Introduction to Transactional Memory

Sami Kiminki
2009-03-12

Presentation outline

Contents

1 Introduction ........................................ 1
   Annotations ........................................... 3

2 High-level programming with TM .................. 3
   Annotations ........................................... 4
   Annotations ........................................... 5

3 TM implementations .................................. 5
   Annotations ........................................... 7
   Annotations ........................................... 7
   Annotations ........................................... 7
   Annotations ........................................... 8
   Annotations ........................................... 8

4 TM in Sun Rock processor ......................... 8
   Annotations ........................................... 9
   Annotations ........................................... 10
   Annotations ........................................... 10

1 Introduction

Motivation

• Lock-based pessimistic critical section synchronization is problematic

• For example
  – Coarse-grained locking does not scale well
  – Fine-grained locking is tedious to write
  – Combined sequence of fine-grained operations must often be converted into coarse-grained operation, e.g., move item atomically from collection A to collection B
  – Not all problems are easy to scale with locking, e.g., graph updates
  – Deadlocks
  – Debugging is sometimes very difficult

• Critical section locking is superfluous for most times

• Obtaining and releasing locks requires memory writes

• Could we be more optimistic about synchronization?
The idea of transactional computing

- Optimistic approach
  - Instead of assuming that conflicts will happen in critical sections, assume they don’t
  - Rely on conflict detection: abort and retry if necessary
- If critical section locking is superfluous most of the time, aborts are rare.
  - Typically threads manipulate different parts of the shared memory
  - Consider, e.g., web server serving pages for different users

High hopes for transactional computing

Some often pronounced hopes for transactional computing but still with little backing of experimental evidence in real-life implementations

- Almost infinite linear scalability
- Scalability to “non-scalable” algorithms
- Relaxation of cache coherency requirements ⇒ still more hardware scalability
- Effortless parallel programming
- Less and easier-to-solve bugs due to the lack of locks
- Saviour of parallel programming crisis

Not a silver bullet

- No deadlocks but prone to livelocks
- Not all algorithms can be made parallel even with speculation
- Mobile concerns: failed speculation means wasted energy
- Real-time concerns: predictability

Transactional memory (TM)

- Technique to implement transactional computing
- The idea
  - Work is performed in atomic isolated transactions
  - Track all memory accesses
  - If no conflicts occurred with other transactions, write modifications to main memory atomically at commit
- Conflict
  - Memory that has been read is changed before transaction is committed — i.e., input has changed before output is produced
  - Transaction is aborted, but may be later retried automatically or manually
Some basic implementation characteristics

- Isolation level
  - weak — transactions are isolated only from other transactions
  - strong — transactions are isolated also from non-transactional code

- Workset limitations
  - maximum memory footprint
  - maximum execution time
  - maximum nesting depth
  - or unbounded if no fundamental limitations

- Conflict detection granularity

Annotations

A good introduction into transactional memory can be found in [1]. Transactional memory is an active research topic, as is indicated by the number of recently published articles in various journals and conference proceedings, see e.g. the bibliography section.

Arguably, the transactional memory techniques are sparked from Tom Knight’s work with LISP in 1986 which considers making LISP programming easier for developers by utilizing small transactions [11]. The modern era with current semantics is presented in [10].

Transactional memory techniques are interesting because they can potentially enable the use of various other techniques. For example, cache coherence protocols in multicore systems could benefit of utilizing transactional memories [9].

However, until very recently, almost all results have been more or less academic. Especially, hardware-related results have almost invariably been produced by simulations. Because of this, one could question the feasibility of these results. Pure software approaches are being criticized in high-level publications [4].

Finally, even if transactional memory techniques prove successful, it is important to note that they are not likely to revolutionize the world—by themselves, at least. Instead, in the author’s opinion, these techniques should be considered complementary.

2 High-level programming with TM

Section outline

Quick glance into high-level programming interfaces

- Transactional statement in C++ (Sun/Google approach)
- OpenTM

A low-level interface will be introduced later in Sun Rock section

Transactional statements in C++ (1/3)

- Sun/Google consideration, but not a final solution
- Basic syntax: `transaction compound_statement`
- Target: STM, weak isolation, closed nesting, I/O prohibited
Transactional statements in C++ (2/3)

- Starting and ending a transaction:
  - Tx begins just before execution of transactional compound statement
  - Tx commits by normal exit (statement executed or continue, break, return, goto)
  - Tx aborts by conflict, throwing an exception or executing longjmp that results exiting the transactional compound statement

- Special considerations for throwing exceptions
  - How to throw an exception if everything is rolled back, also the construction of thrown object(!)
  - Restrictions for referencing memory from thrown objects are likely to apply

Transactional statements in C++ (3/3)

Example code:

```cpp
// atomic_map
// Implemented by inheriting std::map and wrapping all
// data manipulator methods into transactions
#include <map>

template<class key_type , class mapped_type>
class atomic_map : public std::map<key_type, mapped_type> {

    public:

        std::pair<typename atomic_map::iterator , bool>
            insert(const typename atomic_map::value_type &v) {
                transaction {
                    return std::map<key_type, mapped_type>::insert(v);
                }
            }
    }
};
```

Annotations

The Sun/Google consideration with open issues is presented in [5]. It is mentioned that they are more prone to get some useful bits working quickly than making a full specification, in which everything is considered. Considering the authors and timing, this work is likely connected to the forthcoming Rock processor.

OpenTM (1/3)

- Extension to OpenMP
- Targets: strong isolation, open and closed transaction nesting, I/O prohibited
- Speculative parallelism

OpenTM (2/3)

- New constructs to specify transactions
  - #pragma omp transaction — atomic transaction
  - #pragma omp transform — each iteration is a transaction, may be executed in parallel
- `#pragma omp transsections` `#pragma omp transsection` — OpenMP parallel sections, transactionally executed
- `#pragma omp orelse` — Executed if transaction was aborted

- Additional clauses to specify commit ordering, transaction chunk sizes, etc

OpenTM (3/3)

Example code:

```c
#pragma omp parallel for
for (i=0; i<N; i++)
{
    #pragma omp transaction
    bin [A[i]] = bin [A[i]] + 1;
}
#pragma omp target schedule (static, 42, 6)
for (i=0; i<N; i++)
{
    bin [A[i]] = bin [A[i]] + 1;
}
#pragma omp transsections ordered
#pragma omp transsection WORK A();
#pragma omp transsection WORK B();
```


Annotations

OpenTM [2] is a much broader approach to utilize transactional memories than Sun/Google consideration of TM-enhanced C++. There is much more consideration on practical issues such as commit ordering, nesting styles, and speculative parallelization. GCC 4.3-based compiler implementation and simulator exists, see [http://opentm.stanford.edu/](http://opentm.stanford.edu/) for details.

3 TM implementations

Section outline

Glance at transactional memory implementations

- Fundamentals
- Software transactional memory
- Hardware-accelerated software transactional memory
- Hardware transactional memory
- Hybrid transactional memory
- Note on supporting legacy software

Fundamentals (1/3)

Data versioning

- Lazy versioning
  - Transaction hosts local copy of accessed data
  - Writes go to commit buffer
  - Data is written into main memory when transaction commits
- Eager versioning
Transactions write data immediately into main memory. Isolation is provided by locking and/or aborting conflicting transactions. Overwritten values go to undo buffer. Undo buffer is executed when transaction aborts.

**Fundamentals (2/3)**

Conflict detection

- Pessimistic conflict detection
  - Conflicts are detected progressively with reads and writes
  - Conflicts are resolved by aborting or stalling progress
  - Circular conflicts may halt progress altogether unless specifically detected

- Optimistic conflict detection
  - Conflicts are detected at commit time at, resolved by aborts
  - Works only with lazy versioning
  - Efficient only when conflict probability is low
  - Perhaps less latencies but more wasted work than pessimistic

- Granularity of conflict detection is important design property
  - Fine granularity makes conflict detection slow, *e.g.*, word granularity
  - Coarse granularity makes conflict detection report false conflicts, *e.g.*, page level granularity

**Fundamentals (3/3)**

Transaction nesting

- flat
- closed
- open

**Software transactional memory (STM)**

- Compiler and runtime operation, no hardware support
- High overhead
  - Every memory access must be tracked ⇒ extra memory traffic
  - Conflict detection is expensive
  - Typical real-world experimental results: 30–90% of time spent in STM, scalability far from linear
- Legacy code must be specifically considered
- However, flexible solution as no HW requirements
- Unbounded transactions are easily implemented
- Strong isolation is expensive
Annotations
STM compiler by Intel is presented in [20] and can be obtained from http://whatif.intel.com. Criticism on STM is found in [4].

Hardware-accelerated software transactional memory (HASTM)

- Common bottleneck, i.e., memory access tracking and conflict detection, is accelerated by employing mark bits in the cache
- Almost HTM speeds are claimed
- Approach by Intel

Annotations
Hardware-accelerated STM is presented in [18]. In that work, accelerating the bottlenecks of STM with simple best-effort hardware support is considered. They claim almost the speeds of unbounded HTM. This work could hint on Intel’s future directions on transactional memories.

Hardware transactional memory (HTM)

- Hardware support for providing atomicity, versioning and conflict detection
- Versioning typically (but not always) implemented in data cache using existing cache coherency protocols for conflict detection ⇒ almost 0-overhead
- Transactions are far from unbounded both in memory footprint, execution time, and nesting
  - Albeit resource virtualization can overcome hardware limitations (compare to memory virtualization)
- Interrupts, context switches and other irregularities can cause false aborts

Annotations
Important real-world ISA discussion is found in [13], which is a more holistic approach. However, in Sun Rock processor the ISA extension (Sec. 4) is much smaller. But then again, Rock is best-effort only. Different approach to HTM is LogTM-SE, which does not rely on caches for implementation. Instead, it uses signature techniques. LogTM-SE provides unbounded transactional memory by utilizing virtualization techniques. [22]

Hybrid transactional memory (HyTM)

- Use HTM but fallback to STM when HW limits are reached
- HTM mode incurs some overhead compared to pure HTM, as checks must be made whether HW operation is safe
- Typical overhead around 10–20% compared to pure HTM
- Much faster than pure STM, but without HW limits
- Most transactions are small enough for HTM, only few of them fallback to STM
- Approach by Sun
HyTM is initially presented in [6], although some previous considerations exist. For a wrap-up of HyTM techniques utilized by Sun in Rock research can be found in [8] and its references. Perhaps the biggest performance problem in HyTM is that the hardware transactions must always check whether possibly conflicting software transactions are present. Speeding this up by utilizing memory protection is considered in [3]. It is also worthwhile to note that HyTM requires that there must exist HTM and STM version of every piece of software which might be run inside a transaction.

To conclude considerations of different implementations, phased transactional memory (PhTM) is an attempt to bring the best of many worlds. PhTM can switch between multiple implementation strategies, e.g., pure HTM to HyTM, based on the current workload [12].

Coping with legacy software

- Characteristics of legacy code:
  - Code using locks to synchronize critical sections
  - STM: Code which is not produced by STM compiler, i.e., memory accesses are not instrumented
- Workarounds:
  - Critical sections: convert into transactions by using speculative lock elision
  - Memory accesses: apply dynamic binary translation to instrument memory accesses
- Support for legacy code is important if existing libraries are to be used inside transactions!

Annotations

Transactional lock elision (TLE) is a speculative lock elision technique, implemented by utilizing transactions [17]. TLE is also used in works by Sun to speculatively execute lock-synchronized blocks in Java and C++ [8].

Dynamic Binary Translation techniques for transactional lock elision and instrumenting memory accesses is discussed in [21].

4 TM in Sun Rock processor

Section outline

- Transactional memory of Sun Rock processor
  - Sun Rock processor overview
  - Transactional memory implementation
  - HTM ISA
  - Applications

This is preproduction information, details are subject to change.
Sun Rock processor overview (1/2)

![Sun Rock processor diagram](http://www.opensparc.net/pubs/preszo/08/RockHotChips.pdf)

Sun Rock processor overview (2/2)

- Rock is next generation SPARC processor
- 16-core design, organized in 4x4 groups
- L1 ICache (32kB) per core group and L1 DCache (32kB) per core pair, on-chip 4x512kB L2-cache
- Each core executes 2 threads of software and 1 or 2 (configurable) speculative “scout” threads
- Transactional memory support
- 321M transistors, 65nm process, 250W @ 2.1GHz
- General availability in 2009H2

Annotations

Some sources for information:

- publications [19, 7, 8]

Note that there is already at least two revisions on Rock. Generally, material released in 2008 refers to R1 and material released in 2009 refers to R2. Notably, HTM implementation has changed a bit.
TM in Rock

- Lazy versioning:
  - Speculation bits in L1 DCache to track memory accesses, contain also modified data
  - 16 or 32-entry commit buffer containing list of modified cache lines. Buffer size depends on scout thread configuration
  - Modified lines flushed to main memory at commit
  - Abort simply discards commit log and modified cache lines

- Optimistic conflict detection
  - Invalidated lines abort transaction
  - Cache line granularity
  - Based on existing cache coherency protocols

- Best effort only:
  - Interrupts, exceptions, tlb misses, branch speculation misses abort ongoing transaction
  - Also “difficult” instructions such as some common procedure entry/epilogue instructions and div-family

ISA support

- Basically, three instructions
  - chkpt <fail_pc> — start transaction and specify abort address
  - commit — commit transaction
  - rd %cps, <dest_reg> — read transaction abort status

- Contention management and retry/fallback policies are implemented in software

Annotations

The Rock TM instruction set architecture is explained in [14] (Rock R1), but see also [8] for some changes in Rock R2.

Example applications

- Efficient synchronization primitive implementations
- Atomic container updates
- Speculative execution of restricted critical sections
- Implementing new synchronization primitives, such as double-CAS
- HTM part for hybrid HTM/STM implementations

Annotations

More application considerations with simulated results are found in [7].
Some published experimental results (1/3)

![Graphs](image)

Figure 1. HashTable with 50% inserts, 50% deletes. (a) key range 256 (b) key range 128,000.

Source: Dice et al: Early Experience with a Commercial Hardware Transactional Memory Implementation, ASPLOS’09.

© Sun Microsystems, Inc.

Some published experimental results (2/3)

![Graphs](image)

Figure 2. Red-Black Tree. (a) 128 keys, 100% reads, keyrange=[0,128) (b) 2048 keys, 96% reads, 2% inserts, 2% deletes.

Source: Dice et al: Early Experience with a Commercial Hardware Transactional Memory Implementation, ASPLOS’09.

© Sun Microsystems, Inc.

Some published experimental results (3/3)

![Graphs](image)

Figure 3. (a) TLE in C++ with STL vector (b) TLE in Java with HashTable.

Source: Dice et al: Early Experience with a Commercial Hardware Transactional Memory Implementation, ASPLOS’09.

© Sun Microsystems, Inc.

11
References


