IWannaBit!

Cliff Click
Azul Systems
1600 Plymouth Street
Mountain View, CA 94043
cliffc@azulsystems.com

ABSTRACT

Just One Lousy Bit! I want to know if any memory operation misses or any line in my L1 cache gets evicted. Why? Because with this one Bit I can write any number of lock-free algorithms easily. This Bit gives me an N-word atomic read set, and with a typical Store Conditional instruction a 1-word atomic write set. The algorithm writing community has begged for D-CAS or Hardware Transactional Memory for years, but proposals far outstrip implementations: neither are available on any commodity system. With this Bit I hope to lower the hardware costs as low as possible while still being useful.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming – *parallel programming;* D.3.3 [**Programming Languages**]: Language Constructs and Features – *concurrent programming structures*

General Terms

Algorithms, Performance, Languages

Keywords

Hardware Transactional Memory, Software Transactional Memory, Locking, Lock-Free Algorithms, Non-Blocking Algorithms.

1. INTRODUCTION

Just One Lousy Bit! I want to know if any memory operation misses or any line in my L1 cache has been evicted. Why? Because multi-core machines are here to stay, and concurrent programming is hard – very hard[1]. We (as an industry) have tried to address this difficulty with a variety of techniques: locks (by far the most common, and known to not scale to large-scale projects, known to be difficult to scale to large CPU count, deadlocks, priority inversion, not composable, etc – it's the devil we know), transactional memory (hardware [7][12], software [15][4][13] and combinations thereof [3][8][14][13][9], 'atomic' keyword [6]), application specific programming (e.g., stream programming [17], graphics [10]), even whole language changes (e.g., Erlang [1], CSP [2]). Still concurrent programming remains very hard, and non-blocking algorithms are considered harder yet [5]. With this one bit I hope to make non-blocking algorithms a little easier.

I want an *atomic-read* bit to be set if any line in the processors' L1 cache evicts or misses. I also need a way to clear, test and set this bit in software (e.g. move to/from control register or "condition code"), plus crucially: I need a variant of the standard Compare-And-Swap (CAS) or Store-Conditional (the SC of LL/SC) instruction which atomically fails if this bit is set (i.e., fails if any line has been evicted from L1 since the bit was last set). I really want a full Bloom filter on my L1 cache [11] – but I'll settle for a 1-bit atomic-read-set filter. I'll also need the OS to clear the bit (as-if a line was evicted) if another thread is allowed cycles on this CPU.

Typical usage would be to clear the atomic-read bit, issue a series of loads which I hope will be atomic, then do a conditional store and finally test the atomic-read bit. If it remained clear then all operations were cache-resident for the whole time, hence happened atomically. If the bit is set, then the sequence was not atomic (and the store did not happen). Typically I would spin and try again, now that all lines are hot in my cache. The atomic-read bit does not enforce ordering, instead it tells me when ordering did not happen. I then move the usual memory-fence stall from hardware to a software spin loop.

A CAS instruction or an LL/SC sequence gives me a 1-word atomic-read-set (and a 1-word atomic-write-set). DCAS gives me a 2-unrelated-word atomic-read-set (and a 2 word atomic-write-set), and has been proposed for a long time by the algorithm-writing community. It's not available on any commodity system; hardware engineers cite complexity and low usage as reasons not to implement. The *atomic-read* bit will still only allow a 1-word atomic-write-set (i.e., it's not N-CAS) but the atomic-read-set is much larger. It's really only limited by the size and associativity of the L1 cache. The bit allows any load to double as the 'Link' part of Load-Linked. Also this bit effectively turns a CAS into an SC – but with more than 1 line being 'locked'. For lack of a better name, I'll call this instruction a 'SC' but it's really a 'store conditionally if the atomic-read bit is clear'.

1.1 Cost of Instruction Sequences

In all the following code sequences, cache-miss costs dominate usually by 100-fold. Hence I will ignore all costs other than cache-miss costs. In this model, cache-hitting loads, branches and integer ops are free. Memory fences and memory ordering instructions usually run at external bus speeds, roughly the same as a full cache-miss; I will count these instructions as 1 cache-miss. CAS instructions typically have a memory fence built-in; if so I will count these as 1 cache-miss[2]. Multiple unrelated cache-miss-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
MSPC'08, March 2, 2008, Seattle, WA, USA.
Copyright 2008 ACM 978-1-60558-049-4/00/0004…$5.00.

[1]Try googling "notoriously difficult" and "concurrent"

[2] As far as I know, except for Azul Systems, all memory fences on multi-socket systems operate at roughly the speed of an external bus cycle (full cache-miss stall). Single-socket and single-core systems are sometimes faster.

ing loads can often overlap their miss costs. The number of simultaneous outstanding misses varies by hardware; I'll assume 2 here. Cache-missing loads can not overlap with mfences; I will count a cache-missing load followed by an mfence as a cost of 2 cache-misses.

2. SOME EXAMPLES

2.1 No-Fence Dekker's

The atomic-read bit allows me to tell when a sequence of memory operations was atomic. This will let me implement Dekker's algorithm without memory barriers although the critical section itself will still need fencing. There are two threads, *t1* and *t2*, and they have two lock bits in memory *L[t1]* and *L[t2]*. There is a *turn* variable to enforce fairness; it refers to the thread whose turn it is. Thread *t1* does:

do_forever {
  Clear atomic-read bit;
  L[t1]=true; // unconditional store
  rL2 = *L[t2]*; rturn = *turn*; // load into registers
  if(!atomic) **continue**;
  if(rL2==false) **break**;
  if(rturn==*t1*) **continue**;
  L[t1]=false; // unconditional store
  while(*turn* == *t2*) ; // spin-wait
}
acquire/critical section/release
turn=*t2*; *L[t1]*=false; // unlock

And thread *t2* does likewise. The atomic-read bit ensures that each thread operates on a coherent state of memory. E.g. at the statement "**if**(rturn==*t1*)", thread *t1* knows that at some prior point in time *L[t1]*==*L[t2]*==true and *t1* knows the value *turn* held at that moment. This is enough to tell *t1* whether it should spin trying to acquire the lock for itself (outer loop spin), or spin allowing *t2* to acquire the lock (inner loop spin). Since the reads are all atomic, the unlocking writes can happen in any order.

A fencing version would use mfences between memory operations to ensure ordering, and in the process stall the CPUs until ordering happened. The *atomic-read* version has no fences and I did not speed up memory; where did the stall go? The memory ordering stall has moved from mfence instructions (stall in hardware) to the not-atomic retry loop (stall in software). This has the advantage that we can do other work during the stall (e.g. issue more memory ops), as shown in the next example.

2.2 SpecJBB2005

SpecJBB2005 [16] has a hot piece of code that looks roughly like that shown in Figure 1. For a record-breaking SpecJBB2005 score, this code might execute on the order of 50 million times per second. There are 4 lock/unlock pairs in there and all are essentially never contended. Java semantics demand an 'acquire' at each lock and a 'release' at each unlock – which is generally implemented as some kind of memory fencing before and after the locks & unlocks – and that fencing is expensive. The atomic-read bit allows us to speculate across all 4 locks at once.

JIT'ing this code is easy: there's essentially no opportunity to optimize. For each locked region, the JIT will need to emit a lock sequence, the field load, and a matching unlock. The lock sequence

```
synchronized(A) {
  B = A.fld
} // unlock A
synchronized(B) {
  C = B.fld
} // unlock B
synchronized(C) {
  D = C.fld
} // unlock C
synchronized(B) {
  E = B.fld2
} // unlock B
...use D ... use E...
```

Figure 1: Hot code in SpecJBB2005

will itself start out with loading a lock word from the object (and that load often misses in cache), then the lock sequence itself (which typically requires an external bus cycle stall for the CAS & memory fencing), the body of the locked region (a single field load, also typically cache-missing), and an unlock sequence (again often with a CAS & memory fencing). The typical cost for this sequence is thus something like 4 non-overlapping cache-misses per lock region for the locks on A, B & C, and 2 cache-misses more for the repeated lock on B: 14 cache-misses total.

We can use the atomic-read bit to speed this up. We start by preloading the L1 cache with all the lines in question: the lines for A.fld, B.fld, & C.fld and any lock-words associated with A, B & C (which maybe the same lines as the X.fld lines depending on the cache-line size). These loads are all speculative (another thread might acquire the locks and change the field values) so they need to be loaded speculatively. We want to load them up-front because any cache-miss on load will cause an evict which will set the atomic-read bit. As shown in Figure 2, the setup costs will include 6 cache-missing loads in 3 dependent chains. Since our hardware can handle 2 cache-missing loads at once we'll only pay for half of the cache misses.

In Figure 3 we set the atomic-read bit and start loading. All words are loaded again to ensure they are still in cache and checked to make sure they haven't changed: we now know we have an atomic snapshot of memory. Since all locks are unlocked we can act as-if we took the locks, read the fields, and unlocked all in a single instant in time.

```
ld A.lock; ld B=A.fld; // cache miss 1
ld B.lock; ld C=B.fld; // cache miss 2
ld C.lock; ld D=C.fld; // cache miss 3
```

Figure 2: Setup code

```
clear atomic-read bit
// Re-read all fields
ld A.lock; ld B'=A.fld;
ld B.lock; ld C'=B.fld;
ld C.lock; ld D =C.fld
ld E =B.fld2
// Confirm unlocked and no-change
br A-is-locked to fail
br B-is-locked to fail
br C-is-locked to fail
br B != B' to retry
br C != C' to retry
br atomic-read to retry
// Victory!
```

Figure 3: Speculating across 4 locks

```
retry:
    clear atomic-read bit
    ld A.lock; ld B =A.fld
    br A-is-locked to fail
    ld B.lock; ld C =B.fld
    br B-is-locked to fail
    ld C.lock; ld D =C.fld
    br C-is-locked to fail
    ld E =B.fld2
    br atomic-read then retry
```

Figure 4: Again, with no setup code

All the reads are L1 cache-hits and independent so this sequence should complete in a handful of cycles, plus the setup cost: about 3 cache misses total.

The atomic-read bit is also set on evicts and we can take advantage of that. No misses and no evicts implies that we hold all the lines in our L1 cache over the entire speculative region. Since we already have a speculation retry loop there's no need for any initial speculative reads. As shown in Figure 4 we can skip the setup code. Just start reading things – if anything is not in cache, the CPU will miss, set the bit and trigger a retry. The retry should all be L1 cache-hits (assuming low contention) and so will run really fast. Same cost as Figure 3 but without the setup code.

Notice there is no memory fence instruction. To succeed, all lines must be cache-resident for the entire loop hence there are no memory-ordering games to be dodged. Also notice that I didn't need to write any lock-words, it is as-if I took and released all the locks in a single instant in time (no visible locking happened, no other CPU could have witnessed the locks being held). I merely confirmed that no other thread modified these lines.

This trick only works on locks where the compiler can see the whole lock region[3] and verify no writes in the region. In the spirit of Speculative Lock Elision [12] these locked regions can all run concurrently with other threads that are also locking the same locks speculatively. If speculation begins failing too often, you can revert back to locking behavior. Code generation here is particularly easy: an initial retry-label/clear-bit sequence is emitted, then each 'lock' is implemented with a 'ld/br' sequence (the target of the branch is a slow-path that does correct locking), the lock bodies are inlined, and 'unlocks' are empty. A final test&retry ends the sequence.

2.3 A Concurrent Queue

We can easily implement a lock-free (not just obstruction-free) fixed-size concurrent queue [5]. We define the queue as a ring buffer with each word holding either **null** or a value. Values are stored adjacent in the ring buffer with no intervening **null**s, and the remainder of the ring buffer is filled with **null**s. The queue head is denoted by the sequence "value, **null**" and the tail by "**null**, value". We require at least 1 **null** between the queue head and tail. We use a special **empty** sentinel value only when the queue is empty. To use the queue, threads scan[4] the queue for the head or tail and start atomic update operations only after discovering the correct queue end. Since the position of the ends are constantly moving, threads use the *atomic-read* bit to only modify the queue when the ends do not move out from under them.

In the steady-state, when the queue is neither full nor empty, threads atomically read 3 words in a row[5] from the proper queue end. Threads doing a 'put' look for the sequence "value1, **null**, **null**" and atomically change the middle **null** to insert their value – producing the sequence "value1, value2, **null**". Threads doing a 'get' look for the sequence "**null**, value1, value2" and atomically change the middle value1 to a **null**, producing the sequence "**null**, **null**, value2".

If a 'put' thread *atomically* finds the sequence "value1, **null**, value2" then the queue is full. Note that this sequence is possible for a not-full queue if the queue is changing during reading the 3 words – but the atomic-read bit will tell us that. If a 'get' thread *atomically* finds the sequence "**null**, value, **null**" then a successful get will empty the queue and make the queue head position undefined. In this case the 'get' operation replaces the value with the **empty** sentinel, and similarly a 'put' thread will replace the **empty** sentinel with the queue's first value.

3. A HANDY READ BARRIER

These next few examples will assume we have a simple *read-barrier* on pointer loads. Loads need to test for the read barrier – it's presence signals special behavior, usually a concurrent update-in-progress by another thread. For the C language, we steal the low-order bit. For Java, we can use boxing. In any case the original pointer payload needs to be available to all threads after stripping off the barrier (bit-extract for C, unboxing for Java). The barrier can be set with a single CAS instruction.

3.1 Concurrent singly-linked list update

The problem with concurrent singly-linked list update [18] is that one thread can be inserting while another deletes, leading to a broken list. Figure 5 shows a single-threaded insert using LL/SC. The LL/SC verifies that B.next does not change between loading it and changing it (I could use CAS here to the same effect). In Figure 6, two threads (T1 and T2) compete to modify the list, both using a LL/SC (or CAS) sequence – and still the list ends up being corrupted! Although B is no longer in the list (fulfilling T2's wishes), C did not get inserted and T1 isn't aware of that.

To fix this, we add the invariant that any Node being deleted first has it's next field 'poisoned' with the read-barrier. Setting the read-barrier marks the *linearization* point for the delete operation. Any traversal seeing the 'poison' will complete the delete operation before continuing on. In this example T2 sets the barrier on B.next, and then any thread (including T2) seeing the barrier will unlink B. After B.next is poisoned, T1's attempt to update B.next will fail – and T1 will need to retry. This preserves correctness of the list. Unfortunately any number of Nodes just prior to B might also have been deleted, leaving T1 orphaned. T1 may be forced to rescan from the list start to find the new insertion point[6]. In the special case of adjacent partially deleted Nodes A and B, the attempt to delete B that would normally CAS B.next into A.next needs to preserve the read-barrier in A.next.

---

[3] Commonly **not** possible when JIT'ing large Java programs; nonetheless a significant fraction of Java locks are around simple getter's and setter's.

[4] Making this scan efficient is a task for a longer paper, although for slow-moving queues the simple heuristic "start scanning where you last touched the queue" probably suffices.

[5] If the ring buffer is sized and aligned to multiples of a cache-line, only 2 unique lines will ever be touched. In this case it suffices to read only the 1st and 3rd words.

[6] If the list is a *skip-list*, restarting is not very expensive – just O(lg N); otherwise the restart can be very expensive.

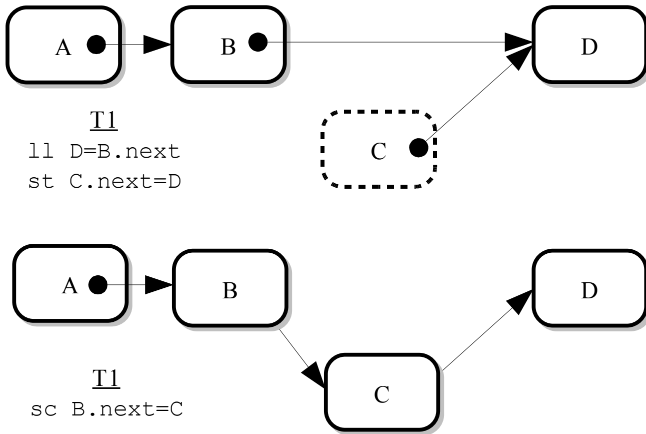

Figure 5: Single-threaded singly-linked list insert

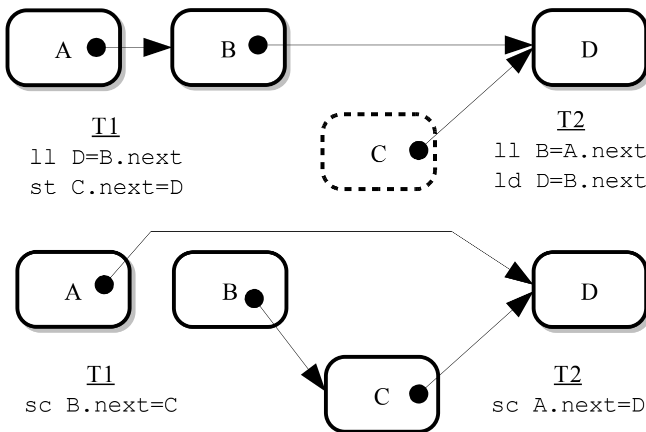

Figure 6: Concurrent singly-linked list update

Singly-linked lists have only linked with 1 word, so they can be concurrently updated with single-word atomic-read-set instructions. Doubly-linked lists need two words to be atomically read and cannot be correctly updated with LL/SC or CAS.

3.2 Concurrent doubly-linked list update

The doubly-linked list update proceeds nearly identically to the singly-linked case, except we will be reading multiple words and using the atomic-read bit to ensure those words are not being changed by concurrent updates. We'll also need to write multiple words, and since we don't have a way to do that atomically other threads can see the updates “half-done”. The invariant we use is that the forward links always represent the “correct” state of the list, and the back links are only useful for efficient removal. At any point if the links are not coherent, we adjust the back link to match the front link.

In Figure 7 we see T1 atomically verifies that B & D are properly linked together (by reading B.next and D.prev), then it initializes and inserts the first pointer to C. The atomic-read bit prevents inserting C while other edits are being done to B and D. However this first pointer write point is the *linearization* point for the insert of C. Sufficient information is now available to other threads to finish the insert of C (swing D.prev to C). In this example T2 discovers that C is partially inserted and finishes the job. Again the atomic-read bit allows the safe completion of the insert of C – if T2's two loads are not atomic it would be possible for another

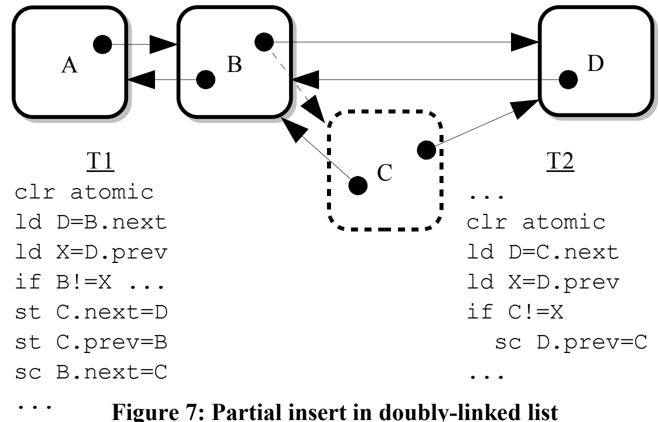

Figure 7: Partial insert in doubly-linked list

thread to finish the insert of C, then start inserting a new link between C and D, and T2's late store of D.prev would break the new link.

Deletion follows the same pattern as the singly-linked list case: to delete B we mark B.next with the read-barrier. This is the *linearization* point for the deletion of B, and prevents any inserts after B. Visiting threads discovering the deleted B still in the list can adjust the pointers as needed to remove B. Setting the read-barrier requires atomically verifying that A.next==B, then poisoning B.next – an atomic-read set of 2 words. Again, removing A.next to skip around B requires an atomic-read set of 2 words, and finally adjusting the back pointer again requires an atomic-read set of 2 words.

In all likelihood, these updates will be done in a single pass by the initiating thread, with a single set and test of the atomic-read bit – but that is not required. Once the first update has started the original thread can block or fail it's atomic-read set spuriously (e.g., badly timed interrupt triggers some cache misses) and other threads can finish the job, restoring the links to a coherent state.

A final note: this algorithm is *lock-free* not just obstruction-free. The setup for any given list edit involves only reads, and the reads from one setup should not kick out the atomic 'sc' from another edit. Once the initial store happens, the edit is conceptually complete – the remaining updates are only there to restore the links and all threads will be making the same updates to correct things.

4. RELATED WORK

Other than the venerable DCAS, most hardware supporting for concurrent and non-blocking coding involves *hardware transactional memory*. There have been a large number of proposals most involving hardware that, while technically possible, have not been economically feasible enough to appear in mainstream CPUs. The original TM paper [7] required fairly extensive reworking of the L1 cache; SLE [12] requires a secondary cache for buffering results, as well as fairly extensive hardware for predicting locks and unwinding failed speculative locks; VTM [13] adds overflow tracking and status words to a presumed already high-performance hardware-only solution. There are numerous other approaches involving extensive caching of speculative results.

In more recent years the hardware requirements have generally gone down mostly by helping *software transactional memory* [3] instead of doing all the work in hardware. LogTM [9] adds a few instructions, a register, and the ability to flash-clear L1 cache status bits. Hybrid TM [8] is one example that requires a few extra

bits per cache line; Azul Systems uses similar hardware support. Most of these TM's assume the ability of hardware to checkpoint and recover architected register state, as-if a speculated branch mis-predicted.

The simplest proposals I am aware of end up with a few extra bits per cache line [14]. None require as little hardware as this work, which requires only a single bit per entire cache. Indeed, the atomic read bit is an obvious replacement for LL/SC and can replace CAS in many applications. Replacing support for these primitives could be a net reduction in hardware.

I was inspired by [11] and by reading a pre-publication version of Purcell's thesis which proposes using a Bloom filter to track atomic-read-sets in the cache. I merely observed that the Bloom filter can be reduced to 1 bit while remaining useful.

5. CONCLUSIONS

This isn't a direct replacement for full-fledged Transactional Memory or N-CAS – this approach can only write 1 word atomically and only read atomically words within the size and associativity constraints of our cache. Algorithms that require reading atomically more words than the caches' associativity will always fail for some unfortunate selection of addresses.

However the hardware costs here are low – one bit per entire L1 cache. This is much lower than the typical HTM proposal. For typical cache sizes, many algorithms can be efficiently constructed: the ability to atomically read even 2 words goes a long way. We can also speculate across many locks at once (effectively 'coarsening' the locks into one big lock) – and pay the fencing and memory-coherency costs only once.

We can use the atomic-read bit and a StoreConditional instruction to implement locks that end in a single word update – which really turns the locked region into a larger spin-update-loop (something similar was done by hand to java.util.Random some years ago, as a way to spam the JavaGrande Monte-Carlo benchmark).

With the atomic-read bit we can write simple lock-free algorithms for at least Dekker's algorithm, dequeues, and concurrent doubly-linked lists, all for very minimal amounts of hardware.

6. REFERENCES

[1] Armstrong, J., Williams, R., Virding, M. *Concurrent Programing in Erlang*. Prentice-Hall, 1993.

[2] C. A. R. Hoare, Communicating Sequential Processes, *Communications of the ACM*, v.21 n.8, p.666-677, Aug. 1978 DOI=10.1145/359576.359585

[3] Damron, P., Fedorova, A., Lev, Y., Luchangco, V., Moir, M., Nussbaum, D. Hybrid Transactional Memory. 2006. *In Proc. of the 12th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 336-346.

[4] Dice, D., Shavit, N. 2007. Understanding Tradeoffs in Software Transactional Memory. *Proceedings of the Intl. Symp. on Code Generation and Optimization,* 21-33.

[5] Doherty, S., Detlefs, D., Groves, L., Flood, C., Luchangco, V., Martin, P., Moir, M., Shavit, N., Steele, G. 2004. DCAS is not a Silver Bullet for Nonblocking Algorithm Design. *Proceedings of the 16th Annual ACM Symp. on Parallelism in Algorithms and Architectures*, 216-224.

[6] Harris, T., Fraser, K. 2003. Language Support for Lightweight Transactions. *Proceedings of the OOPSLA '03 Conference*, 388-402.

[7] Herlihy, M., Moss, E. 1993. Transactional Memory: Architectural Support for Lock-free Data Structures. *In Proceedings of the 20th Annual Intl. Symp. on Computer Architecture (ISCA'93)*, 289-300.

[8] Kumar, S., Chu, M., Hughes, C., Kundu, P., Nguyen, A. 2006. Hybrid Transactional Memory. *Proceedings of the 11th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP'06)*, 209-220.

[9] Moore, K., Bobba, J., Moravan, M., Hill, M., Wood, D. 2006. LogTM: Log-based Transactional Memory. *In Proc. of the 12th Annual Intl. Symp. on High Performance Computer Architecture.*

[10] Owens, J., Luebke, D., Govindaraju, N., Harris, M. Krüger, J., Lefohn, A., Purcell, T. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, V 26 N 1, Mar 2007, pg 80-113.

[11] Purcell, C., Harris, T. 2004. Brief Announcement: Implementing Multi-Word Atomic Snapshots on Current Hardware. *PODC: 23th ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*

[12] Rajwar, R. and Goodman J. 2001. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Microarchitecture*, December 2001

[13] Rajwar, R., Herlihy, M., Lai, K. Virtualizing Transactional Memory. *In Proc. of the 32nd Annual Intl. Symp. On Computer Architecture*, June 2005.

[14] Saha, B., Adl-Tabatabal, A., Jacobson, Q., 2006 Architectural Support for Software Transactional Memory. *Proceedings of the 39th Annual IEEE/ACM Intl. Symp. on Microarchitecture*, 185-196

[15] Shavit, N., Touitou, D. Software Transactional Memory. *Proceedings of the 14th ACM Symp. on Principles of Distributed Computing*, pp.204–213. August 1995.

[16] Standard Performance Evaluation Corporation. *SPECjbb2005 (Java Business Benchmark) Documentation,* release 1.07, 2005.

[17] Thies, W., Karczmarek, M., Amarasinghe, S. StreamIt: A Language for Streaming Applications. *In Proceedings of the 2002 Intl. Conference on Compiler Construction*, Grenoble, France, April, 2002.

[18] Valois, J. 1995. Lock-free Linked Lists using Compare-And-Swap. *Proceedings of the 14th Annual ACM Symp. on Principles of Distributed Computing*, 214-222.\

7. APPENDIX: OTHER APPLICATIONS

7.1 N-Way Dekker's

While Dekker's is normally presented as a 2-cpu algorithm, I can extend it in a straight-forward way to N-way where N is limited by the set of words I can atomically read (minus 1)[7]. I extend the *L* array to size N. The *turn* variable is advanced by one modulo #threads when exiting the critical section. Each thread *t* does:

do_forever {
  Clear atomic-read bit;
  L[t]=true;
  Read all lock bits *L* into rL; rturn=*turn*;
  if(!atomic) **continue**;
  if(all rL[] are false except rL[*t*]) **break**;
  if(all rL[from *turn* to (*t*-1)] are false) **continue**;
  L[t]=false; // unconditional store
  while(*turn* == rturn) /*spin-wait*/;
}
acquire/critical section/release
turn=(*t+1*) % #threads; *L[t]*=false;

The fairness condition, "all rL[from *turn* to (*t*-1)] are false" needs some more explanation. To pass this test requires that no threads before *t* in turn order set their *L[]* bit. This allows the *contending* thread that is next in turn order to acquire the lock. Uncontending threads are ignored, and threads trying to acquire out of order will set *L[t]*=false and spin-wait.

7.2 Helping N-CAS

The atomic-read bit is not an N-CAS, but it can remove the "read-set" portion from the complex write-set portion of N-CAS. i.e., an N-CAS which requires 7 read-set words and 2 write-set words can be reduced to atomically reading the 7 read-set words, plus a 2-CAS. In the transactional memory world, transactions usually contain a much larger fraction of reads than writes. We can reserve the complex (expensive) software layer for just the writes.

7.3 An Atomic Wait-Free "never fails" "well-known" N-CAS...

... for small values of N (but larger than 1!). This section is definitely "work in progress"; I have not dotted my I's nor crossed my T's here!

"well known": I require a description of the N-CAS to be available to all threads before the N-CAS commits; the description is a list of the read-set addresses, the write-set addresses, and functions of the read-set that describe the results to be written. With this description any thread can complete another thread's N-CAS (I'll use this "helping" to achieve wait-freedom). The requirement that the read-set be known ahead of time is an interesting restriction: often one read-set word will point to a data structure which will contain another value in the read-set. I'll also sort the write-set addresses (breaks ties, prevents deadlock), and use my handy read-barrier on the write-set words.

"never fails": Transactions are strongly ordered by who gets to slap down the read-barrier first. "Losers" do not abort their N-CAS, but instead queue up behind the winning N-CAS. Thus I never need to abort any N-CAS; they just get ordered.

Wait-Free: I borrow a page from the N-Way Dekker's algorithm; contending N-CAS's are strongly ordered and will commit in *turn* order. Since the order that N-CAS's commit is known, and how to complete a "well known" N-CAS is public knowledge – I do not need to abort one N-CAS for another. Instead, upon detecting conflict the threads agree on an order for the N-CAS's to commit and then help each other to commit earlier N-CAS's until their own N-CAS commits. This puts an upper bound on the time a thread must wait for it's own N-CAS to commit.

Atomic: The read-barrier on the write set is used to force non-transacting reads to recognize an XTN is in progress. They need to complete the XTN to clear out the read-barrier and see a coherent state of memory (alternatively they could go to the in-progress XTN and read directly from it). Non-transacting writes are more of a problem: I can either make them complete any prior transactions (preserving my never-fail property), or allow them to restart a partially installed-but-not-committed transaction.

The algorithm goes something like this:

1. *T1* prepares a "well known" transaction XTN: a list of read-set addresses, write-set addresses, and functions of the read-set.
2. *T1* flips the write-set with read-barrier'd ptrs to the XTN. *T1*'s transaction is "being installed" at this point and is not yet committed.
    - Readers "peek through" the read-barrier to see the original write-set words.
    - Non-transactional writes act like size-1-transactions (to keep my never-fail property).
    - If *T1* sees a barrier already then there is a conflict. If the conflict includes a committed transaction, *T1* helps complete it.
    - If *T1* sees a barrier, and the barrier points to a collection of uncommitted transactions then *T1* unions into this set. Similar to Dekkers, a global *turn* variable will be used to determine the winning order.
3. *T1* is *installed* and ready to commit. *T1* consults *turn* to determine the order that transactions should commit. *T1* helps other winners first.
4. Any thread seeing an *installed* XTN whose *turn* has come attempts to compute an *snapshot*, a complete atomic read of the XTN's read-set. A *snapshot* is just a list of the atomically-read data.
5. Once the snapshot is atomically inserted into the XTN then the XTN is *committed* and any thread can compute the new write-set values and begin writing.
6. Write out the new write-set values, removing the read-barrier in the process.

[7] I only need a boolean per thread and I can bit-pack; e.g. a single 32-byte cache line will suffice for 256 threads. However, sharing bits will require some form of atomic-update when setting *L[]*.