2013

# A Study of Synchronization Mechanisms in a Distributed Memory Caching System

Trilok Vyas
*Lehigh University*

Recommended Citation

Vyas, Trilok, "A Study of Synchronization Mechanisms in a Distributed Memory Caching System" (2013). *Theses and Dissertations.* Paper 1095.

# A Study of Synchronization Mechanisms in a Distributed Memory Caching System

By
Trilok J. Vyas.

A Thesis
Presented to the Graduate and Research Committee
of Lehigh University
in Candidacy for the Degree of
Master of Science
in
Computer Engineering

Lehigh University
January, 2013

This thesis is accepted and approved in partial fulfillment of the requirements for the Master of Science.

______________________________
**Date**

______________________________
**Thesis Advisor: Michael F. Spear**

______________________________
**Chairperson of Department: Daniel P. Lopresti**

## ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor and mentor, Professor Michael Spear for a number of reasons.  Without his help and support, I could not have even imagined completing this research work.  Especially, I am grateful to him for always understanding and encouraging me during the most difficult times of the project.  I can certainly not overlook the contribution of Yujie Liu towards this project.  He has been of a great help in performing code-reviews, suggesting ways of improvement, as well as resolving my queries in the absence of Professor Spear.  I owe a debt of gratitude to Professor Wagh and Professor Korth for their sincere support in thesis project selection. I am also very thankful to Heidi Wegrzyn for all her help.  Last but not the least, I want to thank all my family members, friends, and well-wishers who supported me and co-operated with me in every possible manner.  A special thanks to Shivani for everything she has done for me.

## TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

## ABSTRACT

As the microprocessors are moving towards having more cores on a single chip (multi-core), the software programs that run on those chips are also increasingly becoming parallel/multi-threaded. At the heart of multi-threaded programming is the use of synchronization mechanisms to control access to the critical sections. There are various methods that can be employed to achieve this goal. Some are lock based, whereas others are not. The use of synchronization mechanism can affect the overall speed of a multi-threaded program. In our project, we studied the performances of various synchronization mechanisms like POSIX thread locks, TestAndSet, Oyama-locks, and Software-Transactional-Memory using an open source distributed memory caching system called memcached. After evaluating the performance of various benchmarks, we found that all the three lock based methods perform equally well at a high concurrency level. There are some other interesting observations as well which are mentioned here. Also mentioned are some of the limitations of our custom developed library routines for generating the benchmarks, and possible enhancements in the future along with other future work.

# Chapter 1: INTRODUCTION

Everyone would agree on the fact that: no other technology industry has grown more rapidly than Computer and Telecommunication industries within the past few decades. Since the advent of integrated circuit, the computer industry started flourishing at an ever increasing rate. In computer-hardware chip manufacturing domain, Moore's law continued to "*hold true for more than half a century".*[1] First there were single-core microprocessors, and then came dual core, quad-core, hexa-core, and so on. Even in the era of single-core microprocessors, some high-end systems like servers used to have multiple such CPUs on their motherboard thereby forming a multiprocessor machine. As the underlying hardware technologies evolved, so did the software running on top. The evolution of multitasking, multithreading, and similar terms is very nicely explained on Wikipedia website which is put together in the following quote:

> *"A computer programming method called multitasking, in which multiple tasks (also called processes) are performed during the same period of time started to become popular. As multitasking greatly improved the throughput of computers, programmers started to implement applications as sets of co-operating processes (e. g., one process gathering input data, one process processing input data, and one process writing out results on disk). This, however, required some tools to allow processes to efficiently exchange data. Threads were born from the idea that the most efficient way for cooperating processes to exchange data would be to share their entire memory space. Thus, threads are basically processes that run in the same memory context. Various concurrent computing techniques are used to avoid potential problems caused by multiple tasks (or threads) attempting to access the same (shared) resource at the same time*". [2]

The concurrent computing techniques mentioned in the quote are basically various mechanisms to achieve synchronization between threads trying to access a sensitive shared region of code (called critical-section). The study of some of these synchronization mechanisms forms the primary focus of our research work.

## 1.1: What is it all about?

Since the evolution of multi-threaded programming techniques, software programmers started to make use of it. The reason was of course, its advantages even in the time of single-core/single-processor systems. With the inception of multiprocessor/multi-core systems, multi-threaded programming has become an increasingly common practice. *"A multithreaded program (usually) runs faster on computer systems that have multiple CPUs, CPUs with multiple cores or across a cluster of machines— because the threads of the program naturally lend themselves to truly concurrent execution"*.[3] A key concept in a multi-threaded program when run on a multi-core/multiprocessor system is the synchronization mechanisms used to handle sensitive pieces of code called critical sections. *"A critical section is a piece of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution"*.[4] The performance of a multi-threaded program can be greatly influenced by the underlying synchronization mechanism, and this fact makes the basis of our project.

## 1.2: Synchronization Mechanisms

The process of accessing the critical-sections in a concurrent or multithreaded software program is handled by using a synchronization mechanism. One such method is using semaphores. Another one is a mutex lock, which is very similar to a (binary) semaphore in many aspects. Mutex lock seems to be one of the most popular and common methods for protecting critical sections of a code. From the following chapters, it can be seen that most of our project work was comprised of converting (POSIX) mutex locks to various other synchronization mechanism. Apart from these two, there are other mechanisms such as MCS-Queue [26] locks, Oyama-locks [11], Flat-Combining [12], TestAndSet/Test&TestAndSet [27] locks, and some others. These all are more or less similar in a way, since they all are based upon the concept of locking for achieving synchronization. There is an API support such as OpenMP™ [28] which is available for shared-memory multiprocessing/multi-threaded programming, which hides the details of the underlying synchronization mechanisms from the programmers. Unlike the rest, a mechanism called Software-Transactional-Memory (STM) is modeled based on the concept of database transactions. STM provides an interesting alternative to the traditional synchronization mechanisms, and is a unique candidate for benchmarking against the mutex-locks in our project. The other candidates are Test&TestAndSet/TestAndSet locks and Oyama-locks. We have provided further details about each of these mechanisms in the following chapters.

## 1.3: Project Goal

The goal of our project is, to study and compare various synchronization mechanisms in an open source high-performance multi-threaded software, by modifying the existing synchronization mechanism(s) present in the software. Memcached, *"a high-performance distributed memory object caching system"*[5], was chosen as our candidate for this project due to a number of reasons. The requirement to test various synchronization mechanisms led us to implement other techniques such as: Test&TestAndSet locks, Oyama-locks, as well as Software-Transactional-Memory version by modifying the original (open) source code of memcached. The prime goal of our project is neither memcached, nor locking mechanisms. It is the study of difference in performances between Pthread-mutex lock and various other synchronization mechanisms in memcached server.

**1.4: Memcached, what and why?**

*Memcached* in simple terms can be explained as follows. It is a memory-object caching system composed of at least one server and one client application. The client application(s) make requests to the server(s) for data objects (such as strings, numbers etc.) identified by unique keys for each object. The server stores the objects in a <key, value> format in the main-memory (RAM) of the machine it is running on. For a given request from the client, if the server finds the object in its memory using the supplied key, then it is considered a cache-hit and the data is returned immediately to the client. On the other hand if an entry is not found in the server for that key (a cache-miss), then usually the client queries a database system for the same object. Upon retrieving the data from the database system, generally it also issues a store command to the memcached server to store the object for a next retrieval. Thus, the client applications can save time of querying the database system, in cases when the objects are found in the memcached servers. A detailed description of memcached is provided in the following chapters.

There are some important reasons for making memcached as our software of choice. First of all it's an open-source software written in C language. Second, it is a multi-threaded software implemented using only POSIX Thread (Pthread) mutex locks for synchronization. Also, there are various benchmarking tools (especially memslap, a memcached client software) available for benchmarking the memcached server. The server running memcached under a heavy load can be run to saturate the CPU usage on the system on which it runs. In order, to modify the synchronization mechanisms one has to look only for the Pthread-mutex locks. Last and the most important reason for choosing memcached is that, it was a preferred choice for testing on a hardware based Transactional Memory system.[25] Even though memcached uses only mutex locks for guarding its critical-sections, there are different types of critical-sections. Some are very short lived, whereas others are a little more time consuming. On the other hand, there are some critical-sections which fit into producer-consumer scenario. Such critical-sections could have been guarded by using semaphores for better performance.

# Chapter 2: MEMCACHED

In this chapter, we explore memcached in some more detail. As mentioned before, *"Memcached is a free & open source, high-performance, distributed memory object caching system"*. [5] It is intended to use for speeding up dynamic web-applications which generally rely on the underlying database system for generating their content. Memcached is an in-memory key-value store for small chunks of arbitrary data (strings, objects) from results of database calls, API calls, or page rendering. Thus, memcached alleviates the load on the database by fulfilling the query request from the main memory, where the response of the query is usually cached. If and only if the content of the main-memory contains stale or no data (cache-miss), then the query request is forwarded to the database system. Thus, memcached system only helps in a high cache-hit scenario. On the other hand if almost every client request results in a memcached server cache-miss, then the performance will actually degrade instead of improving. There is a good amount of documentation that is available on the internet for various aspects of memcached system. Our work focuses on the synchronization related areas of the underlying multithreaded programming. However, in order to understand the synchronization related logic, it is helpful to know the basic internal design of the software. The next section explains the basic functionality of the software, followed by a section for some internal details of the server.

## 2.1: Functionality

Memcached is a distributed memory object caching system. It makes use of primary memory (RAM) of a computer to cache (store) frequently used (unmodified) data-objects of (usually) a web-application so that in case of a cache-hit, the datum can be found in the main-memory instead of the database. It allows any arbitrary format of the data to be stored, and uses (key, value) pair for each datum. A unique key of maximum 250 bytes in size indentifies a unique data-object/string. Memcached clients should be configured for proper operation of the system. The decision of sending a particular data-object to one of the servers in a group (if there is more than one server) is made by an algorithm inside the memcached-client. So in other words, memcached is a double level hashing system. The first or client-level hash function determines the exact-server where a particular data-object resides, if there is more than one. This decision is solely taken based on the content of the key (out of the key, value pair) using the logic in the client library. A memcached client library (libmemcached) implements *"A modular and consistent method of object distribution. Objects are stored on servers by hashing keys. The hash value maps the key to a particular server"*. [8] Once, the data-object reaches the designated server, it is stored in a second or server-level hash table on that particular server. The servers do not require having any knowledge of the other servers. Memcached API provides various features to insert/update and delete data from the memcached server-cache. The server stores the data in so called slab-pages, each of one Megabytes size by default (or they can be of some other custom size). There are various slab-classes for different data-object sizes. Each slab-class contains various numbers of slab-pages, which in turn contain a set of fixed-sized chunks. The data-objects are stored in these chunks, one item per chunk. Hence, different slab-classes need to have different chunk-sizes in their slab-pages, even though the size of the slab-pages is the same across all the slab-classes. A slab-page with a small chunk size can have many chunks, whereas the ones with the largest chunk size (same as slab-page size) can only have one chunk per slab-page. Figure-1 (source: [6]) provides a visual illustration of the same for one Kilobytes sized *slab-pages*.

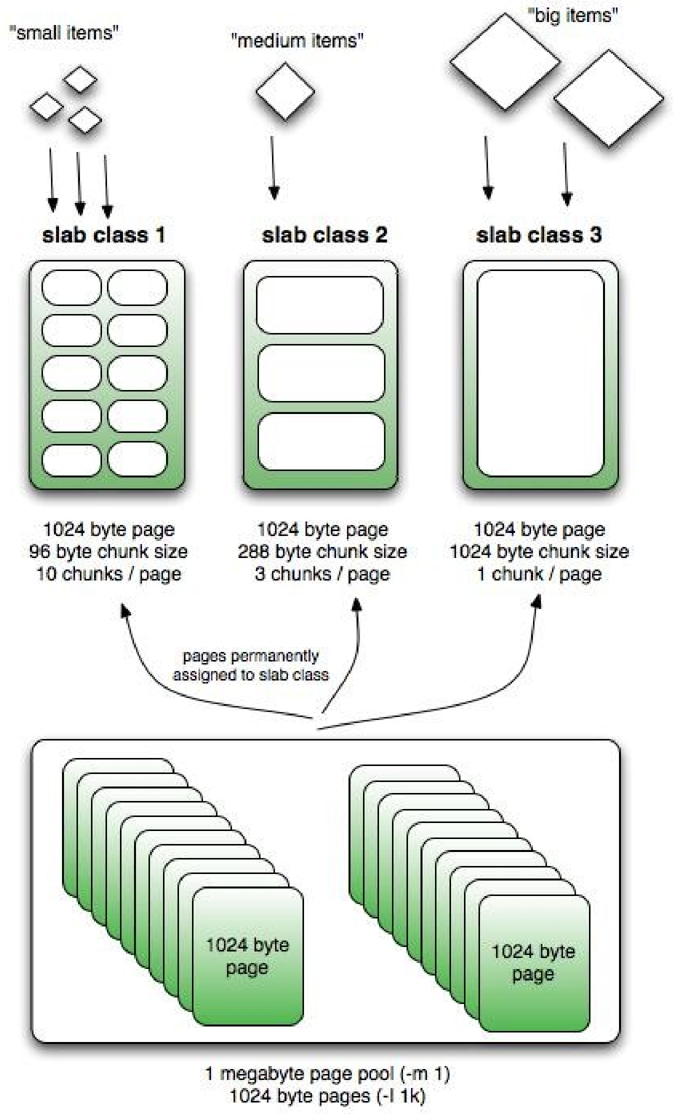

***Figure-1 (Memcached storage structure)***

## 2.2: Internals

Now that we know the basic functionality of *memcached* server (and clients), let us see how the memcached server is designed internally to fulfill those requirements. Memcached uses worker-thread based multithreading for better and faster concurrent performance. After starting, the main thread creates multiple worker threads. The number of worker-threads can be specified by a command line option (-t), and it defaults to four. In addition to that, it also creates two maintenance-threads. One of those is used for managing the size of an internal hash table, while the other is used for rebalancing the slab-memory allocation. Memcached uses libevent[31] library to handle multiple network connections (TCP and/or UDP). Libevent is an asynchronous event notification library, which provides a set of APIs to execute a callback function when a specific event takes place.

Since it is beyond the scope of our project to explain the internals of memcached server in a great detail, the flow of logic from the source code is explained here in a very abstract manner. Like any C-program, the main function of memcached.c file marks the beginning. It is followed by reading a plethora of command-line arguments given by the user/administrator for customization. The arguments are stored in related variables, or some appropriate actions are taken. There are also various custom header files and other c-program files which are used in addition to some library header files. After that, a libevent instance in the main-thread is created. Here it is used to handle a large number of concurrent network connections. Now it creates the main worker threads based on the number of threads specified by the command line parameter "-t" or (four by default). It is followed by creating a maintenance thread for maintaining an internal hash table. A slab-maintenance thread is also created for slab-class related management. Either UNIX socket can be used, or TCP and UDP both listening sockets are created together (default port number is 11211). Then the program waits in an infinite event-loop waiting for serving the client-requests. In the event of an interrupt signal (for stopping the server), it calls some cleanup functions before exiting.

**2.3: Synchronization**

The synchronization between various threads for guarding the simultaneous access to critical section of memcached server is achieved by using POSIX mutex locks (Pthread). It is the POSIX standards for threads. *"The standard, POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995), defines an API for creating and manipulating threads."* [7] In addition to protecting the critical sections, mutex variables are also used to synchronize condition-variables. Condition-variables are explained later in this section.

A lock is *"a synchronization mechanism for enforcing limits on access to a resource in an environment where there are many threads of execution"* [23]. Or, if we consider a quote:

> *"A number of mainly independent sequential-cyclic processes with restricted means of communication with each other can be made in such a way that at any moment one and only one of them is engaged in the 'critical section' of its cycle."*[24],

then a mutex lock might signify the restricted means of communication. The POSIX library provides two functions to acquire a mutex-lock, viz. pthread_mutex_lock and pthread_mutex_trylock. The former one is a blocking function, whereas the later is a non-blocking one. By blocking it means that when a thread tries to acquire a mutex-lock by (calling the former function) which is currently held by another thread, the calling thread will block until the mutex-lock is available to it. The calling (blocked) thread will actually yield its share of CPU resources to another thread if needed, thereby saving valuable resources from being wasted in just waiting for a lock to be released. The blocked thread wakes up only when it has actually acquired the mutex-lock for which it was waiting. Thus using the first version of the locking functions (pthread_mutex_lock) saves CPU resources from being underutilized, by yielding to other threads. The second function (pthread_mutex_trylock) on the other hand is a non-blocking function, which means that it immediately returns a value based on the status of the mutex-lock. As the name suggests it tries to acquire a lock and if it fails to do that, it will return an appropriate error. If it is successful in acquiring the lock, it will return 0 and the calling thread would be the owner of the lock. It is almost always the case that a thread cannot continue its operation until it has acquired the mutex-lock which it is trying to own. Hence, when using the trylock version, the thread has to continuously keep trying to acquire the lock in a loop. Such an implementation is called spin-lock or spinning. During spinning, the thread uses CPU cycles to execute the trylock function, and does not yield the CPU-resources to other threads. Although it seems wastage of CPU resources, using spin-locks has its advantages as well.

In case of yielding, the calling thread has to be put to a blocked state by the underlying operating system. This management of thread-state in itself causes some delay, which is usually much smaller than the time for which the tread blocks. However, in case where the critical-section being accessed is very small, there is a greater chance of having the block-time comparable to the time it takes to manage the states of the thread. In other words, by the time the calling thread is put to blocked-state after doing the internal processing for pthread_mutex_lock function, another thread holding the lock would have already released the lock. This extra overhead can be disadvantageous. In such a case, spinning proves to be a better option. Spinning is like a polling mechanism, where the thread does not have to change its state from running-state to any other state (unless preempted by the scheduler). If at first, the thread calling pthread_mutex_trylock finds out that it could not acquire the lock; during a few next iterations it is likely to grab the mutex-lock. Thus, depending upon the program implementation at hand, both yielding and spinning can have their pros and cons.

The programmers of memcached anticipated many situations where try-lock (spinning) is more effective. They created an inline wrapper function called mutex_*lock* (defined in a header file memcached.h). This function internally uses spinning using a pthread_mutex_trylock function in a while loop, on the mutex-lock parameter passed as a pointer. This wrapper function is used mostly with the fine-grained cache_lock mutex-lock, and is also used with some other locks guarding small critical-sections. At the heart of both spinning and yielding Pthread-lock functions, lies a hardware level atomic instructions like TestAndSet or CompareAndSwap.

### 2.3.1: Lock types

The mutex-locks for synchronization also falls under two categories, viz. fine-grained and coarse-grained locks. Fine-grained locks are the ones which are used to guard small critical-sections, usually at multiple places in the program. These provide better concurrency, but need more efforts from programmers in terms of programming complexity. Coarse-grained locks on the other hand are used to guard relatively bigger sized critical sections. These provide less concurrency, but need fewer efforts from programmers in terms of programming complexity. Sometimes, the boundary between a coarse-grained and a fine-grained lock is a blurred one.

### 2.3.2: Conditional synchronization

There are situations in a multithreaded program when a given thread cannot proceed with its execution until certain condition has been satisfied. Take for instance the case of a program in which, a master thread is waiting for some kind of (sub) results generated by many slave/worker threads. The master-thread is supposed to display the addition of all the sub-results as the final output. Unless, all the worker-threads have generated their results, the master-thread cannot continue to the final step without producing invalid results. The master-thread could be designed to spin continuously on a volatile variable indicating the number of worker-threads finishing their job. Or, it can wait for the worker-threads to signal the (sleeping or blocked) master-thread when the last-worker thread finishes its job. The later choice is more efficient in terms of CPU cycle utilization, and forms the basis of condition-variables.

A condition-variable is *"a data object that allows a thread to suspend execution until a certain event or condition occurs."[*9, Pg. 179] Each condition variable is used in conjunction with a corresponding mutex variable. There are a few condition-variables used in memcached server. The main ones are maintenence_cond, and init_cond. The first one is used to wait on certain conditions in the two maintenance threads (one for the hash-table management and the other for slab-page maintenance). The mutex variable used with it is cache_lock. The second one is used only during the initialization phase with init_lock as the associated mutex variable.

### 2.3.3: How is it used in memcached?

There are about twelve mutex variables used by the memcached server. The most prominent ones are stats_lock, cache_lock, conn_lock, slabs_lock, and init_lock. Following is a brief description of each:

**stats_lock**: it guards the critical sections containing the variables which store the status of the server.

**cache_lock**: it guards the cache operations like slab memory and hash table access.

**conn_lock**: it guards the connection list during concurrent access.

**slabs_lock**: it guards the slab-memory allocator.

**init_lock**: it is used only during the initialization of threads, for helping condition-variable named init_cond. The condition ensures that all the threads are set-up properly during the initialization state.

# Chapter 3: RESEARCH WORK DETAILS

After having introduced the software that is in center of our research work, it is time to get into other details about the research work.

## 3.1: Available options

In concurrent programming, sharing some data among various threads is the biggest issue. As we have seen in the introduction chapter, there are different ways in which thread synchronization can be handled. In the case of locking techniques, the code needs to be effectively serialized in order to process such critical sections. There are some other methods such as STM (and Oyama-Fusion [10]) execute critical sections effectively in parallel, especially in cases where the critical sections are coarse grained. For some of the options open-source libraries or mature implementations are available, while the others are more at academic research level.

Due to various constraints, we had decided to choose Test&TestAndSet (TAS), Oyama-locks (OYAMA), and Software-Transactional-Memory (STM) as our final candidates for implementation. TAS is very similar to the default Pthread mutex locks (PTHREAD) in terms of principle of operation, and could be implemented without much programming efforts. OYAMA locking on the other hand is much more different than PTHREAD locks, and required a custom implementation based on their proposed algorithm. We couldn't find any readily available open source libraries for OYAMA, especially one which was readable and capable of being used in our code. So, we decided to implement our own library functions for OYAMA. Implementing library functions is not the only change required for OYAMA. Every critical section was needed to be modified in terms of a function-call for being passed to OYAMA library functions. So implementing OYAMA took a significant amount of our time and efforts. According to the authors of OYAMA: *"under their scheme parallel programs with potential synchronization bottlenecks run efficiently"*.[11] Since the fundamental principle of operation in case of STM is much different than that of other lock-based methods, we decided to create separate branches in our source repository for implementing STM. Hence, our main trunk contained source code with options for all the three Non-STM methods viz. PTHREAD, TAS, and OYAMA. Our aim was to be able to achieve a fine-grained control over all the existing PTHREAD controlled critical sections, thereby enabling each one to be able to be controlled independently by one of PTHREAD, TAS, and OYAMA. Of course, two critical sections protected by the same mutex-variable cannot have two different methods for achieving proper synchronization. This was taken care of, by separately using a control file lehigh_config.h.

## 3.2: Software tools and platform

The accuracy of the results of any software experiment always depends upon the underlying tools and hardware used. It is the responsibility of the researchers to choose the most optimal environment for benchmarking. In this section we provide a summary of the software and hardware technologies that we used while maintaining our budget restrictions.

### 3.2.1: Hardware platform used

We used only one server for benchmarking against PTHREAD method. So memcached server as well as memslap client, both were run on the same machine. The main reason for that is, we wanted to minimize network related delays. In production environments using dedicated machines for memcached server is the norm, so multiprogramming and preemption are not real issues. We believe that the workload generated by the client software (memslap) is a real/complex workload consisting of nested locking, clever functions, conditional synchronization, and different sizes of critical sections. If an infinite sequence of requests is received by memcached (which means that we can use nuanced mechanisms), we still will have progress. Also, since it had two processors, we made use of core-affinity to attach the server to one of the CPUs and the client to the other. We believe that using in this manner justifies our benchmarks being affected only by various synchronization methods. Again, since memcached is a look-aside system, we could tolerate some progress relaxations. In our benchmarks, we did not try to further optimize the program. For example, by careful existing partitioning of some operations per thread (like statistics gathering) and then combining the statistics when required, the original program could have been optimized. Following table gives relevant details about our test server specification.

| ***HEADING/PARAMETER*** | ***CORRESPONDING VALUE*** |
|---|---|
| CPU/Processor | Intel Xeon processors running at 2.66 GHz (Intel(R) Xeon(R) CPU X5650 @ 2.67GHz) |
| No. of cores per CPU | 6 |
| HyperThreading | Yes (2 threads per core) |
| No. of threads/CPU | 12 |
| No. of CPUs | 2 |
| Total threads | 2 x 12 = 24 |
| Memory (RAM) | 12287216 Kilobytes (12 Gigabytes) |

***Table-1 (Benchmark server hardware specification)***

### 3.2.2: Software used

In this section, we provide a list of software that were used in our research project. Also, we provide a brief introduction of each one where required.

***Operating System:*** We used 64-bit Linux, Ubuntu 12.04.1 LTS, kernel: 3.2.0-29-generic SMP (x86_64 GNU/Linux) on the server.

***Subversion:*** Subversion version: 1.6.17 was used for revision control.

***Memcached:*** We used version: 1.4.13 of memcached, and installed it from a compressed tar source file. We added this code to our software repository, which we later modified to implement other techniques. The original code was able to be compiled to a 64-bit executable file.

***Memslap:*** Version: 1.0 was used as the main benchmarking software.

***Mutrace:*** We used version: 0.2 of mutace, which is a mutex profiler. This software was used to detect the statistics about Pthread mutex-locks used in the source code of memcached server.

***GCC:*** GNU Compiler Collection version: 4.7.1 was our choice of compiler system for compiling even our STM code, although some other implementations of STM are available. The choice was made in order to achieve simplicity in coding STM branches. We are aware of the fact that STM is an experimental feature and might not give the optimal performance. We had to use gcc version >= 4.7 for compiling with STM support using fgnu-tm option.

***GDB:*** The GNU debugger version 7.4-2012.04 was used for debugging purpose.

***Perl/Bash Scripts:*** Some Perl (5.14.2 version) and Bash (version 4.2.24) scripts were used to run the benchmarks.

## 3.3 Chosen synchronization options

Let us go into some more details of the three synchronization options that we have chosen for implementation.

### 3.3.1: TestAndSet

TestAndSet lock is the first of our three choices.

#### 3.3.1.1: Overview of TestAndSet

TestAndSet lock is the one that is very similar to the default Pthread mutex locks. However it does not require a use of mutex lock. It requires a hardware that supports an atomic test_and_set instruction on a shared (integer) variable at the assembly level. Even Pthread mutex operations depend upon an underlying hardware support for atomic operation, but they operate somewhat like wrapper functions around the low level atomic operations. The operations of TestAndSet locks can be explained as follows. A test_and_set instruction takes the lock (integer) variable as its parameter which usually has 0 or 1 value. It writes 1 to the lock variable and returns whatever was the previous value of the variable, in an atomic manner. So, a lock can be implemented using test_and_set by spinning continuously on the lock variable until it returns its previous value as 0. This can happen only when the lock variable was initialized first time to 0, or when some other thread which held the lock wants to relinquish it by writing 0 to the lock variable. When another thread writes a 0 to the lock variable, the spinning thread is guaranteed to see the 0 value before overwriting it to 1. After that, the spinning loop is broken and the thread becomes the owner of the lock, as long as it doesn't set the lock variable back to 0.

Although TestAndSet is simple in implementation, continuous use of test_and_set instruction in a loop can be expensive, since *"it can lead to resource contention in busy lock (caused by bus locking and cache invalidation when test-and-set operation needs to access memory atomically)"*.[13] In order to avoid this scenario, a similar technique called Test&TestAndSet is used. The concept behind this is to avoid spinning on the lock variable using test_and_set instruction, but using a simple comparison operator. When the value of the lock variable is found to be 0, then there is a much higher probability of acquiring the lock using test_and_set instruction. This avoids the expensive test_and_set instruction from being run continuously while spinning.

#### 3.3.1.2: TestAndSet implementation

We have implemented TestAndSet/Test&TestAndSet locks using built-in gcc function __sync_lock_test_and_set. Its prototype is as follows:

```
type __sync_lock_test_and_set (type *ptr, type value, ...);
```

*"It writes value into *ptr, and returns the previous contents of *ptr"*.[14] In this function we pass the value 1 as the second parameter, unlike the traditional test_and_set function in which we assumed that it will overwrite the value in the lock variable to 1 during each call. A sample from the program code for acquiring a TestAndSet lock looks like:

```
while(__sync_lock_test_and_set(<address of a lock variable> ,1)) { }
```

A code sample for acquiring a *Test&TestAndSet* looks like:

```
while(__sync_lock_test_and_set(<address of a lock variable> ,1))
{
        while( <same lock variable's value> );
}
```

A code sample for releasing a lock looks like:

```
__sync_lock_release( <address of the lock variable> );
```

The above function simply sets the value of the lock variable to 0. This is an optional function, and could be replaced by a simple assignment statement which sets the value of lock variable to 0. We have used Test&TestAndSet lock everywhere, except while replacing Pthread mutex locks which used spinning instead of yielding. The manner in which various synchronization methods are invoked is as follows. Each occurrence of the original PTHREAD mutex critical section was embedded under the preprocessor directives #ifdef or #if defined. For example, a critical section protected by stats_lock used "#if defined(CRITSEC_048_PTHREAD)". If CRITSEC_048_PTHREAD was defined in the global header file, then PTHREAD method would be selected. On the other hand, if CRITSEC_048_TAS was defined then TAS method would be executed due to the next #elif defined(CRITSEC_048_TAS) directive and so on. It is made sure that only one of the methods (PTHREAD, TAS, and OYAMA) is defined for each critical section. It should be clear by now that, this method of controlling the synchronization options holds true for TAS and OYAMA as well. For STM, we decided to create separate branches due to the complexity of the modifications involved.

### 3.3.2: Oyama locks

Oyama-lock was the method of synchronization for our next choice.

#### 3.3.2.1: Overview of Oyama

Oyama-lock is based on the paper by Oyama et al.[11] Before going into the details of Oyama-locks, it is better to first understand flat-combining logic. According to the authors, flat-combining is *"a new synchronization paradigm based on coarse locking"*.[12] In simple terms, *flat-combining* is a technique in which concurrent contending threads co-operate with each other in terms of executing their critical sections. A single thread which grabs a global-lock first gains a special status of being a combiner. All other contending threads publish their work in a linked list known as publication-list. The combiner thread executes on behalf of every other thread the functions that were requested by the waiting threads. The waiting threads wait on their respective publication entries, till they get the results produced from their respective functions run by combiner thread. Thus instead of wasting time in acquiring and releasing a lock by all the contending thread, it saves this time by co-operation between the thread holding a lock and the requesting threads.

Oyama is very similar to flat-combining, except a few differences. The central idea behind the implementation remains the same. The thread which grabs a lock first among contending threads, becomes an owner thread. The lock here doesn't mean a mutex-lock variable; instead it can be a shared integer. Oyama requires a low level hardware support for two atomic instructions (or their equivalents), viz. compare_and_swap and swap. These low level instructions are similar to the one in TestAndSet lock. The other requesting threads send pointes to their respective critical-section functions to the owner thread for executing on their behalf. The owner thread performs the execution of all the requested functions, including its own function, and remains as an owner thread until all the requests are not satisfied.

### 3.3.2.2: Oyama implementation

In order to implement Oyama-locks, we had to write our own library routines. The original algorithm is given in their research paper by Oyama et al.[11] The main routine that interfaces with the source code is oym_get_lock. Following is its prototype:

```
void oym_get_lock(volatile oym_mutex_t *mutex,void(*func)(void*), volatile void *args);
```

This function takes 3 parameters, viz. Oyama-lock variable, a pointer to functions, and a pointer to arguments of the function. Oyama-lock is nothing but a 64 bit unsigned integer which can hold an 8 byte pointer value (since we used a 64 bit platform). The argument pointer args is by default a void pointer (void*), which is usually type-casted to an array of multiple values. Implementing Oyama-locks not only required creating library functions, but also making changes in the source code. We had to convert every critical section originally protected by PTHREAD lock into subroutines/functions. In other words, we created a wrapper function for every critical section in order to pass it as a parameter to oym_get_lock function. Of course, corresponding arguments were also identified and passed in the form of an array accordingly along with the function pointers.

### 3.3.2.3: Other thoughts

By looking at Oyama-locking method, we can think of Oyama Progress Property as the one in which, the system is guaranteed to make progress as long as the lock holder in not swapped out indefinitely. One issue of a serious concern while using Oyama is that, the starvation caused by the lock-owner to other requesting threads in case of nested calls. There are two type of situations which can cause starvation (or similar situation).

In one case, a thread (say T1) grabs the lock. While working on its requested task too much work comes in, and so the thread never gets a chance to release its lock. This case is not clearly a case of starvation, since thread-T1 finishes with its critical section but cannot move forward with further execution as long as all the pending requests are not finished. Since memcached service is look-aside by nature, this situation should not affect much in our case.

In another case, a thread (say T1) grabs the lock A, and then the same thread grabs another lock B and gets stuck in processing requests for lock B as in the previous case. Since T1 is stuck in processing work for lock B, those thread waiting for work done on lock A cannot make progress, and they starve. In this case, the Oyama Progress Property is definitely lost. The problem arises because of composition or nesting of Oyama-locks. It is worth noticing that the same problem does not exist in Flat-Combining.

**3.3.3: STM**

Our third and final choice to test synchronization mechanisms was Software-Transactional-Memory. Unlike the other two lock based methods; this one is based on the concept of atomic transactions from the database systems.

**3.3.3.1: STM overview**

Software-Transactional-Memory is an alternative to lock-based synchronization mechanisms. STM uses optimistic concurrency control. Optimistic Concurrency Control methods are:*"'optimistic' in the sense that they rely mainly on transaction backup as a control mechanism, 'hoping' that conflicts between transactions will not occur".*[15] In case of such systems, before committing each transaction verifies the data which it had accessed with their original values at the start of the transaction. If any datum is modified, the transaction has to rollback. This way of operation makes a huge impact on the manner in which programs are written for using STM. It simplifies programming, since the programmers have to think in terms of atomic blocks instead of spending time in working with shared data management like identifying critical-sections, minimizing them, avoiding deadlock, and so on.

STM is *"a shared object which behaves like a memory that supports multiple changes to its addresses by means of transactions".*[16] STM in brief can be understood as follows. The root of STM lies in atomic transactions. A programmer encloses a piece of code in the form of a compound statement (similar to the critical sections in a lock-based method), and annotates that block as an atomic block. So, any operation that take place within that block of code can either fully completed (committed) or failed/aborted (rolled-back). Thus this atomic statement behaves like a database transaction. Any changes made to variables by this block can be visible if and only if it commits. If many threads start executing the same atomic block simultaneously, they all execute the same code and possibly modify the same variables. Before committing a given thread, the STM logic verifies whether any other thread changed any of the variables that the current thread modified. If there is no modification of variables by other threads, then and then only the current thread will commit and make permanent changes to the variables. On the other hand, if any variable was modified by some other thread, then the transaction has to abort/rollback and tries again later from the beginning. In case of a contention, the progress might be slow since the contending threads have to effectively serialize. A major problem in using STM is use of non-reversible operations (like systemcalls, I/O etc.) inside the atomic transactions. Since, these operations cannot be undone; the transactions cannot rollback/abort. So, all STM implementations need to handle such non-reversible operations in some way or the other.

#### 3.3.3.2: STM implementation

As mentioned before, we used gcc compiler to utilize its new STM support. Initially, we wanted to convert all the critical sections of the original code to STM. However, due to various constraints and challenges paused by STM, we decided to create ten separate branches in our repository using STM. Following table lists all the ten branches along with the progress we made in each one.

| ***BRANCH*** | ***MODIFICATIONS MADE*** |
|---|---|
| 001_memcached-1.4.13_no_cachelock_condvar | Replaces only the conditional variables which are guarded by cache_lock |
| 002_memcached-1.4.13_stm_no_cachelock | 001 + Replaces cache_lock mutex with atomic transactions whenever possible |
| 002_memcached-1.4.13_stm_no_slabslock | 001 + Replaces slabs_lock mutex with transactions |
| 002_memcached-1.4.13_stm_no_statslock | 001 + Replaces stats_lock mutex with transactions |
| 003_memcached-1.4.13_callable_everywhere | 002_no_cachelock + Makes all function declarations in header files as *transaction_callable* |
| 004_memcached-1.4.13_no_asserts | 003 + Removes/comments assert statements |
| 005_memcached-1.4.13_safe_refcount | 004 + Makes reference counts transactional |
| 006_memcached-1.4.13_slabs_lock | 005 + Swaps lock orders in transactions that acquire slabs_lock |
| 007_memcached-1.4.13_nocl_nostatslock | 006 + Replaces all stats_lock with transactions |
| 008_memcached-1.4.13_nocl_nosl_noslabslock | 007 + Replaces all slabs_lock with transactions |

***Table-2 (STM branches in subversion repository)***

Before going further with STM implementation details, it is advisable to understand how STM is provided by gcc. The atomic block explained before is implemented by __transaction_atomic keyword. When it is placed before a compound statement, the statement becomes an atomic transaction. Similarly, to take care of irreversible operations inside a transaction, there is a keyword called __transaction_relaxed. Unlike atomic transactions, *"the relaxed transactions may contain unsafe statements. Relaxed Transactions that execute unsafe statements may appear to interleave with non-transactional operations from other threads"*.[17] Also, as per the official documentation, *"relaxed transactions cannot be cancelled. Irrevocable actions may limit the concurrency in an implementation; for example, they may cause the* *implementation to not execute relaxed transactions concurrently with other transactions"*.[17] It should be clear that, in an ideal scenario all the atomic blocks would be __transaction_atomic blocks. We tried to convert the targeted PTHREAD locks to atomic-transactions first. However, when the block of code contained irreversible actions (such as print statements, assert statements etc.), we had to convert those to relaxed-transactions. In addition to that, whenever a transaction safe function is used inside an atomic-transaction block, it should be declared as such with transaction_safe attribute. The exact syntax is:

```
__attribute__((transaction_safe)) <function declaration/definition>;
```

STM using GCC 4.7 allows using different algorithms for the actual STM internal logic implementation. This option can be specified by setting ITM_DEFAULT_METHOD, and ITM_METHODS environment variables to suitable values. For example, we used two algorithms Orec-Eager/Orec-WT (value=ml_wt) and Serial-Irrevocable (value= serialirr). The gcc implementation is explained in greater details on their official documentation page.[17]

### 3.3.3.3: Other thoughts

STM implementation was one of the most challenging of all the three methods. The process of converting functions to transaction_safe and converting critical sections to atomic-transactions was a subtle one. There are other challenging issues that we encountered as well, like converting reference counts to atomic transactions, making sure that the code is going to work with a non-Privatization safe algorithm like Orec-Eager that uses Orecs[18,19], and so on. The privatization problem can be informally explained as: *"an action taken by a transaction that modifies program state in such a way that some previously shared data structure will henceforth be accessed by only one thread"*.[20] The main reason for these challenges was the fact that, we modified the existing code to make it work with STM. The code was not originally designed with STM in mind. However, we think that without using STM in our project, it would have been of not much interest.

# Chapter 4: PERFORMANCE EVALUATION

Now that we have mentioned the details about our research work setup, we move on to the next step which consists of the observation of the result.

## 4.1: Setup environment and options

As described in the previous chapter, for benchmarking against the default synchronization method (PTHREAD) we have chosen three options viz. *TestAndSet* (TAS), Oyama-locks (OYAMA), and Software-Transactional-Memory (STM). To use Non-STM methods (TAS, and OYAMA), we used our main trunk without creating separate branches in our subversion repository. Since we used the preprocessor directives *(*#ifdef or #if defined etc.) to select a particular option for each critical section, we decided to create a control file named lehigh_config.h having the corresponding #define directives. The main purpose of this file is to aggregate all the instances of a particular lock variable under a single controlling #define directive. Otherwise, it would have been very difficult and inefficient to manage the synchronization methods for each critical section separately. For instance, if in one of the critical sections protected by stats_lock, the method is PTHREAD and in another it is OYAMA, then it would be a major bug which might even go undetected after producing wrong results. Using the control file, we had restricted the individual critical sections from being accessed directly. There is a single line which needs to be modified for each of the lock variables used. As an example, defining PTHREAD_INIT_LOCK_CONTROL, TAS_INIT_LOCK_CONTROL, and OYAMA_INIT_LOCK_CONTROL (using #define directive) will set the critical sections for init_lock to Pthread, TestAndSet, and Oyama-lock respectively. Only one of the 3 options is allowed to be chosen for each such mutex variable. There are 12 such control lines for a total of 12 lock variables, one per each.

The use of 12 independent lines (one per lock variable) each with 3 options (PTHREAD, TAS or OYAMA) can lead to 3^12 or 531441 combinations. We call it a fine-grained control, as opposed to coarse-grained control in which a global define directive is used to override these 12 independent lines. If a directive COARSE_GRAINED_PTHREAD is defined (using #define), then all the critical sections follow Pthread locking mechanism. Use of COARSE_GRAINED_TAS and COARSE_GRAINED_OYAMA changes it to TestAndSet and Oyama respectively. So using a coarse-grained control only 3 combinations can be achieved viz. all Pthread, all TestAndSet, and all Oyama. Of course, it was impossible for us to blindly generate benchmarks against the default Pthread method for 531440 combinations. Therefore, we took some other factors into consideration for choosing a few combinations.

## 4.2: Various results

In this section we provide the results obtained by running our modified code(s) using the benchmarking software memslap. In each of the tests, we used our memcached server to run with variable number of worker threads using -t command line option. In all the graphs that we plotted, the X-axis consists of the number of server-threads. Y-axis consists of average time (sum of 10 results divided by 10) in seconds it took to enter a total of 5,000,000 entries of random key/value pairs by memslap software during each of 10 different times. Since our test machine had two processors, each with 6 cores and 12 threads (Intel HyperThreading), it was capable of running 24 threads in parallel. We tried to bind the server threads to lower numbered CPU cores/threads, and memslap threads to higher order CPU cores/threads using taskset command. Our purpose was to produce results that are as much close to the real world scenario as possible. The server was allocated 6 Gigabyte of ram for data caching.

### 4.2.1: All non-STM (4, 8, 12 client threads)

The first benchmark that we ran against the default (all PTHREAD) configuration involved all-TAS and all-OYAMA configurations. Since we almost always used server threads varying from 1 to 12 during each run, we decided to vary the number of client threads from 4, to 8, and then to 12. Thus for each configuration of client-threads we generated 3 plot, one for each of the non-STM methods. The result generated a total of 3x3 = 9 line plots on the plotted graph for this benchmark. The following table (table-3) lists more details about our selection.

| ***HEADING/PARAMETER*** | ***CORRESPONDING VALUE*** |
|---|---|
| No. of client threads | 4, 8, and 12 |
| No. of server threads | 1 to 12 |
| server CPU affinity | 1 to 12 (0x00000FFF) |
| client CPU affinity | 13 to 24 (0x00FFF000) |
| Branches | All-PTHREAD, All-TAS, All-OYAMA |
| Total line-plots | 3 (total client configurations) x 3 (no. of branches) = 9 |

***Table-3 (Configuration for all non-STM 4, 8 & 12 client threads)***

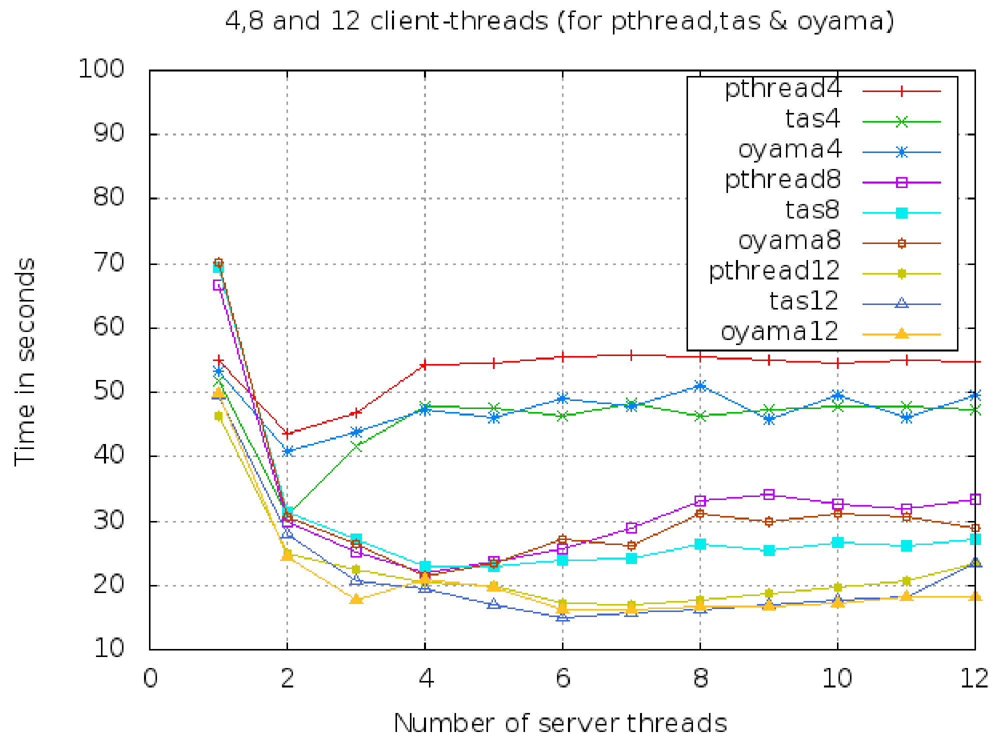

***Figure-2 (Graph plot for all non-STM 4, 8 & 12 client threads)***

As found from the graph of figure-2, for higher number of client threads (8 and 12) the time taken to enter 5,000,000 records decreases with the increase in the number of server threads. Also, for a given number of server-threads greater than 3, the time taken decreases with increase in the number of client threads. After reaching to 12 client-threads, the performance difference between PTHREAD, TAS and OYAMA diminishes. However, it is clearly visible in case of 4 and 8 client threads, that TAS and OYMA perform almost equally better than PTHREAD. Also, it is worth noticing that for a small number of client threads (here 4), the performance is optimal when server threads are set to 2.

### 4.2.2: Combinations – 1 & 2 (4 client threads)

In this benchmark, we used two combinations against their respective main methods. Our 1st combination consisted of setting stats_lock, slabs_lock, and cache_lock to TAS method while keeping others to PTHREAD. We compared this plot against ALL-TAS mode. In our 2nd combination, we set the same three locks to OYAMA while keeping others to PTHREAD. We compare this against ALL-OYAMA. Of course, the ALL-PTHRED line plot is provided for comparing all four line plots against the default method. So in all, we present 5 line-plots here. The reason for this comparison is the result from running mutrace on memcached in ALL-PTHREAD mode. Following is a collection of various parts from the output of mutrace program showing the information about top 5 contending mutexes:

```
mutrace: Showing statistics for process memcached (pid 16084).
mutrace: 8247 mutexes used.

Mutex #1726 (0x0x626f60) first referenced by:
   /home/trv211/myroot/usr/lib/mutrace/libmutrace.so(pthread_mutex_init+0xf2) [0x7f4bdafe84b2]
……………..
Mutex #8192 (0x0x61ef20) first referenced by:
   /home/trv211/myroot/usr/lib/mutrace/libmutrace.so(pthread_mutex_lock+0x49)
[0x7f4bdafe86b9]
……………
Mutex #2469 (0x0x627900) first referenced by:
   /home/trv211/myroot/usr/lib/mutrace/libmutrace.so(pthread_mutex_init+0xf2) [0x7f4bdafe84b2]
…………....
Mutex #1675 (0x0x626ec0) first referenced by:
   /home/trv211/myroot/usr/lib/mutrace/libmutrace.so(pthread_mutex_init+0xf2) [0x7f4bdafe84b2]
……………...
Mutex #3930 (0x0x6222a0) first referenced by:
   /home/trv211/myroot/usr/lib/mutrace/libmutrace.so(pthread_mutex_lock+0x49)
[0x7f4bdafe86b9]
……………..
mutrace: Showing 10 most contended mutexes:
 Mutex #   Locked  Changed    Cont. tot.Time[ms] avg.Time[ms] max.Time[ms]  Flags
   1726       19       18       13       0.029        0.002        0.005 M-.--.
   8192       41       31        2       0.009        0.000        0.001 M-.--.
   2469 17031622 15432123        0    28077.633        0.002        3.992 M-.--.
   1675 14633284  8098540        0     2794.728        0.000        3.985 M-.--.
   3930  8211072  4996818        0     2502.469        0.000        0.812 M-.--.
```

This output was converted into readable variable names using gdb as follows:

```
(gdb) info symbol 0x626f60 : init_lock in section .bss

(gdb) info symbol 0x61ef20: conn_lock in section .bss

(gdb) info symbol 0x627900: cache_lock in section .bss

(gdb) info symbol 0x626ec0: stats_lock in section .bss

(gdb) info symbol 0x6222a0: slabs_lock in section .bss
```

What we found is, the three locks mentioned here take the maximum time in their critical-sections and also fall under top 5 contending locks. Therefore, we wanted to make sure that these three locks are mainly responsible for the delay in all-PTHREAD method when using 4 client threads. By replacing these three culprits by TAS (combination-1), and OYAMA (combination-2), we expected that the performance should match to those of all-TAS and all-OYAMA respectively. Table-4 gives the details of the configuration for this benchmark.

| ***HEADING/PARAMETER*** | ***CORRESPONDING VALUE*** |
|---|---|
| No. of client threads | 4 |
| No. of server threads | 1 to 12 |
| server CPU affinity | 1 to 12 (0x00000FFF) |
| client CPU affinity | 13 to 24 (0x00FFF000) |
| Branches | All-PTHRED, All-TAS, TAS + PTHREAD, All-OYAMA, OYAMA + PTHREAD |
| Total line-plots | 1(total client configurations) x 5 (no. of branches) = 5 |

***Table-4 (Configurations for combinations 1 & 2)***

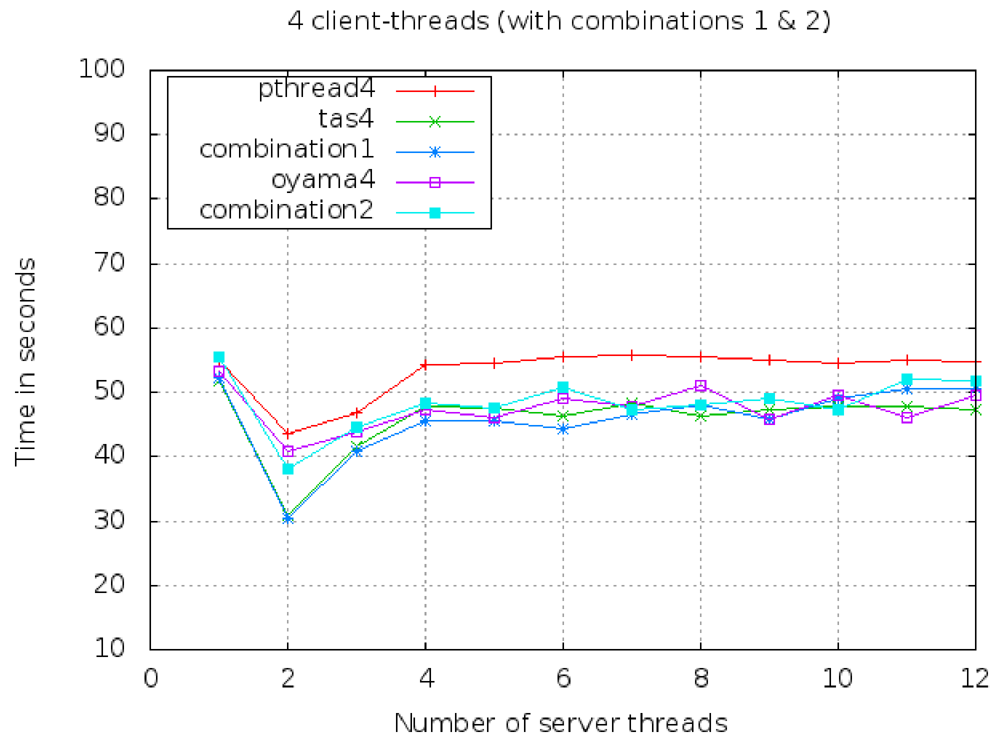

***Figure-3 (Graph plot for combinations 1 & 2)***

As seen from the graph of figure-3, the three locks mentioned here are indeed mainly responsible for the delay in all-PTHREAD mode.

### 4.2.3: STM Orec-wt (Orec-Eager) algorithm (12 client threads)

The next benchmark that we ran was using 10 different branches of STM code that we had created. Here also we vary the number of server threads from 1 to 12, but keep the number of client threads constant to 12. The STM algorithm that we used for this benchmark was Orec-Eager or Orec-WT. In order to use this algorithm, we had to set ml_wt into the environment variables ITM_DEFAULT_METHOD, and ITM_METHODS. The following table (table-5) gives the configuration details about this benchmark.

| ***HEADING/PARAMETER*** | ***CORRESPONDING VALUE*** |
|---|---|
| No. of client threads | 12 |
| No. of server threads | 1 to 12 |
| server CPU affinity | 1 to 12 (0x00000FFF) |
| client CPU affinity | 13 to 24 (0x00FFF000) |
| Branches | All-10 STM branches (Orec-WT/Orec-Eager algorithm) |
| Total line-plots | 1(total client configurations) x 10(no. of branches) = 10 |

***Table-5 (Configurations for STM OrecWT algorithm)***

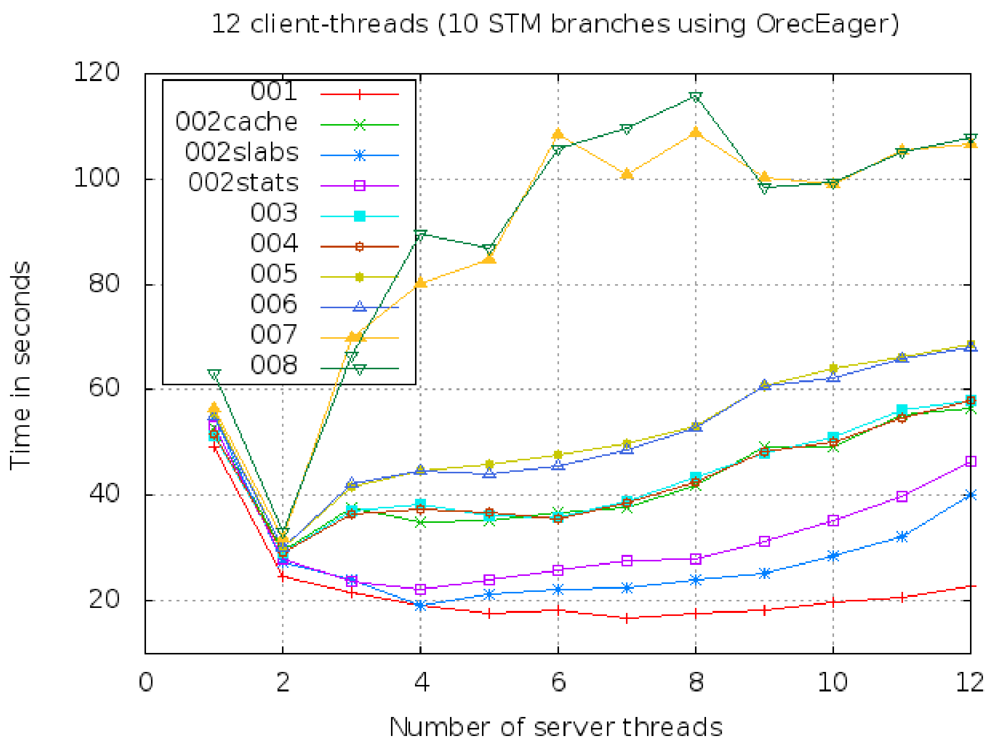

***Figure-4 (Graph plot for STM OrecWT algorithm)***

Since GCC support for STM is "an experimental feature, with several parts being not optimized"[21], its performance was not comparable to that of TAS, PTHREAD or OYAMA. Anyway, *"The extent to which STM systems can be fast enough for use in practice remains a contentious research question in itself"*.[22] So, instead of comparing with non-STM methods, we analyze the performance difference among the ten STM based branches that we had created in our subversion repository. As described in table-2, we started from replacing the condition-variables guarded by cache_lock under the branch with its name starting with 001. We continued to make changes in the original code in order to convert more locks into transactions. As seen from the graph of figure-4, there is a significant difference in performance between line plots for branches starting with 007/008, and the rest of the branches. At first we suspected that the percentage of relaxed transactions might have increased in the two branches. However, we realized that the percentage of relaxed transactions was lower in the two branches than that in branch beginning with 003. It was found that the penalty we paid in terms of time was because of the conversion of stats_lock into transactions.

### 4.2.4: All STM default/serialirr algorithms (12 client threads)

This is very similar benchmark to the previous one, except that here we used SerialIrrevocable algorithm in STM by setting serialirr in the environment variables ITM_DEFAULT_METHOD, and ITM_METHODS. The configuration details are mentioned in the following table (table-6).

| ***HEADING/PARAMETER*** | ***CORRESPONDING VALUE*** |
|---|---|
| No. of client threads | 12 |
| No. of server threads | 1 to 12 |
| server CPU affinity | 1 to 12 (0x00000FFF) |
| client CPU affinity | 13 to 24 (0x00FFF000) |
| Branches | All-10 STM branches (serialirr algorithm) |
| Total line-plots | 1(total client configurations) x 10(no. of branches) = 10 |

***Table-6 (Configuration for STM SerialIrrevocable algorithm)***

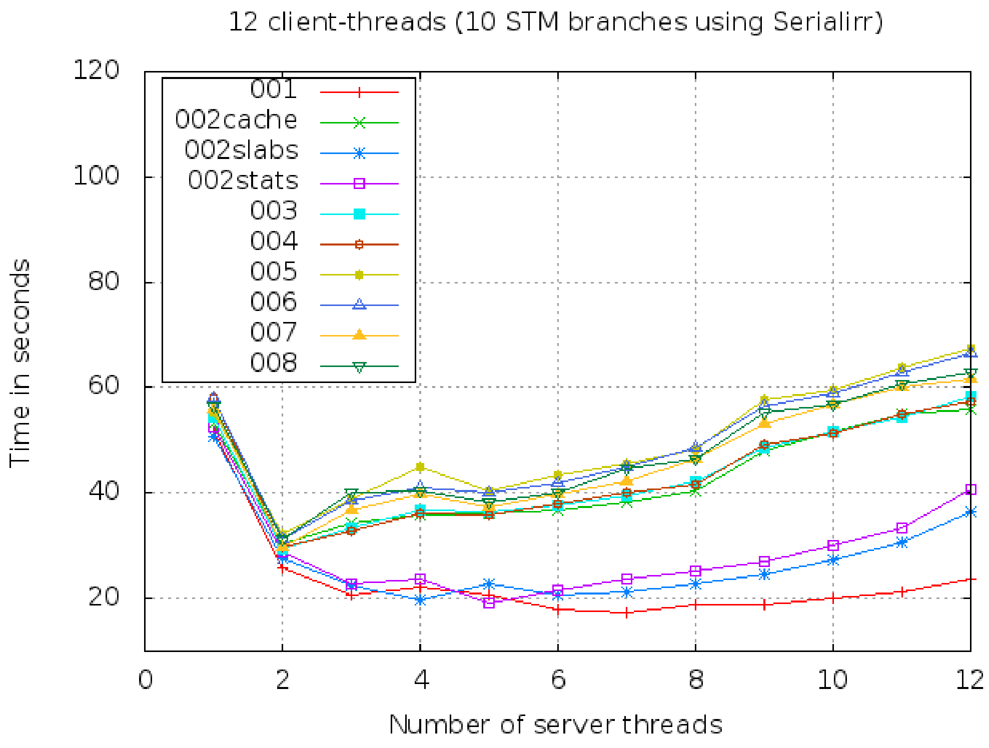

***Figure-5 (Graph plot for STM SerialIrrevocable algorithm)***

As seen from the graph of figure-5, it should be clear that SerialIrrevocable algorithm performs better than Orec-Eager, especially in the case of stats_lock related transactions which are introduced in the branches ending with 007 and 008. This difference is clearly visible when compared with figure-4, where the time taken reaches to around 120 seconds for 5 server threads in case of those two branches. In case of other branches, there is not much performance difference between SerialIrrevocable and Orec-Eager algorithms.

# Chapter 5: CONCLUSION AND FUTURE WORK

In the previous chapter, we analyzed the results of various benchmarks that we had conducted as a part of our project. Of course, the benchmarks that we ran using different configurations were just a small subset of possibly limitless combinations. However, we think that these runs were sufficient in helping us to arrive to some basic conclusions about various synchronization mechanisms.

One interesting observation in case of non-STM methods was that, as the number of clients threads increases the performance of PTHREAD improves to be equal to that of TAS and OYAMA. Also, for higher number of client threads (here greater than 7), the performance reaches to optimal value as soon as the number of server threads increase to 3 or 4. It implies that the software is efficient in terms of its design. Also, we found that mutrace is a very effective tool in finding out the locks which are most contending, or the most time consuming. Yet another interesting observation was in case of STM methods, where SerialIrrevocable was found to perform better than Orec-Eager, at least for the two branches that had stats_lock converted into transactions. Except for those two branches, the performances of both the algorithms were almost identical. So we conclude that stats_lock played a bigger role in decreasing the performance of memcached server using Orec-Eager algorithm under STM method. That mutex lock is a fine grained mutex lock in the original source code. However, it is used so frequently that it was among top five most contending locks as well in top three most time consuming mutexes, when we dynamically traced the original software using mutrace.

Finally, we conclude that there is a lot of room for enhancements, which we are planning to incorporate into our future work. First thing that we want to do in the future is to add some more lock based mechanisms like MCS locks, Fair locks, Ticket locks, and some others. We also want to add back off feature in Test&TestAndSet locks, which can be tuned to find an optimal performance. We are planning to test out the use of normalizing for having function calls. The current implementation of Oyama-locks library is not optimized. It uses malloc function which is time consuming. We want to use thread local variables, and eliminate the use of malloc in the future implementation of Oyama library functions. There is an equally great amount of work that is needed to be done on STM branches as a part of the future work. In the future, we wish to use other more mature STM implementations, such as RSTM [29] or TinySTM [30]. In the present research work, we did not make much use of many command line options available in memcached server as well as in memslap client. As far as possible, we continued to use the default options in both the cases. We would like to apply more tuning to these software, in order to verify for any performance difference with a greater accuracy.

# REFERENCES

1] http://en.wikipedia.org/wiki/Moore's_law
2] http://en.wikipedia.org/wiki/Thread_(computer_science)#Multithreading
3] http://en.wikipedia.org/wiki/Thread_(computing)
4] http://en.wikipedia.org/wiki/Critical_section
5] http://memcached.org/
6] http://nosql.mypopescu.com/post/13506116892/memcached-internals-memory-allocation-eviction
7] http://en.wikipedia.org/wiki/POSIX_Threads
8] http://docs.libmemcached.org/libmemcached.html#description
9] An Introduction to Parallel Programming (Book) By Peter Pacheco *(ISBN: 978012374260590000)*
10] Fusion of Concurrent Invocations of Exclusive Methods, by: Oyama et al. (PACT 01)
11] Executing parallel programs with synchronization bottlenecks efficiently, by: Oyama et al.
12] Flat Combining and the Synchronization-Parallelism Tradeoff by: Hendler et al.
13] http://en.wikipedia.org/wiki/Test_and_Test-and-set
14] http://gcc.gnu.org/onlinedocs/gcc-4.1.1/gcc/Atomic-Builtins.html
15] On Optimistic Methods for Concurrency Control, By Kung et al. *(ACM Transactions on Database Systems, 1981)*
16] Software Transactional Memory, By Shavit et al. *(Proceedings of the 14th ACM Symposium on Principles of Distributed Computing)*
17] Draft Specification of Transactional Language Constructs for {C}++, by: Tabatabai et al. *(2009)*
18] A Transactional Memory with Automatic Performance Tuning *(Proceedings of the 7th International Conference on High-Performance and Embedded Architectures and Compilers, 2012)*
19] Lightweight, Robust Adaptivity for Software Transactional Memory, by: Michael Spear (Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures, 2010)
20] Scalable Techniques for Transparent Privatization in Software Transactional Memory, by: Marathe et al.
21] http://gcc.gnu.org/wiki/TransactionalMemory
22] Transactional Memory, 2nd edition (Book), by: Harris et al.
23] http://en.wikipedia.org/wiki/Lock_(computer_science)
24] Solution of a Problem in Concurrent Programming Control, By Edsger Dijkstra *(Communications of the ACM, 1965, page 569)*
25] From Lightweight Hardware Transactional Memory to Lightweight Lock Elision, by Pohlack et al. *(Proceedings of the 6th ACM SIGPLAN Workshop on Transactional Computing, 2011)*
26] Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors, by: Mellor et al. *(ACM Transactions on Computer Systems, 1991)*
27] The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors, by: Anderson, T. E (IEEE Trans. Parallel Distrib. Syst.)
28] OpenMP Application Programming Interface, Version 3.0 *(OpenMP Architecture Review Board, 2008)*
29] http://www.cs.rochester.edu/research/synchronization/rstm/
30] http://tmware.org/tinystm
31] http://libevent.org

# VITA

Mr. Trilok J. Vyas, son of Mr. Jayaprakash Vyas and Mrs. Rupa Vyas was born in India in year 1979. He has earned a Bachelor's of engineering degree with Computer Engineering major in year 2001 from the prestigious Mumbai University, India. He joined Investment Research & Information Services Limited (India) for the position of a PHP-Developer and resigned as a Technical Leader. He then joined Zycus Infotech Private Limited (India) as a Senior Software Engineer. He came to the USA in year 2004, and started working as a web-application developer at PRG Systems Inc (New Jersey). After working there for about 5 years, he moved to Cooperative Communications Incorporated (New Jersey) as a VoIP Analyst/Web-Application Developer. During his tenure at the telecommunication company, he finished two graduate level courses in Electrical and Computer Engineering from Villanova University with flying colors. He was enrolled at Lehigh University as a full time graduate student in August, 2011. During his tenure at various organizations as a software-professional he acquired a vast amount of skills, especially in the web-development, and open-source software areas. His areas of interest are: Parallel Programming related technologies, Operating Systems, Digital System Design, Database Systems, Open-source software, Web-Development technologies, and many others.