[Come up with title]

Ethan Zou and Henry Liu

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 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.

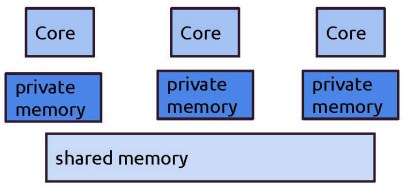

Figure 1: Simple diagram of a shared memory system.

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 by Xiangyao Yu and Professor Srini Devadas 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 unless otherwise specified)

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.

Still need to touch up on this paragraph: 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 is known as the lease. When PTS is in that range, inclusive, 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 DRAM 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 DRAM sends back the newest version of the data (?).

The cachelines in shared memory can be in two states, shared and exclusive. A cachenline is in the shared state such that multiple cores can read from it. When a core wants to modify it, the cacheline is set to the exclusive state, meaning 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 DRAM with updated timestamps.

Take the following example: [Example program from original paper here]

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.

3.2 Lease Prediction

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.

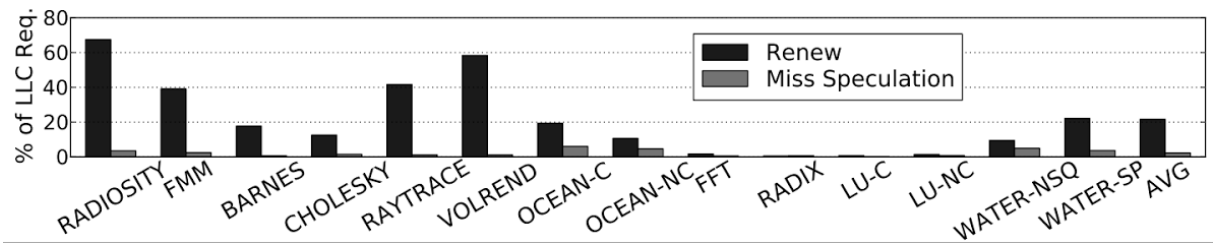

Figure 2: Graph illustrating the percentage of renew requests and miss speculations (requests that actually necessary) for various benchmarks.

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

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.

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

[Xiangyao to write about this]

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

6 Evaluations

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

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.