Overview

- transactional memory
  - motivations, ideas, programming model
  - relation to existing coherency mechanisms
  - (lock-based) software transactional memory
  - hardware TM: cache coherence protocols and ‘bounded’ HTM
  - intermission
  - speculative execution and HTM on the Rock processor

- speculative parallel execution (thread level speculation)
  - the Hydra processor

- heterogeneous multicore design
  - motivation and uses
  - Amdahl’s Law

There is more to multicore than multiple CPUs!
Transactional Memory - Motivations

or, problems with lock-based solutions:

● convoying: a thread holding a lock is descheduled, possibly holding up other threads waiting for a lock

● priority inversion: lower-priority thread is pre-empted while holding a lock needed by a higher priority thread

● deadlock: two threads attempt to acquire the same locks do so in different order
  ⇒ a SE nightmare!

● inherently unsafe: relationship between data objects and locks in entirely by convention

● lacks the composition property: difficult to combine 2 concurrent operations using locks into a single larger operation

● pessimistic: in low-contention situations, their protection is only needed < 1% of the time!

  ■ recall overheads of acquiring a lock (even if free)
Transactions - Basic Ideas

- A transaction is a series of reads and writes (to global data) made by a single thread which execute atomically, i.e:
  - if it succeeds, no other thread sees any of the writes until it completes
  - if it fails, no change to (global) data can be observed

It must also execute with consistency:
  - the thread does not see any (interfering) writes from other threads

- It has the following steps:
  - Begin, read and writes to (transactional) variables, end

  If consistency is met, the end results in a commit; otherwise it aborts

- A condition that meets this is serializability:
  - If transactions cannot execute concurrently, consistency is assured
  - Note: this assumes that (transactional) variables are only written (in some cases, read) by transactions

  Can also achieve this if no other thread accesses the affected variables during the transaction
Transactions - Programming Model

- e.g. to add an item `newItem` to a queue with tail pointer `tail`:
  
  ```c
  atomic {
    struct node *newTail = malloc(sizeof(struct node));
    newTail->item = newItem;
    newTail->next = tail;
    // tail must not be changed between these 2 points!
    tail = newTail;
  }
  
  here, `tail` is the single read and write transactional variable

- the `atomic` block is executed atomically (it can be assumed that the underlying system will re-execute it until it can commit (what about the malloc?))

- e.g. typical code from the Unstructured Adaptive benchmark from NAS suite
  
  ```c
  for (ije1 = 0; ije1 < nnje; ije1++) {
    ...
    for (ije2 = 1; ije2 < nnje; ije2++)
      for (col = 1-shift; col < lx1-shift; col++) {
        ig = idmo[v_end[ije2], col, ije1, ije2, iface, ie];
        #pragma omp atomic
        tmor[ig] += temp[v_end[ije2],col,i je1] * 0.5;
        ...
      }
  }
  ```
Transactions: Limited Implementation by Compare-and-Swap

- the atomic compare-and-swap operation may be expressed as:

```c
typedef unsigned long uint64;
atomic uint64 cas(uint64 *x, uint64 x_old, uint64 x_new) {
    uint64 x_now = *x;
    if (x_now == x_old) *x = x_new;
    return (x_now);
}
```

(1st 2 lines are implemented by a single cas instruction; see locks.c from Lab 03)

- we can implement the (essentially atomic part of) atomic code for queue insertion by:

```c
do {
    newTail->next = tail;
} while (cas(tail, newTail->next, newTail) != newTail->next);
```

- and for the Unstructured Adaptive benchmark

```c
do {
    x_old = tmor[ig];
    x_new = temp[v_end[ije2],col,ije1] * 0.5 + x_old;
} while (cas(&tmor[ig], x_old, x_new) != x_old);
```

note: current OpenMP runtime systems implement atomic sections using a global lock (why does this perform abysmally?)
Lock-Based Software Transactional Memory

using *timestamps*, for object-based transactions (avoids locking on reads):

- **transaction objects:** `{{/*last write*/stamp, lock, value}}`
  maintain (private) sets of (pointers to) objects read and written to

- **begin:** private readStamp = globalClock

- **(first) read(obj):** if (obj→lock) abort; add(readSet, obj)

- **(first) write(obj):**
  if (obj→lock) abort; add(writeSet, {obj, clone(obj→val)})
  (updates are then performed on the cloned value)

- **end (abort implies releasing any held locks):**
  ```
  for ({obj, v}: writeSet)
      if (!acquire(obj→lock, TIMEOUT)) abort;
  private writeStamp = (globalClock++ /*atomic*/);
  if (writeStamp > readStamp + 1)
      for (obj: readSet) // concurrent update may have occurred
          if (obj→lock || obj→stamp > readStamp) abort;
      for ({obj, v}: writeSet) // commit \& release
          *obj = {writeStamp, 0, v};
  ```
Software Transactional Memory: Example

- overhead (for commit): 1 atomic operation; 1 lock acquisition per write
  - still better than naive method of acquiring a global lock at begin and releasing it at end! (although serializability is clearly guaranteed!)

- why it works: a transaction only reaches the commit stage only if:
  - all values read were up-to-date
  - no other transaction has begun to commit on the same variables

Transactions are serialized on globalClock

- example (Fig 18.24, Herlihy&Shavit 2008):

  A aborts:
  A: readStamp = 97
  B: writeStamp = 98; commit, updating c (c→stamp = 98)
  A: reads c; aborts, as c→stamp > readStamp

  A commits:
  B: writeStamp = 98; commits

  A: readStamp = 98
  A: reads c; timestamp is OK
  other threads update globalClock = 120
  A: writeStamp = 121; acquires locks on b & e
  A: commits writes to b and e, releasing locks

- hardware TM is similarly based on the 07 idea of tentative updates
Hardware Transactional Memory: Basis on Coherency Protocols

- idea: tentative values can be held in cache lines (assuming each processor has a separate cache)

- review the standard MESI cache coherency protocol

  - cache line states:
    - Modified: only this cache holds the data; it is 'dirty',
    - Exclusive: as before, but not 'dirty',
    - Shared: > 1 caches hold data, not 'dirty',
    - Invalid: this line holds no data

  - key transitions (Fig B.5, Herlihy&Shavit 2008):
    - (a) A: load x (miss; then line in E state)
    - (b) B: load x (miss; then lines in S state (A and B))
    - (c) B: store x (hit; then lines in M state (B) and I state (A))
    - (d) A: load x (miss; then B copies back line to memory and lines in S state (A and B))

  - when a line is evicted (due to a conflict with a new address needing to be loaded), the data is copied-back to memory
Hardware Transactional Memory (Bounded)

- add a transactional bit (T) for the state of each cache line
  - T = 1 if entry is placed in cache on behalf of a transaction (o.w. 0)
- simply extend the MESI protocol as follows:
  - if a line with T = 1 is invalidated or evicted, abort (with no copy-back)
  - note: can record fact that transaction will later abort instead
- an abort causes all lines with T = 1 to be invalidated (with no copy-back, if dirty)
- a commit requires all dirty lines with T = 1 to be written atomically to memory
- requires transactional variants to load and store instructions, plus an (attempt to) commit instruction (which clears all T bits)
- why this works:
  - if a line with T = 1 is invalidated, we have a read-write or write-write conflict
  - if a line with T = 1 is evicted, the transaction cannot complete
- note: size of transaction is limited by cache size (in practice, is much smaller)
- upon abort, hardware needs to indicate whether to retry (synchronization conflict) or not (error or software detected resource exhaustion)
Speculative Execution on the Rock Processor

- a general mechanism: (recall the Rock: Chaudhry et al, Fig 1)
  - each hardware register has a ‘shadow’ copy
  - when entering speculative execution mode, a checkpoint is made
  - all register updates are made to these copies
  - if speculation succeeds, these copies become the main copy (atomically)
  - if it fails, all values are discarded, and execution resumes from the checkpoint
  - success or failure causes exit from speculative mode

- this mechanism is transparent to software!

- can be used to execute non-dependent instructions after a (e.g.) load miss (before the miss is satisfied)
  - the load and any dependent instructions are placed in a deferred queue, to be replayed when the data arrives
  - benefits: can be used to scout ahead in the execution path (‘warm’ caches, TLBs, branch predictors)

- simulations indicate such hardware scouting gives $\approx 30\%$ and $\approx 40\%$ improvement in SPECfp2000 and TPC benchmarks, respectively
Hardware Transactional Memory on the Rock Processor

- based on best-effort HTM:
  - begin: execute **checkpoint** `fail_addr` instrn. (enters speculative mode)
  - all loads now set an extra s-bit in the L1$
    - if the line gets invalidated or evicted, will cause an abort
  - all stores are kept in the store buffer (c.f. T2, p96) but their **addresses** are sent to the L2$
    - if any other processor has a pending update on the same line, the L2$ signals to the processor to abort
  - end: execute **commit** instruction
    - L2$ ‘locks’ all lines to be updated; the store buffer is drained; L2$ then ‘unlocks’ lines
    - clears s-bits in L1$; exits speculative mode
  - the abort occurs immediately: registers are restored from the checkpoint, s-bits cleared, (transactional) entries in store buffer invalidated; exit speculative mode

- transaction size is limited by that of the store buffer (32 entries)

- good results: red-black trees (combined with STM), skiplists (HTM implementing double compare-and-swap), hash trees and maps (using lock elision)
Exploiting Parallelism in Sequential Programs

Consider the execution of a sequential (non-threaded) program

- potentially, ‘segments’ of execution could execute in parallel
- i.e. function calls (task ||ism) and different iterations of loops (data ||ism)
- examples (comments indicate segment numbers)

```c
quickSort(int p, int r, void *A) {
    if (p < r) { //i
        int q = partition(p, r, A); //i
        quickSort(p, q-1, A); //i
        quickSort(q+1, r, A); //i+1
    }
}
```

// malloc a, b, c ...
for (i=0; i<N; i++) {
    a[i] = b[i] = i; //i+1
}

for (i=0; i<N; i++) {
    t = i * a[i]; //i+N+1
    c[i] = t + b[i]; //i+N+1
}
Thread-level Speculation

leads to the idea of Thread-level Speculation (TLS): H/W detects suitable execution segments, converts to threads and allocates these to different cores

- threads are ordered to ‘age’ in the original (sequential) execution sequence
- ‘age’ = degree of speculativity; (current) oldest thread is non-speculative
- if H/W detects that a data dependency violation in a thread (an older thread writes data that it has read*), it aborts and any effects are cancelled
  * unless data is written before first read (e.g. t in prev. example)
- also all writes must be committed to memory in the original order;
  the (memory-visible) effect of a thread is atomic (c.f. transactions)

requires compiler support with feedback from a runtime profiler to identify appropriate segments (e.g. ‘blocking’ iterations)

wouldn’t it be nice if the hardware could do all this for us? (guaranteed correct!)

- potentially very useful for irregular applications where potential data dependencies (races) are unpredictable but rare (e.g. Unstructured Adaptive)

note: only for multicore, communication latencies (invalidations & subsequent cache line refills) are potentially very low
The Hydra Processor (with TLS)

- Hydra is a fairly conventional CMP design
  - with a write-back L1$, whose coherency is facilitated by a separate snoopy write bus
    - all processors listen in on all transactions; if another processor writes to address $x$, and data for $x$ is an L1$ line, the line is invalidated
- the following is needed to support TLS (Olokotun et al, Fig 3.5):
  - a ‘co-processor’ to create threads & maintain thread orderings
  - the L1$ is augmented with per-byte status bits:
    - modified (m): set if written to (by this or an older thread)
    - written (w): set if written to before 1st read
    - read (r): set if read and w=0
    - pre-invalidate (p): set if byte is updated by a younger thread (can discard this value when the thread terminates)
  - per thread speculative write buffers in the L2$
    (when these get filled, the thread must be stalled)
How Hydra (with TLS) Works

● upon an L1$ miss, bytes in line get ‘assembled’ from any matching entries in the write buffers (of older threads, in order of age) and then in the L2$’s lines

(Olokotun et al, Fig 3.8)
  ■ if from the former, the m-bit is set
  ■ if there is a matching entry from the buffer of a younger thread, the p-bit is set

● when a thread becomes the oldest (hence no longer speculative), its write buffer is drained to the L2$

● when a (speculative) thread gets an invalidation on a line with $r=1$ and $w=0$, caused an older thread (writing to the same data), it aborts
  ■ upon abort, the thread’s write buffer entries are discarded and lines with m-bits set invalidated; it will be later restarted

● limits to the scalability of this approach (“beyond 8–16”): bandwidth limitations for the snoopy bus

● (the use of the p-bit seems dangerous; what if a thread executes differently upon re-start?)

● in some sense, a generalization of HTM
Heterogeneous Multicore Processors

- motivation: cores are as cheap as chips! (c.f. Chaudhry et al, Fig 1)
  - applications (or functions within) are heterogeneous
  - e.g. high / low ILP, FPU/integer intensive, large/small memory footprint, high/low temporal locality
- why not tailor (a mix of) cores for the expected mix? (custom-made processors?)
- note: a super scalar core with FPU and large L1$ may be 4+ times larger than an in-order core with no FPU and a small but highly associative L1$ (Hill&Marty, Fig 1)

Figure 1. Varieties of multicore chips. (a) Symmetric multicore with 16 one-base core equivalent cores, (b) symmetric multicore with four four-BCE cores, and (c) asymmetric multicore with one four-BCE core and 12 one-BCE cores. These figures omit important structures such as memory interfaces, shared caches, and interconnects, and assume that area, not power, is a chip’s limiting resource.

(courtesy Hill&Marty08, Fig 1)
Heterogeneous Multicore Design

- the latter could be sufficient (with ‘tuning’) for:
  - interrupts & other kernel services / hypervisor / control domain, garbage collection, web and database servers (need large L1$)

- (normally) cores would have same instruction set
  - ‘specialist’ instructions (e.g. floating point) could be trapped and emulated by software
  - how about the Cell BE?
  - issue: which core type should a compiler try to optimize code for?

- it might be possible to aggregate small clusters of cores into larger ones
Amdahl’s Law for Heterogeneous Multicore

- recall Amdahl’s Law for parallel execution with \( p \) processors:
  \[
  R = \left( \frac{1-f}{R_s} + \frac{f}{pR_s} \right)^{-1}
  \]
  \( f \) is the fraction of parallel computation
- this assumes the processors are homogeneous!
- what about the heterogeneous case? (⇒ week 11 tute!)

Does it make sense to run a single application on heterogeneous cores?
Is there an argument for including a few very fast (and/or speculative) cores for the serial computation?

References


Mark Hill and Michael Marty, Amdahl’s law in the Multicore Era, IEEE Computer, June 2008. (handout)