margins for high-latency memory
  - sequential: processors see writes on the same target in the same order but the order may appear different for an "external observer" two requirements: program order and write atomicity
  - relaxed: in terms of the constraints defined by sequential consistency as to (i) program order, (ii) write atomicity, or (iii) both:
    - write to read, write to write, read to read and read to write
    - read other's, write early
    - read own, write early
  - weak: pertaining to (i) different or (ii) same memory locations
    - "limited to hardware-recognized synchronizing variables" yet weaker tendencies: release and entry consistency implemented by operating system machine level programs usually not provided by the instruction set architecture level
  - state of the art processors provide relaxed or weak consistency models

state of the art processors provide relaxed or weak consistency models

weaker than "strict consistency" that requires a read from a memory location to return the value of the most recent write.
Résumé

- elementary operations at instruction structure set architecture level
  - atomic load/store of a naturally aligned machine (double-) word
  - atomic read-modify-write of complex machine instructions
    - TAS, CAS and FAA or FAΦ, resp., for CISC and LL/SC for RISC
  - equality of atomic operations as to their consensus number
- memory-access properties that are relevant to multi-threading
  - atomicity, visibility, and ordering of memory operations
- memory architectures of type UMA, NUMA, and COMA
- dynamic ordering at instruction set architecture level
- memory barriers or fences, resp., to enforce ordering properly
- sequential, relaxed, and weak data consistency
- hardware features that are of importance for the implementation of any synchronisation algorithm
  - including but not limited to non-blocking synchronisation, especially

Reference List I


Reference List II


Reference List III


Unconditional Store: Workaround

- "textbook semantics" of TAS has a deleterious effect for the cache:
  1. word tas(word *ref) {
     2. atomic { word aux = *ref; *ref = 1; }
     3. return aux;
     4. }
  - same is true when using the GCC atomic built-in function (x86, cf. p11):
     5. #define TAS(ref) __sync_lock_test_and_set(ref, 1)
  - use of CAS, with #define CAS __sync_bool_compare_and_swap
    6. int tas(long *ref) {
       7. return CAS(ref, 0, 1);
    }
  - worst-case overhead of five instructions (cf. p11)
  - pays off, depending on processor and cache architecture

ABA Exemplified

- given a LIFO list (i.e., stack) of following structure: head \(A \ldots\)
  - with head stored at location \(L\), shared by processes \(P_1\) and \(P_2\)
  - push (cf. [16, p.11]) and pull adding or removing, resp., list items:
    1. chain_t *cas_pull(stack_t *this) {
       2. chain_t *node;
       3. do if ((node = this->head.link) == 0) break;
       4. while (!CAS(&this->head.link, node, node->link));
       5. return node;
    }
- assuming that the following sequence of actions will take place:
  - reads head item \(A\) followed by \(B\) on the list, gets delayed at line 4
  - remembers \(node = A\), but has not yet done CAS: \(head \downarrow A \uparrow B \downarrow C\)
  - pulls head item \(A\) from the list: \(head \downarrow B \uparrow C\)
  - pulls head item \(B\) from the list: \(head \downarrow C\)
  - pushes item \(A\) back to the list, now followed by \(C\): \(head \downarrow A \uparrow C\)
  - resumes, CAS realises \(head = A\) (followed by \(B\)): \(head \downarrow B \uparrow \circ\)
  - list state \(head \downarrow A \uparrow C\) as left behind by \(P_2\) is lost...
ABA Design Risk Reduction

- prevalent approach is to add a **change number** to the "control word" [8, p. 125], i.e., to practice some kind of **versioning**
- this number increments at each CAS attempt on the control word
- appropriate techniques depend on the change-number parameters
  - a. the values margin has a whole word size available
     - both the control and change-number word must be updated, indivisibly
     - compare double and swap (CDS, [8, p. 124]) of two consecutive words
     - double compare and swap (DCAS, also CAS2 [14, p. 4-66]) of any two words
  - b. the values margin utilizes fully unused bits in the control word itself
     - CAS facilitates indivisible updates of control word including change number
     - workaround, especially suitable for handling aligned data-structure **pointers**
     - gimmick is in data-structure padding for an object size of a power of two

[a] an object size of $2^n$ bytes then gives $n - 1$ low-order bits always 0
[a] these $n - 1$ low-order bits then will be used as a **change-number tag**
[a] for **pointer operations**, the change-number tag is temporary neutralised

but the ABA problem never disappears, it only gets more improbable

---

**FAA Exemplified**

GCC Atomic Built-in Functions, x86

```c
#define FAA __sync_fetch_and_add
int faa(int *p, int v) {
    return FAA(p, v);
}

#define AAF __sync_add_and_fetch
int aaf(int *p, int v) {
    return AAF(p, v);
}
```

Linguistic Devices for LL/SC

- as GCC does not provide atomic built-in functions for this case:
  - **INLINE**
  ```c
  long LL(long *ref) {
      long aux;
      asm volatile(
          "lwax %0, 0, %1", "=r" (aux)
      :
      : "r" (ref)
      :
      "lwarx %0, 0, %1"
      :
      "mwcr %0"
      :
      "="r" (aux)
      :
      "="r" (ccr)
      :
      "r" (ref), "r" (val)
      );
      return aux;
  }
  return ccr & 0x2;
  }
  ```

- with "#define INLINE extern inline" for GCC to ensure that
  stand-alone object code is never emitted for in-line functions

Noninvertible Operation

- **fetch-and-Φ**, with Φ = max

- safe-load of global variable $G$ and conditional-store of $\text{max}(G, L)$ at $G$
  ```c
  word fax(word *ref, word val) {
    word aux;
    atomic { if ((aux = *ref) < val) *ref = val; }
    return aux;
  }
  ```

- conditional-store of $\text{max}(G, L)$ at $G$ and return of $\text{max}(G, L)$
  ```c
  word xaf(word *ref, word val) {
    atomic { word aux = (*ref > val) ? *ref : *ref = val; }
    return aux;
  }
  ```

- assuming that $G = 42$ and $L = 4711$:
  - $XAF(G, L) \equiv \text{max}(FAX(G, L), L)$: both terms result in 4711
  - $FAX(G, L) \not\equiv \text{max}(XAF(G, L), L)$: FAX may result in 42 < 4711