Nonblocking Synchronization for Emerging Parallel Systems

Michael L. Scott
University of Rochester
(joint work with Bill Scherer & Virendra Marathe)
17 May 2005
Parallelism Goes Mainstream

- Microprocessors have hit the “heat wall”
  - SMT/CMP processors now in servers; soon to be in desktops and laptops
- Parallel programming no longer for experts
- Been at this for 40 years; it’s time to deliver
  - For user code
  - For systems — scalability without Herculean effort
Mutual Exclusion Locks

Coarse grain easy to use, but not scalable
- Serialization
- Preemption or page faults in critical sections
- Failures lead to data corruption — can’t share between mutually untrusting threads

Fine grain scalable but *hard to use*
- Deadlock
- Priority inversion
- Every data structure a publishable result
- Enormous SW effort to build a scalable kernel
Outline

- The “nonblocking religion”
- Ad hoc nonblocking *dual data structures*
- Software transactional memory (STM)
  - The design space
  - Adaptivity
  - Contention management
  - Possible (optional) hardware assist
Nonblocking Synchronization

- My thread never has to wait for other thread(s) to reach some acceptable state.
- Formalized via *linearizability*
  - Operation takes effect with a single atomic instruction, typically CAS or SC.
  - There may be significant advance set-up, which can be tossed on abort and restart.
  - There may be significant subsequent cleanup, which can be done by others.
Nonblocking Progress Conditions

- **Wait-free**
  - Every thread makes progress in a bounded number of its own time steps (usually impractical)

- **Lock-free**
  - *Some* thread makes progress in a bounded number of my time steps (admits starvation)

- **Obstruction-free**
  - I make progress, regardless of the state of other threads, if they all hold still long enough (admits livelock)
Nonblocking Advantages

Semantics
- No deadlock or priority inversion
- No data corruption on thread failure
- No performance loss from preemption or paging

Performance
- Potential high concurrency
- Low constant-time overhead for important ad-hoc structures: counters, stacks, queues, lists, deques, skip list heaps, hash tables
Queues on a Dedicated Machine

![Graph showing normalized time against processors for different configurations: TATAS-B (1), TATAS-B (2), and M&S.](image)
One Extra Process Per Processor
Two Extra Processes Per Processor

![Graph showing normalized time vs. number of processors for TATAS-B (1), TATAS-B (2), and M&S.](image)

- **TATAS-B (1)**: x-x
- **TATAS-B (2)**: o-o
- **M&S**: x-x

The graph illustrates the normalized time performance for different numbers of processors, with TATAS-B (1) and TATAS-B (2) showing distinct trends compared to M&S.
Nonblocking Challenges

- **Condition synchronization**
  - Methods traditionally required to be *total* — callable in any situation
  - Dual data structures

- **Programmability**
  - Ad hoc nonblocking structures at least as complex as those with fine-grain locks
  - Each variant a publishable result (e.g. `malloc`, PLDI 2004)
  - STM
Partial Methods Matter!

- Dequeue, pop, deletemin, . . .
  - Condition synchronization in general
    (anything for which a thread might wait)
- Busy-wait “solution”:
  
  ```
  repeat
  t = Q.dequeue()
  until t != ⊥
  ```

  - High contention
  - No fairness guarantees
Dual Data Structures [DISC’04]

Don’t spin on the return value; insert a request and spin on that instead

Future fulfilling operation terminates the spin
- Simple programming interface
- No contention: spin only on local locations
- Fair: object semantics determine order in which pending requests are satisfied

Can be lock-based or nonblocking
(focus on the latter in this talk)

(Name and idea inspired by firmware mechanism on the c.1980 BBN Butterfly Parallel Processor)
Nonblocking Dual Data Structures

1. Every operation either completes or registers a request in a nonblocking fashion.
2. Fulfilled requests complete in a nonblocking fashion.
3. Threads that are waiting for their requests to be fulfilled do not interfere with the progress of other threads.

Examples: nonblocking dualstack, dualqueue; Java SynchronousQueue, Exchanger.
NB DualQueue Results

16-processor machine
SynchronousQueue Results

Handoffs/s [Patience: 100us]

16-processor machine
Outline

- The “nonblocking religion”
- Ad hoc nonblocking *dual data structures*

  ➔ Software transactional memory (STM)
  - Adaptivity
  - Contention management
  - Possible (optional) hardware assist
Software Transactional Memory

- General purpose mechanism to turn correct sequential code into correct, highly concurrent code (disjoint access parallel, or nearly so)
- Algorithms date from early 1990s; term coined by Shavit and Touitou [1995]
- Many papers in mid 1990s, but time and space overheads too high
- Largely inspired by Herlihy & Moss [ISCA 1993] and related HW proposals; also considered impractical until recently
STM Reborn

- Sudden urgency: have to face the heat wall
- Multiprocessor and memory architecture mature enough to consider the HW designs

STM breakthroughs

- Sun DSTM [PODC’03]
- Cambridge/Microsoft OSTM and WSTM [OOPSLA’03]

- Reasonable time and space overheads
- Performance better than coarse-grain locks
- No special HW requirements, but may benefit from HW support
STM and Coarse-Grain Locks

IntSetHashtable benchmark, 16 buckets, 256 values, 10% writes
Object-Based STM Design Space

- Lock-free v. obstruction-free
- Eager v. lazy object acquisition
- Per-object v. per-transaction metadata
- Direct or indirect access for readers
Classifying STM Systems

White areas have nothing in particular to recommend them.
Grey areas seem like distinctly bad ideas.
ASTM adapts across entire quadrant.

- White areas have nothing in particular to recommend them.
- Grey areas seem like distinctly bad ideas.
- ASTM adapts across entire quadrant.
ASTM Reader Access

Direct access for mostly-read object
Acquiring a Recently-Read Object

- Writer installs DSTM-style locator;
  retained by subsequent writers
Reading a Recently-Written Object

- Reader reverts to direct access

Available for garbage collection
ASTM Advantages

- Obstruction-free: no helping, no sorting
- Per-object metadata: no searching for current copy
- Adaptive incremental cleanup
  - Avoid indirection in read-intensive workloads
  - Avoid cleanup in write-intensive workloads
- Both eager and lazy acquire...
Acquisition Time

- **Eager acquire (early conflict detection)**
  - Reduces wasted work in doomed transactions
  - Increases chance that transaction will abort in favor of a peer that eventually also aborts

- **Eagerness usually wins by a little bit**

- **Exception: applications that do a lot of early releases and a few writes**
  - Lose if I read an object that you’ve acquired for writing
  - Win if you lazy acquire it and I early release it first
Indirection Adaptivity Results

LFUCache

RBTTree
Acquisition Adaptivity Results

RandomGraphList benchmark, max 256 nodes
Contention Management

When conflict is detected (attempt to access a currently-acquired object), should we
- Wait
- Self-abort
- Abort the “enemy”?
  (must always be willing to do so eventually)

CM interface provides answers, using hooks at all relevant points in the code
- Start, abort, commit
- Open, upgrade, early release
- Validation failure
CM Policies

- Wide variety of options
  - Using time, backoff, turn-taking, greed, relative investment, relative status (running, blocked), randomization . . .
- “Polka” does well in general
  - Combination of “Polite” (exponential backoff when other transaction has priority) and “Karma” (retained cumulative investment)
- Not easy to change in HW!
Hardware Assist?

Several very fancy systems under development

- Transactional Lock Removal [Rajwar & Goodman]
- Speculative Synchronization [Martinez & Torrellas]
- Unbounded Transactional Memory [Ananian et al.]
- Transactional Coherence and Consistency [Hammond et al.]
- Thread-level Transactional Memory [Moore et al.]
- Speculative Lock Reordering [Rundberg & Stenström]

Sun pursuing a simpler HW/SW hybrid (HSTM)

Is even that necessary?
“DCAS is not a Silver Bullet”

- Paper by Doherty et al., SPAA 2004
  - Observe that double-word compare-and-swap doesn’t make ad hoc NB synchronization easy
- But what if it made STM fast?
  - Reuse inactive locator
  - Steal active locator (2 DCASes or a 3CAS)
  - Join read list (visible reads) w/o corner case
  - Early release from read list (visible reads)
  - Lazily acquire N nodes at once (NCAS)
  - Kill N visible readers at once (NCAS)
  - Avoid mutual kill in contention manager
  - Release word and hash table entry in WSTM [PODC'05 poster]
- Worth considering!
Summary Recommendation

- Nonblocking synchronization in place of locks
- Dual data structures for condition sync
- Fast ad hoc structures in special cases
- STM for the general case
  - Addresses deadlock, priority inversion, preemption/faults/failures
  - The programming ease of coarse-grain locks with (most of) the performance of fine-grain locks
- Possible HW assist
Future Work

DDS
- More data structures
- Integrate with STM

STM
- Find the right programming model
  - IO, nesting, conditions
- Integrate with existing models, languages, compilers
- Overhead reduction
- Visible v. invisible (eager v. lazy) reads
- Hardware acceleration/hybrids; portability
www.cs.rochester.edu/~scott/synchronization/