[Come up with title]

Abstract

In a world where data and information are being processed and transferred at an ever-increasing rate, the need for multiple processors has become popular in recent years. With multi-processor machines, computing power is significantly improved, but the problem of sharing memory between cores, or memory coherence, arises. A recently proposed solution that is both simpler and more scalable than the widely used directory coherence, TARDIS, uses timestamp counters to logically order memory operations to maintain sequential consistency, as opposed to using physical time. The vanilla TARDIS protocol, however, uses long hardware counters for timestamp storage. Additionally, for some applications, TARDIS generates a large number of renew requests which consumes the precious network bandwidth. Thus, we propose several optimizations: a timestamp compression scheme to reduce the memory cost of storing timestamps, and several lease predictor protocols to increase efficiency by minimizing the number of renewal requests due to cacheline expiration. [briefly mention results]

1 Introduction

In the modern day, there is a continuously increasing demand for faster computer processors. Since one of the most effective methods to address this is to add multiple cores to a CPU and allowing tasks to execute in parallel, the number of cores is increasing exponentially (a certain IBM chip contains 4,000 processor cores [cite?]). Multiple cores allow for tasks

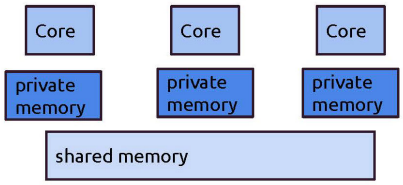


Figure 1: Simple diagram of a shared memory system.

to be split up and handled simultaneously, but accurate and efficient performance requires parallelism and exploitation of shared memory. The essence of maintaining the correctness of data between cachelines is to prevent the use of stale data, which is old information that is being used when it has already been altered by another core. This problem is known as memory coherence, and a coherence protocol is necessary to facilitate performance and scalability of the system.

Memory coherence is most commonly and traditionally addressed by directory based coherence protocols [cite something], which is relatively intuitive. In such a system a central directory tracks the coherence between the caches, or how they are being owned or shared between cores. Thus, if a core requests private ownership of a cacheline, it must go through the directory so that all other lines can be invalidated and updated accordingly. Since the directory must maintain the sharer information for every cacheline, the system requires O(N) storage per cacheline if there are N cores. As N increases, the directory based protocol does not scale well.

This problem is addressed by TARDIS, a new coherence protocol that is currently in development and has been mathematically proven to be sequentially correct [citation?]. TARDIS utilizes logical timestamps (to be explained in more detail in Background) to enforce the global memory order. The protocol only requires O(log N) storage per cacheline and is therefore significantly more scalable as well as being simpler. However, the major disadvantages of the vanilla TARDIS protocol is managing very large timestamps and dealing with potentially many renewal requests, most of which are unnecessary.

The goal of this paper is to explore the expansive design space of TARDIS and optimize the protocol through timestamp compression and renewal minimization. We implemented a clever base timestamp + delta scheme to reduce the storage overhead of timestamps, implemented and analyzed several configurations of lease protocols to reduce the number of renewals, and designed a livelock detection algorithm to prevent several worst-case-scenarios of spinning variables

2 Background

TARDIS uses timestamps to logically organize shared memory and ensure coherence. It excels at being simple and very scalable. By operating with logical time instead of physical time, TARDIS is able to have actions that seemingly travel in time. (Henceforth, time will refer to logical time.)

To give an analogy, think of a book in the library that two people want to read and then a third wants edit. Intuitively, it would seem that the two readers must first finish reading the book before the third person can edit, but TARDIS allows all three people to perform their tasks at the same time (Need to verify this). Using arbitrary but realistic numbers, the first person may read the book from logical time 0 to 10, the second person reads from 0 to 20, and the editor travels in time and starts editing the book at time 21.

This phenomenon is allowed to happen because of the use of timestamps to indicate logical time. Each cacheline, a basic unit of data storage, has two timestamps: Write Timestamp (WTS) and Read Timestamp (RTS). There is also a timestamp in each core called the Program Timestamp (PTS), which is only incremented when accessing data and should not be confused with the processor clock. PTS marks the cores position in time and can vary between cores. RTS is always greater or equal to WTS, and the span of time between them dictates when the data is valid. This period is known as the lease because the core is basically reserving the information for a duration of time so that it will remain valid

by specifying any change to that piece of data to be timestamped after the lease period. When a cores PTS is in that range, the core may use that data, and when PTS is not, then the core must send a request to shared memory, known as the LLC, to request the newest version of the data, known as a renew request. If the WTS in the LLC is still the same, then the data has not been changed and the core can request another lease to keep using said data. If the data has been changed, then the renew request fails and the LLC sends back the newest version of the data.

The cachelines in shared memory can be in two states, shared and exclusive. A cachenline being in the shared state means that multiple cores can read from it. When a core wants to modify it, the cacheline is set to the exclusive state, signaling that it is exclusively owned by said core. When another core needs to read that cacheline, a writeback request is sent to the owner core, which then writes the updated data back to the LLCwith updated timestamps.

Take the following example:

[Picture of example here] http://people.csail.mit.edu/devadas/pubs/tardis.pdf

0. The initial state of the two cores and two cachelines in the LLC

1. Cacheline A is moved into core 0 and the LLC registers that it is exclusively owned by core 0 since its value is being edited. PTS in core 0 is increased to 1 because an action, in this case changing its value to 1, is performed and thus it is advancing in logical time.

2. pts of core 0 remains 1 because although cacheline B was loaded, no action was performed on it. B is only being read, so its rts increases to pts+10=11, where 10 is an arbitrarily picked lease value, to indicate a period of validity that lasts until pts exceeds 11. Methods of choosing lease will be discussed in the optimizations section. In the LLC, Bs timestamps and changed to reflect that it is being leased.

3. Core 1 needs to edit B, but that cacheline is currently being read by core 0, so when it puts B in the exclusive state, PTS, RTS, and WTS are all set to 11+1=12 to depict

that core 1 is logically ahead of core 0. Bs timestamps show that the Bs in each core are existing at different places in logical time which means it having value of 0 in core 0 and a value of 1 in core 1 is perfectly valid. This is the phenomenon of traveling in time that is unique to TARDIS because cacheline B is existing in different logical times at the same instant in physical time.

4. Core 1 now needs to read A, so it leases the cacheline for 10 timestamps, which makes its RTS = 12+10 = 22 where 12 is the PTS of core 1 and 10 is the arbitrarily chose lease. Since it is now being read and not edited, its exclusive ownership by core 0 changes to shared ownership, as reflected in the timestamps in the LLC.

3 Optimizations

In this section, we will detail the three optimizations we have implemented to improve the efficiency and storage overhead of TARDIS.

3.1 Timestamp Compression

One of the major structural changes implemented in TARDIS is the inclusion of two timestamps in each cacheline. This translates to a large additional requirement for memory, so the optimization of timestamp compression was researched to reduce storage costs. We exploit the fact that the two timestamps of each cacheline are usually fairly close, so we implement a base timestamp (bts) and a delta = rts-wts, the difference between the timestamps. The timestamps were originally 64 bits each, so two timestamps per 512 bits of data is a 25% storage overhead, and not efficient. With the bts and delta, we were able to dramatically reduce storage costs, but we encountered the problem with timestamp rollover, where a timestamp would numerically increase beyond what the bts + delta scheme could support.

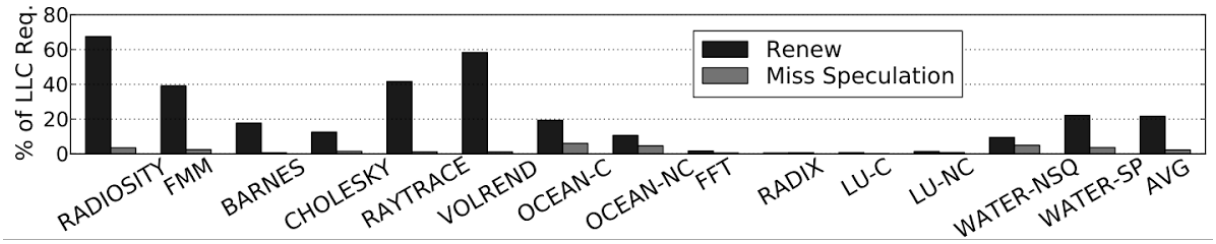


Figure 2: Graph illustrating the percentage of renew requests and miss speculations (requests that actually necessary) for various benchmarks.

3.2 Lease Prediction

Core 0

while *true* **do**

    read A;

    B++;

end

Core 1

while *true* **do**

    read A;

    B++;

end

One of the major issues with TARDIS is the fact that because of the concept of a lease, timestamps, specifically the PTS, will increase rapidly if a cacheline is write-intensive. Consider the example above. Assume the lease is arbitrarily chosen as 10. Since B is write-intensive, many renew requests for exclusive ownership will be made to maintain coherence between the two cores, thus causing the PTS to increment quickly in steps of 10. Therefore, although A is read-only, the cacheline needs to be repetitively renewed unnecessarily, as its lease constantly expires. Since renew requests incur extra latency and network traffic, this primitive static lease protocol is undesirable. Further motivating the need for a better lease protocol can be seen in 2. For certain benchmarks, about 60% of requests are renews, while a significantly small percentage of renewal requests are miss speculations, which are necessary renews.

With these observations in mind, we notice that write-intensive lines should have very small lease sizes so that when renew requests are sent, we prevent prolific increase in timestamps. On the other hand, since renewing is unnecessary in read-only lines, there is a

potential for much larger leases. The basic implementation idea is that if data is renewed, then we progressively give the cacheline a smaller lease whereas data that are suspected to be read-only are incrementally given longer leases, for later renewals. In order to maintain these lease states, we add an n-bit counter to the cacheline that will keep track of a particular lines state. Thus, if n is 2, for example, four states exist, namely 00, 01, 10, and 11.

Since this lease protocol is a very broad idea, we developed implementations that require several parameters, or different variations. One parameter is the value of n. If n is larger, this could allow for a more calibrated lease predictor, as the number of states increases exponentially, but also creates much more storage overhead (extra n bits per cacheline). Another choice is either decrementing the state progressively when a line requests exclusive ownership (10 01) or clearing the state counter immediately (10 00). In either case though, if a lease expires, we increment the state from 00 to 01, for example. The third parameter to be examined is the mapping from state to the actual lease. We have two basic algorithms, one is an exponentially growing lease and the other is a linearly growing one. Both are similar in nature, and revolve around the same ideas previously mentioned, but increase differently, as evident from the names. We also investigate the effect of using different start points for the lease (the lease assigned to 00), and the increase factor (how much a previous lease is added, or multiplied by). The sweep studies for these parameters can be found in Evaluations.

4 Livelock Detection

Core 0	**Core 1**
while *!done* **do**	
| (nothing)	done = false → true
end	

The problem with livelock occurs when a core needs to wait for one or more cachelines

to satisfy certain parameters before continuing forward. However, those cachelines never change because PTS does not increase since the cores processes are not going forward in the first place. Take the example above. Core 0 is waiting for done to be true before continuing via an empty while loop. Core 0 will never realize that done becomes true because an empty loop will not increase the cores PTS. We developed three solutions to this.

1. Increase the PTS after a constant number of cycles.

   This simple solution definitely worked because a constantly increasing PTS meant that every cacheline would expire eventually and be updated from the L2 cache with data that allow the core to move out of livelock. However, it didnt seem like the most effective solution because there could still be a relatively long time between the cachelines causing the livelock to be changed in the L2 cache and when the core realizes the change in the L1 cache. This scheme was used as the baseline for comparing other livelock detection techniques.

2. Implement a livelock bit in the core, a livelock counter in the core, and an access counter in each cacheline. This is a two-step process that first checks whether there is a livelock, and then renews the cachelines that seem to be causing the livelock. This scheme is invoked every memory access. By using variable last_cts, the core is able to determine whether the PTS has increased since the last memory access. If it did increase, then that means the core is progressing that there is no livelock. If the PTS did not change, then the livelock counter is incremented by 1. When it reaches a certain livelock threshold value, the livelock bit, initially false, is set to true to signal that there is a livelock going on. Henceforth, whenever a cacheline is accessed, its individual access counter is incremented by one until it reaches an access threshold, in which case that specific cacheline is renewed. At any time during this process, if the PTS changes when compared to last_cts, then all counters are reset to 0 and the livelock bit is set to false.

Table 1: Default Configuration of Tardis.

Timestamp Compression	
Delta timestamp size	20 bits
L1 Rebase Overhead	128 ns
L2 Rebase Overhead	1024 ns
Livelock Detection	
Livelock detection	turned on
Self increment period	1000
Livelock Detection	
Lease prediction	turned on
Start lease	8
Increase factor	2

This method aims to target specific cachelines that could be causing the livelock, thus reducing the network traffic overhead when compared to the first scheme. The removal of self increment also means that cachelines that are not related to the livelock will not prematurely expire.

3. Use a cacheline buffer in addition to self increment. By keeping a buffer of the four most recently accessed cachelines, the core can guess the cachelines causing the livelock since they will be the most often checked. In the unlikely case that there are more than four cachelines simultaneously causing the livelock, self increment is in place to catch those outliers. [Need test results]

5 Methodology

We used the Graphite simulator [2] to model our multicore architecture. Graphite was developed at MIT and is able to simulate systems with up to 1000 cores. In this report, we only simulate a 64-core system, which matches the latest Intel multicore processor [1]. Each core has a 32 kB private cache and all cores share a 8 MB shared cache. All the cores and caches are connected using an on-chip network with MESH topology.

Table 1 shows the default configuration of Tardis. All the three optimizations introduced

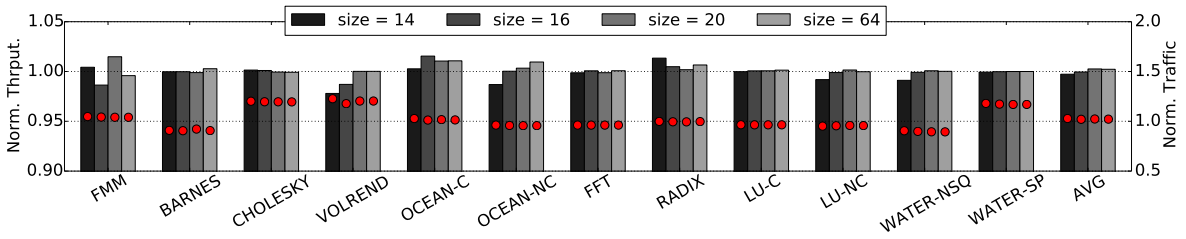


Figure 3: Performance of Tardis sweeping the delta timestamp size

in Section 3 are turned on by default. The default delta timestamp size is 20 bits. The rebased overhead for L1 and L2 cache is 128 ns and 1024 ns respectively. During this rebase time, the cache is not able to serve other requests. The default self increment period is chosen to be 1000. The start lease is 8 for each cacheline and the increase factor is 2. Unless otherwise stated, all the experiments use these parameters by default.

We use a subset of Splash-2 [3] benchmarks to evaluate our optimization techniques. For each experiment, we show the speedup (in bars) and the amount of network traffic (in red dots) for each benchmark. For some Tardis experiments, we also show the rate of renewals, i.e., the precentage of renew requests out of all L2 cache requests.

6 Evaluations

In this section, we evaluate the performance of the optimization techniques introduced in Section 3.

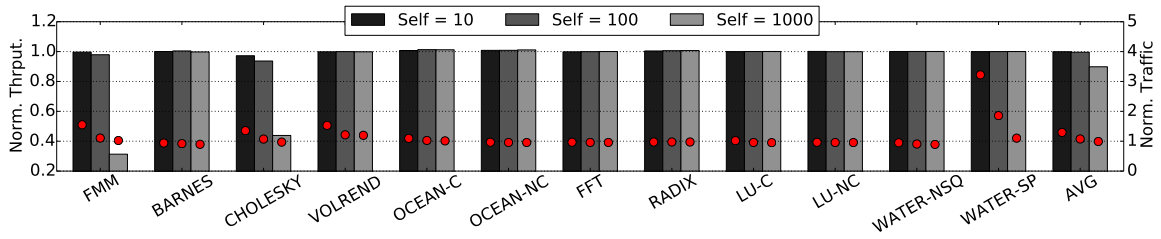
6.1 Timestamp Compression

Please cover at least the following points in Fig. 3:

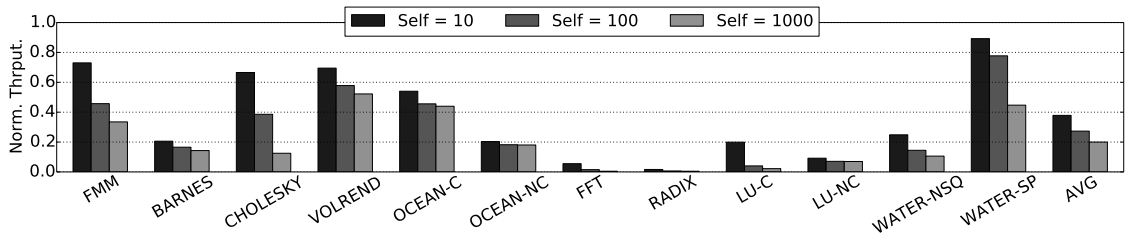
- size = 14 has bad performance because of frequent rebase.

- As logn as delta timestamp size is large enough, performance is not sensitive to the timestamp size.

- We chose size = 20 as the default value since it provides reasonable performance over all benchmarks.



(a) Thruoghput and Network Traffic



(b) Renew Rate

Figure 4: Baseline Tardis without livelock detection.

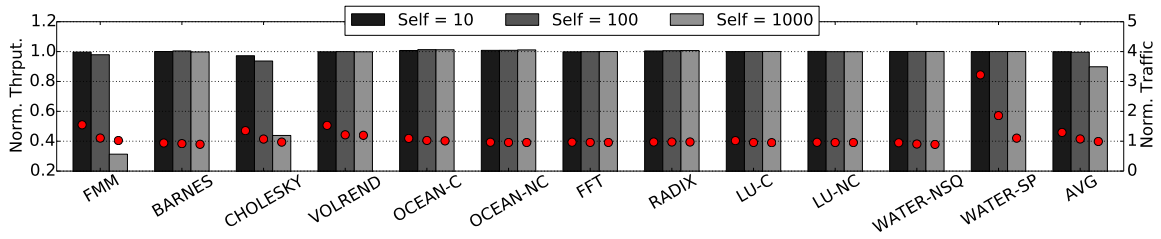
6.2 Livelock Detection

For Fig. 5, at least discuss the following points:

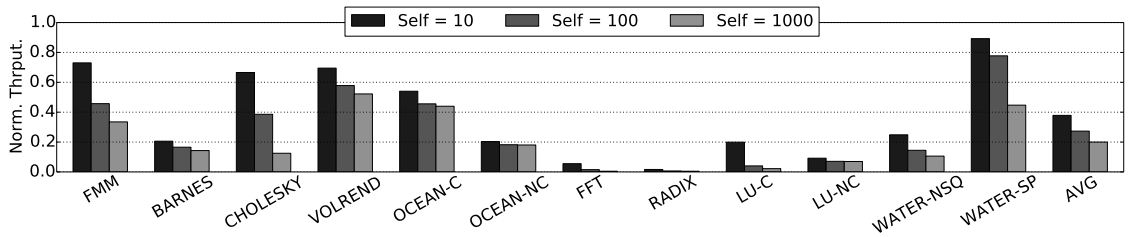
- For baseline Tardis, both performance and renew rate go down as self increment period increases. Performance goes down for some benchmarks because of spinning.

7 Conclusion

TARDIS is a new scheme to ensure memory coherence. Its main advantages are its scalability, simplicity, and to perform operations that travel in time. It uses logical time by incorporating timestamps in cachelines and cores. We tested multiple optimizations involving timestamp compression, lease prediction, and livelock detection, and were successful in increasing TARDISs speed and efficiency overall (we should run a test to comepare baseline TARDIS with all of our optimizations). We tested a variety of benchmarks on 64 core simulated machines via the Graphite simulator on MIT CSAILs Cagnode servers.



(a) Thruoghput and Network Traffic



(b) Renew Rate

Figure 5: Baseline Tardis without livelock detection.

References

[1] Intel. Intel Xeon Phi Coprocessor System Software Developers Guide, 2014.

[2] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A Distributed Parallel Simulator for Multicores. In *International Symposium on High-Performance Computer Architecture*, 2010.

[3] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *International Symposium on Computer Architecture*, 1995.