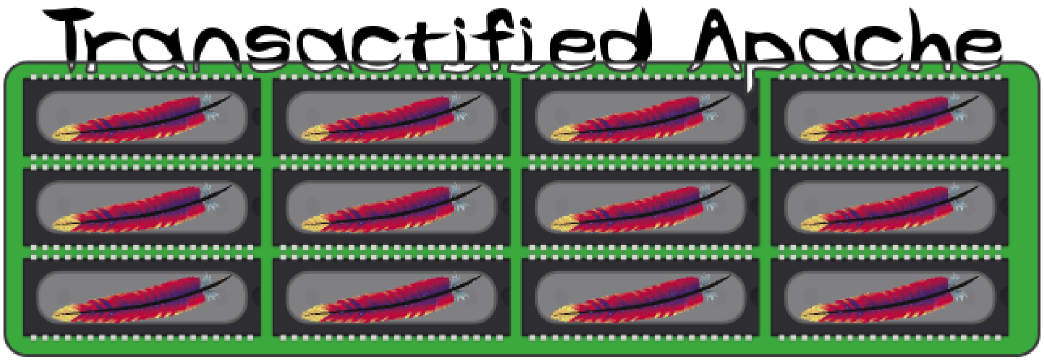
Transactional Memory Evaluation using Apache Webserver - Project Report

Haggai Eran, Ohad Lutzky
Advisor: Zvika Guz, Idit Keidar



Abstract

Transactional Memory systems attempt to give multiprocessor programmers the ease of use of course-grained locks with the performance scalability of fine-grained locks. With the development of these systems a need for performance evaluation methods has emerged. This project offers a benchmark for transactional memory systems based on the popular Apache webserver. Results of running it with Intel's transactional memory manager have shown comparable performance to locking.

We would like to thank Torvald Reigel from Dresden University of Technology, who offerred us great help with the Tanger tool, and even though we didn't eventually choose this tool, we appreciate the effort of his help.

Contents

List of Figures

1 Introduction

1.1 Transactional Memory

Transactional memory has emerged recently as an alternative synchronization method for parallel processing. A transaction is defined as an series of read and write operations and computations that should be executed atomically. The transactional memory system executes each transaction speculatively, without waiting for locks, and detects conflicts where two transactions might change global data in a way that distrubs atomicity, and *abort* one of the transactions, discarding all of its changes. If no such collision occurs, the transaction can *commit*, and its changes become visible to all other transactions.

Many implementations for transactional memory, both in hardware had been suggested. Software transactional memory (STM) works by managing the transactions and all their accesses to shared memory in software. Some implementations in hardware has been offerred, that take advantage of the cache coherency mechanism to provide a faster mechanism for detecting collisions and storing temporary transaction data. This project offers a way to evaluate performance of software transactional memory systems.

1.2 Project Goal

Since transactional memory is a rather new technology, without many working applications, there has been a need for an evaluation tool to compare performance of various implementations. In the past researchers used a simple toy applications that would manipulate a simple shared data structure as a benchmark. *STAMP*[3], a more comprehensive benchmarking suite have been published, which contains several algorithms, but still running out of the context of a real application.

The goal of this project was to create a benchmark for software transactional memory systems based on a real-world application, and examining the issues of working using STM with a large scale application.

2 Design

2.1 STM Library and Tools

2.1.1 Library or Compiler

C and C++ STM systems divide into two kinds: Library based and compiler based. Library based STMs are built as a C library. Every transaction begins with a call into the library, and commits by another call. All reads and writes to global variables must be done through special library functions when in a transaction. This requires a great amount of work for converting an application to use STM. Not only accesses to global memory in the function that started the transaction must be converted, but also any access from any function being called from this function.

In contrast, compiler-based STM use a specialized compiler, which has extended syntax for transactional memory atomic blocks. The compiler can then automatically convert memory accesses inside transactions into calls to the underlying library, a process sometimes referred as *transactifycation.*

2.1.2 Tanger

The Tanger[4] transctifying compiler is an open-source academic compiler extension for LLVM[8], an extensible compiler framework. Tanger aims at creating a transactifying compiler that is independant of the STM system used. It works with the tinySTM[5] library, but can easily be extended to use other STM libraries by writing a simple plugin.

Tanger creates a transactified version of each function in a compilation unit. Every function call inside a transaction is then converted to a call to the new version. This method is a major disadvantage when working on a large application. Many functions do not need a transactified version and this causes uneeded work for the compiler and the linker. Moreover, sometimes the transactifyication might fail because of calls to functions whose source is not available and cannot be transactified. This can cause the entire build process to fail, where in fact the code can be transactified without any error.

2.1.3 Intel C++ STM Compiler Prototype

Intel has published[10] an experimental STM compiler based on their industrial compiler ICC. It solved the above problem by adding some new function attributes to the language that tell the compiler which functions need to be transactified. The attribute `tm_callable` tells the compiler that a transactified version of the function will be needed. This way only functions that are required inside transactions can be marked as `tm_callable` and be transactified.

ICC works with Intel's own transactional memory manager. It has the disadvantage of being a closed source commercial application, and up until late phases of our project, didn't enable switching to different STM libraries. Recently Intel have published their ABI[6], allowing other libraries to be used similarly to tanger.

We chose working with ICC since its selective transactifycation ability was crucial to working with a large application such as Apache.

2.1.4 GCC for Transactional Memory

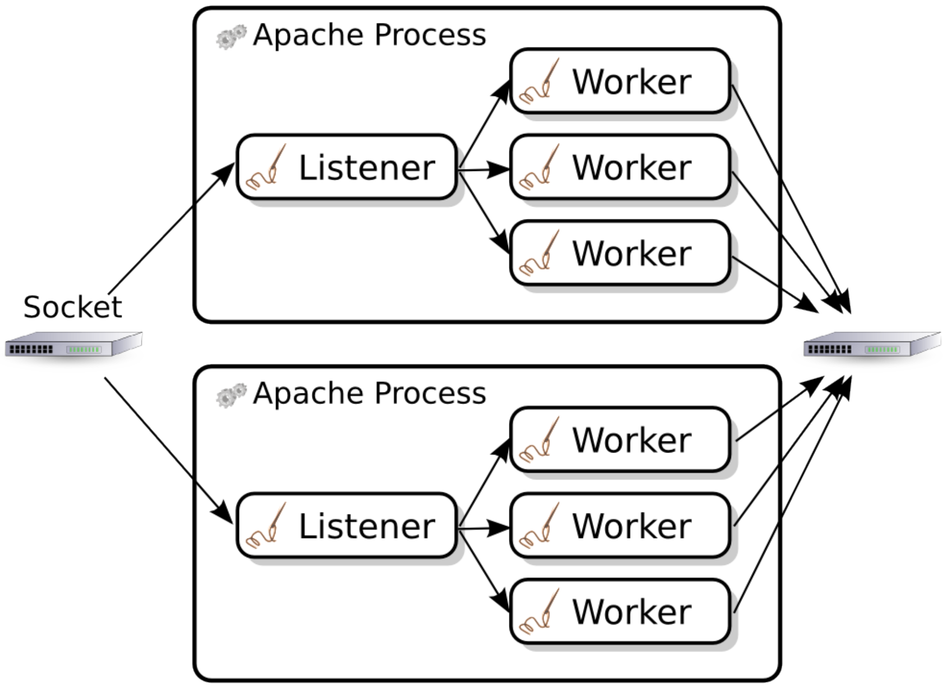
An extension to the Gnu Compiler Collection (GCC) is being developed[2] to enable transactional memory support for GCC. It is intended to work with tinySTM, but being open source, other STM systems will probably be ported too. The syntax of the C/C++ language extensions is designed to be compatible with ICC. This means that our project will probably be compilable under GCC with this extension, without much modification.

2.2 Apache Webserver

2.2.1 Introduction

Apache[11] HTTP server is a popular web server application written in C. It supports working on multiprocessor machines with several multi-processing modules (MPMs) each offering a different strategy for handling requests and distributing the work. The most popular threaded MPM is the *worker* MPM, which works by running multiple worker-threads under several processes, each thread handles a single request at a time. In each such process there are several worker threads, and also a listener thread that fetches incoming requests and dispatches them to the available workers.

Fig. 1: Apache worker MPM architecture



2.2.2 Cache module

There are not many points of interaction between the worker threads themselves, where transactional memory can be used. One such place is Apache's memory cache implemented by the `mod_mem_cache`[14] module. This module enables the workers of each process to share a cache of recently served requests. A new request can be served from the memory cache, and save the time required to access the disk and generate the requested page. Since the cache is shared between multiple threads, it is synchronized by a single lock, therefore a good candidate for converting into transactional memory.

Apache's cache is implemented with a couple of modules. The first, `mod_cache`[12], implements the logic related to caching. It tests the metadata of each requests to see if it can be supplied from the cache, according to the request's HTTP headers and the system configuration. It uses one of the underlying cache implementation modules, `mod_mem_cache` or `mod_disk_cache[13]` to do the actual caching.

The `mod_mem_cache` module implements a memory cache using a shared hash table and priority queue. The key to the hash table is the URL of the request, converted into a canonical form. The cache is limited both by size and by the number of elements, and by memory size, so on insertion, sometimes lower priority entries are removed from the cache. The priority is deterimined by one of two algorithms: LRU, removing the least recently used entries first, and GDSF (Greedy Dual Size Frequency) assigning score to entries based on the cost of a cache miss, and the entry size.

2.2.3 Conversion Process

The conversion process included converting critical sections protected by the cache module's lock into atomic blocks, and decorating required functions as `tm_callable`. The module had used atomic instructions for some memory accesses, and these were converted to full transactions in atomic blocks, so that collisions with these accesses will be detected.

Some transactions, after conversion contained code that belonged with the transactions, but didn't need neccessarily to run atomically with the transaction. An example might be a transaction removing an object from the cache, and freeing its memory. While the removal operation must be protected inside a transaction, as it is using the shared memory structure of the cache, the memory release can happen any time later, since no other thread can point to the removed object after it had been removed from the cache.

For lock based systems, having the memory release as part of the critical section might cause a thread to hold the critical section a little longer than needed, but doesn't cause any problems other than that. On transactional memory systems, having accesses to other memory structures such as those required by memory management might cause collisions with other threads, thus slowing down the system in a similar way. In addition, the cleanup functions need to be transactified, which requires additional work both from the programmer and the compiler.

In our case, we chose not to transactify such functions, but instead remove them from the atomic section, and execute them after the transaction had committed. Although this requires some changes to the code, the changes are limited to the call-site, and need not modify any of the called libraries.

3 Evaluation

The transactified web server was evaluated using *Siege*[7], an HTTP load testing tool. The server was loaded with the set of UNIX man-pages - a set of small textual files typical of some web sites. Each page was served using the *man2html*[1] program, uncompressed and converted into HTML, to make sure the serving of files requires enough computational resources to make the use of cache worthwhile.

3.0.4 man2html

The man2html program is a Common Gateway Interface (CGI) program that serves unix manual (man) pages on internet sites. The pages are usually stores compressed in gzip format, and formatted using the troff format. The program receives a request for a man page from the webserver, uncompresses the required file and converts it to HTML. As every CGI program it outputs the result with relevant HTTP headers.

The default caching policy of apache forbids caching dynamically generated pages such as those of man2html, unless the HTTP headers of the resulting page clearly specify otherwise. To make caching of the man2html pages possible, we modified man2html to output such headers, specifying the output can be cached for one hour.

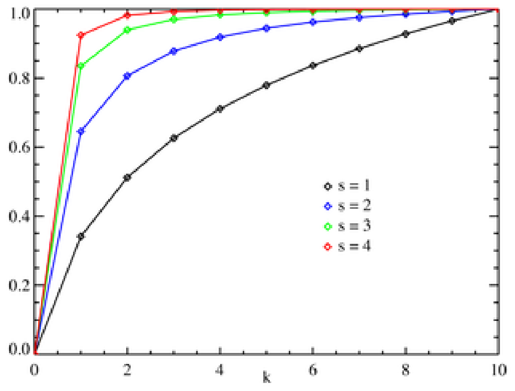
3.0.5 Zipf distribution

The pages were requested randomly according to Zipf distribution , whose paramter s determines how frequently the most popular pages were visited, thus controlling the amount of locality in the requests.

3.0.6 Experiment Hardware

The experiments were done with two computers connected using Gigabit ethernet. The machine running the server was a 4 processors SMP of dual core 2.66GHz Xeons with 8GB of RAM, and the client machine was a 2 processors SMP of quad core 2.33Ghz E5410 Xeons with 8GB of RAM.

Fig. 2: Zipf Distribution



3.0.7 Results

We compared the average latency and request throughput when running on different number of cores, and with different s values. For every graph there are three experiments comparing the results of an Apache server running without a cache, a cached version without our transactional modifications, and the transactified version.

Fig. 3: $s = 1$

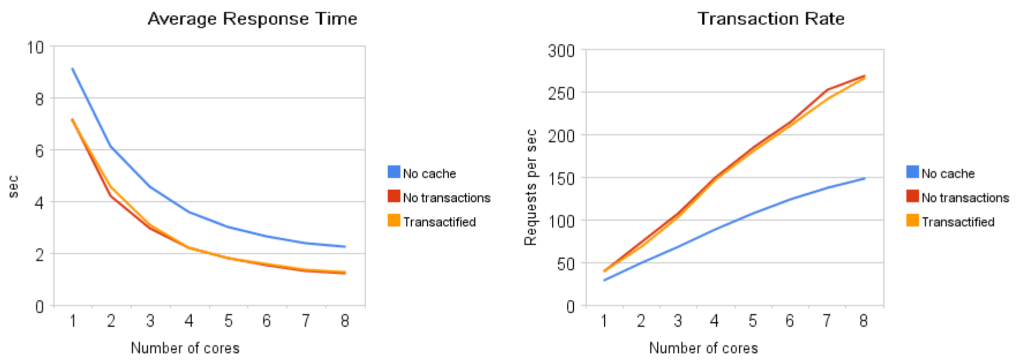


Fig. 4: $s = 2$

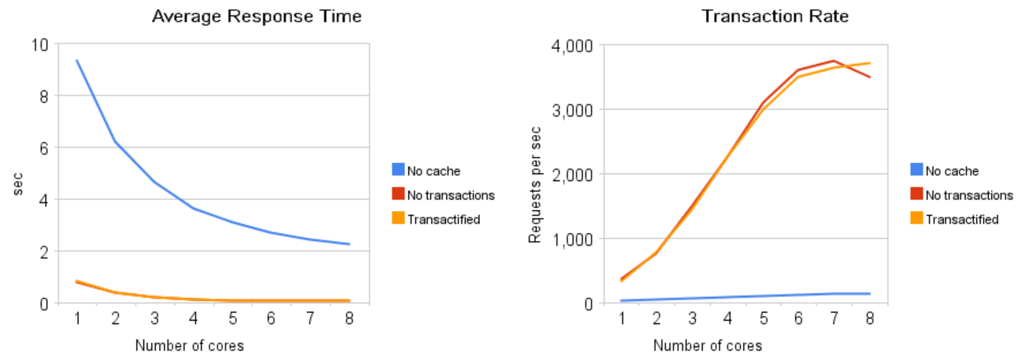
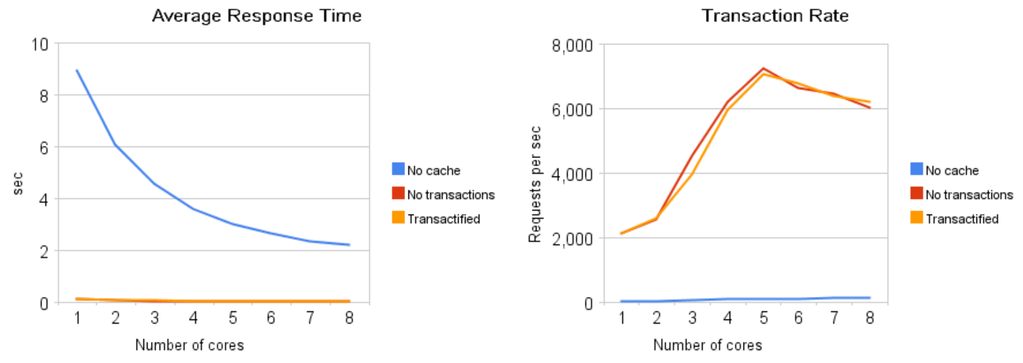


Fig. 5: $s = 3$



4 Conclusions

4.1 Results

The results show the STM version achieving comparable performance to that of the original version, and both improve the performance in a significant way with regard to the uncached version. With higher locality, and a large number of processors, there is a decrease in performance due to the higher contention on the cache.

4.2 Commit Handlers

The pattern we described in section 2.2.3, of moving transactions after-effects out of the transactions could be made more simple for the programmer by the addition of compiler or library support. Commit handlers allow a transaction to register actions to be executed only after the transaction has committed. Using such a construct would allow the modifications to current code to be simpler and make conversion to transactional memory easier.

This mechanism, along with abort and violation handlers, was suggested in [9].

Commit handlers are described there as a mechanism that allows finalization of tasks, for instance, a transactional system call such as write to file might have its permanent side effects be executed in a commit handler. Abort and violation handlers are user functions that are called when a transaction is aborted by the user, or becase of a conflict, respectively. They allow transactions with permanent but reversible side-effects to undo their effects.

To summarise, our experience has shown that these handlers aren't only needed to enhance the transactional memory system's features, but can also make conversion of legacy code easier.

4.3 Further Research

There are many STM systems currently available, and one immediate direction would be to compare them using this benchmark. This would require writing plugins for any such system to match Intel's TM ABI.

In addition, there are other applications that might be interesting as transactional memory applications, following the methods we used.

References

[1] Man2html package source code. Available from World Wide Web: `http://packages.ubuntu.com/source/intrepid/man2html`.

[2] Albert Cohen. GCC for Transactional Memory. HiPEAC, 7 June 2008. Available from World Wide Web: `http://www.hipeac.net/node/2419`.

[3] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. Stamp: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[4] Pascal Felber, Christof Fetzer, Ulrich Müller, Torvald Riegel, Martin Süßkraut, and Heiko Sturzrehm. Transactifying applications using an open compiler framework. In *TRANSACT*, August 2007.

[5] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2008.

[6] Intel. Transactional memory ABI. Available from World Wide Web: `http://software.intel.com/file/8097`.

[7] Jeffrey Fulmer. Siege HTTP regression testing and benchmarking utility. Available from World Wide Web: `http://www.joedog.org/JoeDog/Siege`.

[8] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[9] A. Mcdonald, J. Chung, B. Carlstrom, C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural semantics for practical transactional memory, 2006. Available from World Wide Web: `http://citeseer.ist.psu.edu/mcdonald06architectural.html`.

[10] Yang Ni, Adam Welc, Ali-Reza Adl-Tabatabai, Moshe Bach, Sion Berkowits, James Cownie, Robert Geva, Sergey Kozhukow, Ravi Narayanaswamy, Jeffrey Olivier, Serguei Preis, Bratin Saha, Ady Tal,

and Xinmin Tian. Design and implementation of transactional constructs for c/c++. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 195–212, New York, NY, USA, 2008. ACM.

[11] The Apache Software Foundation. Apache HTTP Server Project. Available from World Wide Web: `http://httpd.apache.org/`.

[12] The Apache Software Foundation. Apache's mod_cache Memory Cache Module. Available from World Wide Web: `http://httpd.apache.org/docs/2.2/mod/mod_cache.html`.

[13] The Apache Software Foundation. Apache's mod_disk_cache Memory Cache Module. Available from World Wide Web: `http://httpd.apache.org/docs/2.2/mod/mod_disk_cache.html`.

[14] The Apache Software Foundation. Apache's mod_mem_cache Memory Cache Module. Available from World Wide Web: `http://httpd.apache.org/docs/2.2/mod/mod_mem_cache.html`.

Appendices

A Developer Guide

A.1 Directory Structure

The project main directory contains the following subdirectories.

httpd-2.2.x The modified version of Apache, configured to use transactional memory

httpd-2.2.x.no-transactions Another copy of the modified version of Apache, configured not to use transactional memory.

cache-conf An Apache configuration folder configured to use the man2html workload, with mod_mem_cache enabled.

no-cache-conf An Apache configuration folder configured to use the man2html workload, with mod_mem_cache disabled.

doc The documentation of the project, including this document, and the project's presentation.

man The man program distribution, including the man2html cgi script.

man2html symbolic link into **man/man2html/scripts**, reffered to by Apache's configuration folders **cache-conf** and **no-cache-conf**.

test Scripts needed for running the benchmarks.

- runall_siege.py A script that runs all the needed experiments one by one. (See section A.4.1 for details).
- zipf_distrib.py A utility script that takes a list of URLs as an input, and outputs a list where every url appears a number of times according to Zipf distribution. (See section A.3.1 below).
- url-lists A subdirectory with already generated URL lists.

A.2 Server Side Installation

The following applies to both versions of apache.

- Install Intel STM C++ Compiler (`http://tinyurl.com/66hkyt`)

- Install the *schedtool* utility for limiting the number of processors Apache will use (`http://freshmeat.net/projects/schedtool/?topic_id=136`)
- Unpack apr and apr-util packages into `<httpd>srclib/apr` and `<httpd>/srclib/apr-util` respectively. (Where `<httpd>` is the path of the version you are compiling).
- Configure apache using the parameters given in the `configure-cmd` file.
- Build using the `make` command.
- Configure and install the man2html program as described in `man/INSTALL`.

A.3 Client Side Installation

- Unpack, configure and install siege, from the `siege-2.67.tar.gz` file. Further information is in the package in the `INSTALL` and `README` files.
- Copy the URL lists from `test/url-lists` to the client machine.

A.3.1 URL Lists

This section elaborates the generation of URL lists for the client machines.

- The starting point for the process is a file containing a list of unique URLs, each in its own line.
- Using the `test/zipf_distrib.py` program, a new list is created that is randomly distributed according to the Zipf distribution. The command line for doing that should be:

  ```
  zipf_distrib.py s length < input-file > output-file
  ```

  Where

  s is the *s* parameter of the Zipf distribution

  length is the number of URLs in the output.

  input-file is the input list of URLs.

  output-file is the output result.

A.4 Execution

A.4.1 Test Configuration

The `test/runall_siege.py` script is intended to run from the server machine, and connect to the client by ssh. To avoid typing the password for every connection it is recommended to set up a public-key login to the client machine.

Before running, the following fields must be set in the script:

CLIENT_HOST Client host name.

MAIN_DIR Directory where the project is placed.

SCHED_TOOL Path of the schedtool binary. (Can be just `schedtool` in case its on the system PATH).

URL_LISTS A list of filenames on the client host of the URL list files (Usually there is one for every s value needed in the experiment).

MIN_CORES,MAX_CORES The minimum and maximum number of cores to test.

A.4.2 Execution

Set the siege output file in `~/.siegerc` on the client machine, by editing the *logfile* entry.

Run the `test/runall_siege.py` script, the results will be saved on the client machine in the chosen output file.