Transactifying Apache’s Cache Module

H. Eran  O. Lutzky  Z. Guz  I. Keidar

Department of Electrical Engineering
Technion – Israel Institute of Technology

SYSTOR 2009 – The Israeli Experimental Systems Conference
1. Introduction
   - Why legacy applications are important
   - Previous STM benchmarks

2. Transactification Process
   - The need for STM compilers
   - Transactification Challenges

3. Results

4. Summary
The shift to multicore machines challenges software developers to exploit parallelism. Transactional Memory is one approach to make this easier.

Our Goals:

- Transactifying a large-scale legacy application.
- Creating a benchmark for STM systems.
The shift to multicore machines challenges software developers to exploit parallelism. Transactional Memory is one approach to make this easier.

Our Goals:

- Transactifying a large-scale legacy application.
- Creating a benchmark for STM systems.
Why Apache?

- Large-scale
  (∼340,000 lines of code).
- Popular
- Already parallel

And why Apache’s Cache module?
- One of the points of interaction between Apache’s worker threads.
- Well encapsulated.
- Currently implemented using one big lock.
Why Apache?

- Large-scale
  (~340,000 lines of code).
- Popular
- Already parallel

And why Apache’s Cache module?
- One of the points of interaction between Apache’s worker threads.
- Well encapsulated.
- Currently implemented using one big lock.
Previous work

- Concurrent data structures (e.g. red-black trees and skip lists.)
- STMBench7 – Measures operations on a more complex yet still artificial object graph.
- STAMP – Standford Transactional Applications for Multi-Processing: A collection of transactified scientific algorithms.
Transactifying C Programs
Library-based STM or compiler-based

Originally: Library-based
- Transactions delimited by special function calls.
- Access to shared memory through function calls.
- Manual handling of function calls inside transactions.

Too cumbersome for legacy code.

Recently: Compiler-based
- Syntactic support for transactions. (e.g. __tm_atomic blocks).
- Compiler automatic wrapping of access to shared memory.
- Nested function calls are either handled automatically, or by special attributes on declaration.

Haggai Eran, Ohad Lutzky, Zvika Guz, Idit Keidar

Transactifying Apache’s Cache Module
Transactifying C Programs
Library-based STM or compiler-based

**Originally: Library-based**
- Transactions delimited by special function calls.
- Access to shared memory through function calls.
- Manual handling of function calls inside transactions.

Too cumbersome for legacy code.

**Recently: Compiler-based**
- Syntactic support for transactions. (e.g. \_\_tm\_atomic blocks).
- Compiler automatic wrapping of access to shared memory.
- Nested function calls are either handled automatically, or by special attributes on declaration.
Which STM to use?

- **Tanger**
  - Open source
  - LLVM compiler extension
  - Supports tinySTM and other STM systems.
  - The version we used had problems with transactifying only a small part of the code base.

- Intel STM Compiler
  - Experimental version of Intel's ICC
  - Proprietary STM system.
  - Has published interface for other STM systems.
Which STM to use?

- **TANGER**
  - Open source
  - LLVM compiler extension
  - Supports tinySTM and other STM systems.
  - The version we used had problems with transactifying only a small part of the code base.

- **Intel STM Compiler**
  - Experimental version of Intel's ICC
  - Proprietary STM system.
  - Has published interface for other STM systems.
What to transactify

1. Convert mutex critical sections into transactions.
2. Wrapping atomic instructions inside transactions.
3. Decorate functions with Intel’s `tm_callable` attribute.
What to transactify

1. Convert mutex critical sections into transactions.
2. Wrapping atomic instructions inside transactions.
3. Decorate functions with Intel’s `tm_callable` attribute.

Example

1. `atomic_dec32(&obj→refcount);`

   1. Begin transaction
   2. ...
   3. `atomic_dec32(&obj→refcount);`
   4. ...
   5. End transaction
What to transactify

1. Convert mutex critical sections into transactions.
2. Wrapping atomic instructions inside transactions.
3. Decorate functions with Intel’s `tm_callable` attribute.

Example

1. Begin transaction
2. \(--\text{obj} \rightarrow \text{refcount};\)
3. End transaction

1. Begin transaction
2. \(\ldots\)
3. \(--\text{obj} \rightarrow \text{refcount};\)
4. \(\ldots\)
5. End transaction
What to transactify

1. Convert mutex critical sections into transactions.
2. Wrapping atomic instructions inside transactions.
3. Decorate functions with Intel’s `tm_callable` attribute.
Step 1: Convert mutex critical sections into transactions.

Get an object from the cache

1. **Mutex lock**
2. obj ← find key in cache
3. if obj found
   1. increment reference count on obj
   2. register obj for reference count decrementation later.
4. **Mutex unlock**
Defining Atomic Blocks

An interesting example

Step 1: Convert mutex critical sections into transactions.

Get an object from the cache

1. Begin transaction
2. \( \text{obj} \leftarrow \text{find key in cache} \)
3. if \( \text{obj} \) found
   1. increment reference count on \( \text{obj} \)
   2. register \( \text{obj} \) for reference count decrementation later.
4. End transaction
Defining Atomic Blocks
An interesting example

Might not be optimal:

Get an object from the cache

1. Begin transaction
2. obj ← find key in cache
3. if obj found
   1. increment reference count on obj
   2. register obj for reference count decrementation later.
4. End transaction
Provided registration is local to the current thread:

Get an object from the cache

1. Begin transaction
2. \texttt{obj \leftarrow find key in cache}
3. if \texttt{obj} found
   1. increment reference count on \texttt{obj}
4. End transaction
5. if \texttt{obj} found
   1. register \texttt{obj} for reference count decrementation later.
Commit Handlers

- Pieces of code a transaction requests to be run on commit.
- Can be used in our scenario to clean up the code.
Commit Handlers

Example

register_dec(obj)

register obj for reference count decrementation later.

Get an object from the cache

1. Begin transaction
2. obj ← find key in cache
3. if obj found
   1. increment reference count on obj
   2. Register commit handler (&register_dec, obj)
4. End transaction
In languages that support closures (e.g. ML, Smalltalk, Java’s inner classes), the use of commit handlers for our purpose would be much cleaner.

Get an object from the cache

1. Begin transaction
2. obj ← find key in cache
3. if obj found
   1. increment reference count on obj
   2. On commit: Register obj for reference count decrementation later.
4. End transaction
In languages that support closures (e.g. ML, Smalltalk, Java’s inner classes), the use of commit handlers for our purpose would be much cleaner.

Get an object from the cache

1. Begin transaction
2. \texttt{obj} ← find key in cache
3. if \texttt{obj} found
   1. increment reference count on \texttt{obj}
   2. \texttt{On commit}: Register \texttt{obj} for reference count decrementation later.
4. End transaction
| **Client** | The Siege HTTP load testing tool. |
| **Workload** | The set of unix man-pages, served using the `man2html` CGI program. The program uncompressed the man-pages and rendered them to HTML. |
| **Distribution** | Request files by Zipf distribution, whose $s$ parameter determines the level of locality in the requests. |
| **Setup** | Two 32-core machines (8-processors $\times$ quad core), connected by Gigabit ethernet, with 2.3GHz AMD Opteron processors and 126GB of RAM each. |
| **Experiments** | no-cache, no-transactions, transactified |
STM Disadvantage:
- Incurs an overhead for each read/write inside a transaction.

STM Advantage:
- Conflict only when same memory is accessed, not due to the coarse-grained lock.
**Results – Requests per Second**

$s = 0.1$

![Graph showing requests per second vs. number of cores. The graph includes lines for no-cache, no-transactions, and transactified scenarios.](image)

Very low locality. Cache not effective. STM penalty high.
Results – Requests per Second

\[ s = 1 \]

Medium locality. Cache is an improvement. STM incurs penalty.
Results – Requests per Second

$s = 2$

High locality. Cache is vital. STM version works best.

Haggai Eran, Ohad Lutzky, Zvika Guz, Idit Keidar

Transactifying Apache’s Cache Module
We started with looking how is it to transactify a large legacy application.

Our lessons:

- Choose Compiler-based STMs.
- Encapsulation support is important.
- Commit handlers can simplify code changes.
- Real-world applications are challenging and important to work on.
Questions?