Lock vs. Lock-free Memory
Project proposal

Fahad Alduraibi   Aws Ahmad   Eman Elrifaei
Electrical and Computer Engineering
Southern Illinois University

1. Introduction

The CPU performance development history showed that it reached a point that exhausted current system resources where fixed performance rates no matter what increase in speed gain. And any demands to increase the clock speed will increase rates of system failure due to hardware limitations mainly memory bottleneck and other limitations e.g. limited size and increased temperature on the system. So the research trend went on the part considering the memory trying to decrease memory access time in order to enhance the overall performance of the system.

With the introduction of multi-core processors, the idea of incorporating transactions into the programming model used to write parallel programs became attractive. This approach, known as transactional memory, offers an alternative and better way to synchronize concurrent threads.

Locking techniques have been used to synchronize parallel tasks in order to protect concurrent access to shared data. However, some problems associated with locking like deadlock, convoying, and priority inversion made parallel programming very difficult. Transactional memory intends to replace the locking techniques to make parallel programming lock-free hence allow easier coding and efficient programs. The idea behind the transactional memory is the following [7]:

- Each transaction is a set of instructions executed by a single process, that do not interleave with instructions from another transaction, which ensures the transaction produces the same result as if no other transactions were executing concurrently (serialization).
- A transaction is executed atomically (either it completes successfully and commits its result in its entirety or it aborts).

Current researches are working on three types of transactional memory:

- Hardware Transactional Memory (HTM): Proposed some changes to the existing hardware design and introduced new instructions for accessing memory. Since it is implemented in hardware the performance is very high, however, there are some boundaries on the size of instructions inside a transaction [7].
- Software Transactional Memory (STM): Even though it cannot compete with the performance of HTM, STM has advantages in terms of applicability to existing machines and the flexibility in size of transactions [3].
- Hybrid Transactional Memory (HyTM): Combines HTM and STM together to take advantage of hardware performance and software flexibility and applicability to existing machines [1] [8].

Although transactional memory is not a parallel programming solution, it shifts much of the burden of synchronizing and coordinating parallel computations from a programmer to a compiler, runtime system, and hardware [9].

2. Related Work

Our work relies on using libraries to implement software transaction memory applications that will be used for the comparison with the lock-based applications.

2.1 Dynamic-STM

DSTM is a software implementation of transactional memory by Herlihy et al. and it is the first to support dynamic sized data structures. While a transaction is running, objects can be created dynamically. Earlier research has shown that DSTM outperforms locking on some data structure such as linked lists and red-black trees. The proposed API's have some functions that are used to commit or about a transaction. The state and accessed objects are stored in a TMObject.

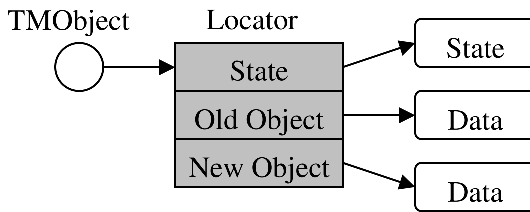

Figure 1. TMObject

The TMObject keeps track of transaction state (either Active, Committed, or Aborted) and object old data (will needed in case a transaction is to be aborted) and object new data (in case of transaction commit). If two transactions open the same object at the same time, a synchronization conflict occurs, and one of the conflicting transactions must be aborted. The Contention Manager will decide which transaction to abort.

To reduce synchronization conflicts, an object can be opened in one of several access modes. An object opened in WRITE mode can be read or modified, while an object opened in READ mode can only be read. WRITE mode conflicts with both READ and WRITE modes, while READ mode conflicts only with WRITE. Another mode RELEASE indicates that the transaction may release the object before it commits.

Assume transaction A has modified and object X and committed (in the TMObject, the Old-Object will have the old value of X and New-Object will have the new value of X) then another transaction B want to access the same object so the Old-Object of transaction B will point and copy the data from the New-Object of transaction A. However, in case transaction A did not commit but instead it aborted then if transaction B want to access that object it will point and copy the data from the Old-Object of transaction A. [2][4]

2.2 High Performance STM

In order to enhance the performance of software transactional memory, Saha et al. proposed a new implementation called McRT-STM (High Performance Software Transactional Memory System for a Multi-Core Runtime) which supports advanced features like nested transactions with partial aborts, conditional signaling within a transaction, and object based conflict detection for C/C++ applications.

They described a software transactional memory (STM) system that is part of McRT-STM, which is an experimental Multi-Core RunTime. The McRT-STM was implemented using a number of novel algorithms, and supports advanced features such as nested transactions with partial aborts, conditional signaling within a transaction, and object based conflict detection for C/C++ applications. The McRT-STM exports interfaces that can be used from C/C++ programs directly or as a target for compilers translating higher level linguistic constructs.

In addition, the researchers presented a detailed performance analysis of various STM design tradeoffs such as pessimistic versus optimistic concurrency, undo logging versus write buffering, and cache line based versus object based conflict detection. Also in order to provide a baseline they compared the performance of the STM with that of fine-grained and coarse-grained locking using a number of concurrent data structures on a 16-processor SMP system. They also showed that their STM performance on a non-synthetic workload that is the Linux send mail application.

The results of the above study showed that the performance of STM is also application dependent, in some cases STM performed better than Locking, but the usual case is for lock to take less execution time than STM because of the well know overhear of STM logging mechanism [5].

3. Experimental Approach

The main object of the project is comparing the performance of the system running certain programs that require multi-threading synchronization using Transactional memory and lock based synchronization.

The work will start by understand the SXM software transactional memory system, and see the way it works so an equivalent or similar system but using locks for synchronization.

The benchmark will be programmed in C# language, because we want to compare it to a benchmark for Software Transactional memory (SXM) which is programmed in C# too, to be consistent and to get accurate results we need to program it in the same language. Also taking advantage of the C# language we can easily program and debug using the Microsoft Visual Studio Development Environment, codes can be easily debugged and traced [6].

The benchmark for locks should be able to almost the same parameters that the software transactional memory benchmark takes. These inputs are the number of threads, and the program name. The main output of the benchmark should be the execution time it takes to run the program. The same set of programs will be run on both SXM system and our benchmark, so a comparison of the performance can be meaningful, below is a block diagram representing the system.

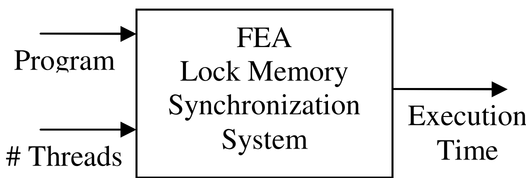

Figure 2. Block Diagram of Lock Memory Synchronization System.

We have four main programs that we are going to write and synchronize using C# locks, these programs are simulating the following shared data structures: Buffer, RBTree, List, and SkipList. These data structure are already written and simulated using the SXM software Transactional Memory. After writing those programs we give them to the system we are building to run them and calculate the execution time for the parameters we give to it.

The procedure to calculate the execution time is simplifies in the flow chart (Figure 3) that describes the functionality of our benchmark.

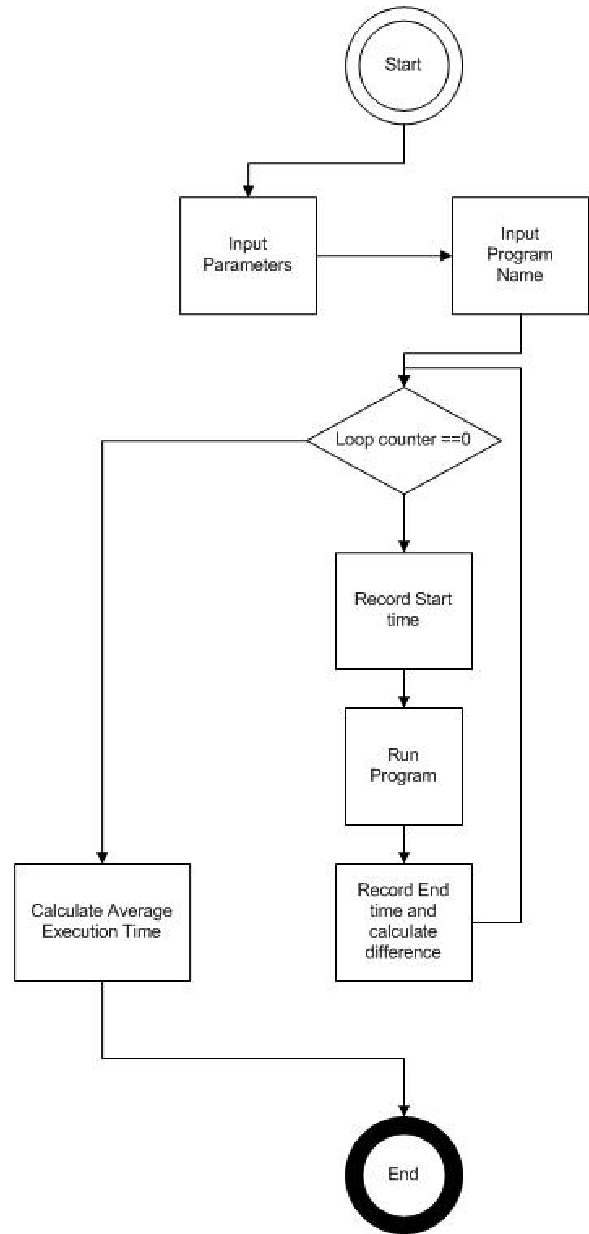

Figure 3. Procedure to calculate the execution time.

4. Expected Result

As mentioned in the experimental approach, we are going to run four programs and calculate the execution time for each program using Software Transactional Memory and Lock Memory for Synchronization.

The graph in Figure 4 represents the results we are expecting.

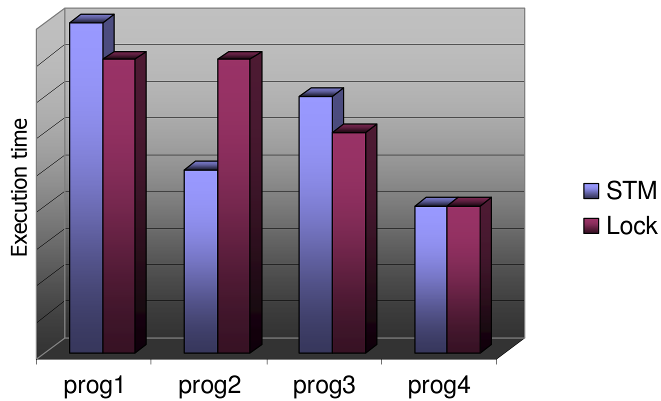

Figure 4. Expected results

As we can see from Figure 4 the results might vary depending on the program we are running. We are expecting to see the STM performing better that Lock and other cases were Lock would perform better.

We expect the dominant case to be lock performing better or slightly better than STM because of the overhead introduced by STM to log the actions the transaction takes in case of transaction abortion.

5. Summary

Software Transactional Memory is a promising technology to synchronize concurrent assess to shared data in Transactional Memory. Our project goal is to compare the performance of transactional memory library SXM built in C# language with the performance of lock memory system, which we will build in C# language too. The main inputs to the benchmark are the number of threads and the program name to calculate the execution time for. Four programs are going to be tested for performance; these programs represent the data structures (Buffer, RBTree, List, and Skip List).

The expected results for comparing the two models are expected to differ according to the application we are running, though the dominant result is expected to be for Lock to perform better or slightly better than STM, as mentioned above because of the overhead the Transactional memory logging introduces to the system, but in spite of this overhead Transactional memory represents the future for parallel programming because of the programming simplicity it presents to the programmer, and also the avoidance of commonly known lock programs like deadlock, and priority inversion.

6. References

[1] Hybrid Transactional Memory.
By: Sanjeev Kumar† Michael Chu‡ Christopher J. Hughes† Partha Kundu† Anthony Nguyen†
†Intel Labs, Santa Clara, CA ‡University of Michigan, Ann Arbor

[2] Herlihy, M., Luchangco, V., Moir, M., and Scherer, W. Software transactional memory for dynamic-sized data structures. In Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing (July 2003), pp. 92-101.

[3] Software Transactional Memory
By: Nir Shavit, Dan Touitou
Tel-Aviv University

[4] C. Cole and M. P. Herlihy. Snapshots and Software Transactional Memory. In Proceedings of Workshop on Concurrency and Synchronization in Java Programs, 2004.

[5] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. Cao Minh, and B. Hertzberg. A high performance software transactional memory system for a multi-core runtime. In PPoPP ’06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming, New York, NY, USA, March 2006. ACM
Press.

[6] “How to Write High-Performance C# Code" By: Jeff Varszegi, .NET Developer's Journal

[7] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural
support for lock-free data structures. In Proceedings of the 20th
Annual International Symposium on Computer Architecture, 1993.

[8] Damron, P., Fedorova, A., Lev, Y., Luchangco, V., Moir, M., Nussbaum, D. Hybrid transactional memory. In Proceedings of the

12th International Conference on Architectural Support for Programming Languages and Operating Systems. San Jose, CA (October).

[9] Transactional Memory
James R. Larus, Ravi Rajwar
Synthesis Lectures on Computer Architecture, 2006, Vol. 1, No. 1, Pages 1-226