15-418 Final Project - Cachesim

Sam Flattery (sflatter) Brian Wei (bwei1)

December 13, 2020

Project Website: https://samflattery.github.io/cachesim

1 Summary

We implemented a cache simulator for a NUMA multiprocessor machine using directory based cache coherence and used the simulator to analyze the cache behavior of various synchronization lock implementations. We used Intel's `pin` program and a custom written pintool to gather memory access traces of different parallel programs written using these locks and compared their performance under the MSI, MESI and MOESI cache coherence protocols. We created plots illustrating the differing cache performance of the locks and the differences between the protocols under various metrics such as cache hits/misses, main memory accesses and interconnect events.

2 Background

2.1 NUMA

NUMA (Non Uniform Memory Access) is a multiprocessing computer architecture in which memory access times depend on the proximity of the memory location to the processor requesting the access. It is often used in distributed machines, as it means that processors have faster access to a subset of localized memory while still retaining the shared memory abstraction at the expense of occasional higher latencies when accessing memory further away.

2.2 Cache Coherence and Directories

In multiprocessor machines, where each processor has its own cache(s), lines of memory become replicated in multiple places, which means that different processors can observe different values for reads to the same memory address. For example, if two processors have the same line of memory in each of their caches and one writes to it, the other will observe stale data on future reads. In order to fix this problem, modern multiprocessor machines implement cache coherence systems within which caches communicate with each other under various coherence protocols to ensure each cache is aware of the current state of the line.

There are two main ways to implement cache coherence for a write-back, write-allocate cache - snooping-based and directory-based. Snooping-based cache coherence is suited to smaller machines with relatively few processors, as it relies on the idea that coherence information is broadcasted on some system-wide interconnect to all other caches. However, this protocol doesn't scale very well to larger machines as the cost of the broadcasts scales with the number of processors. Instead, large scale machines often use a directory-based implementation of cache coherence, in which a directory is used to track the set of caches that contain a line of memory. Under this scheme, coherence messages can be sent point-to-point to just the caches that need to be updated with new coherence information, which means the protocol scales well with the number of processors. In NUMA machines, a distributed directory scheme is often used, meaning each NUMA node has its own directory which tracks the state of each line of memory in that node and these directories can communicate with each other to coordinate the system (see figure 1).

For each line in memory, the directories maintain a set of P presence bits, where P is the number of processors in the system, as well as two bits representing the state of the line. A line can be in one of three states:

1. **Uncached** (`U`) - the line is not present in any processor's cache
2. **Shared** (`S`) - the line is in one or more caches and is clean
3. **Exclusive/Modified** (`EM`) - the line is in exactly one cache who owns the data

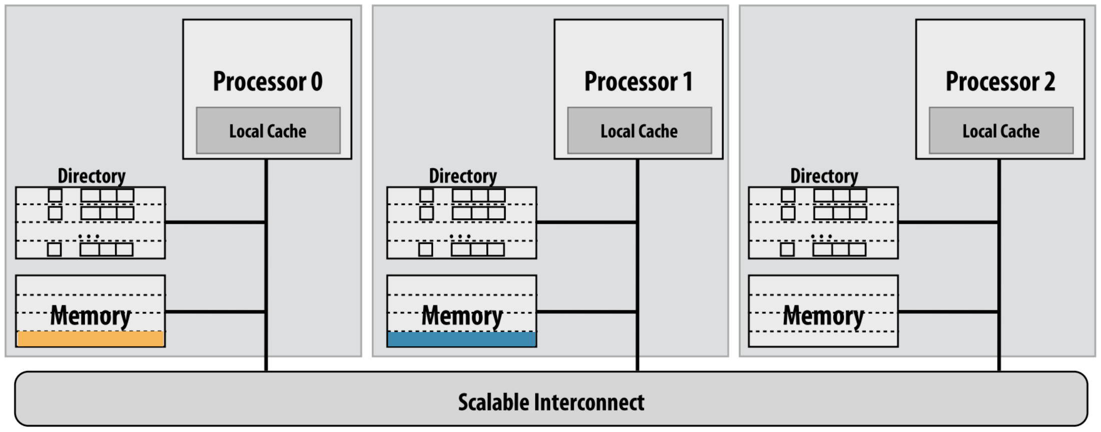


Figure 1: A distributed directory on a NUMA system (source)

In order to implement the MOESI coherence protocol (discussed in 2.3), our directory also stores a cache identifier of which cache has the line in an owned state so the directory can forward read requests from other sharing caches to that cache.

The directory receives messages on the interconnect from caches, such as a `BusRd` when a cache wants to read a line or a `BusRdX` when a cache wants to read a line with the intention of writing to it (i.e. it needs exclusive access to the line) and responds to the messages by transitioning its state and sending its own coherence messages to the relevant caches. The state transitions are illustrated in figure 2 below.

2.3 Cache Protocols

The caches also maintain a few bits of state for each line they have cached. Many different protocols are used here, but we chose to implement and analyse 3 - MSI, MESI and MOESI (each letter corresponds to a state the cache can be in).

- **M** (modified) - the cache holds a dirty modified copy of the line, no other caches have copies
- **O** (owned) - the cache holds exclusive rights to a line, other caches may read but only this cache may modify and must broadcast changes to all sharers
- **E** (exclusive) - the cache has the only copy of the line and the line is clean

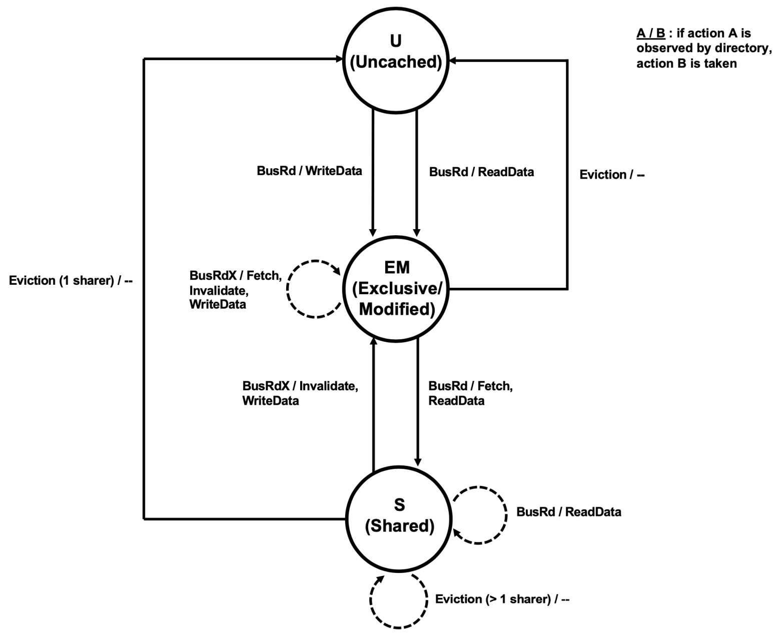

U (Uncached)
A / B : if action A is observed by directory, action B is taken
BusRd / WriteData
BusRd / ReadData
Eviction / --
Eviction (1 sharer) / --
BusRdX / Fetch, Invalidate, WriteData
EM (Exclusive/ Modified)
BusRdX / Invalidate, WriteData
BusRd / Fetch, ReadData
S (Shared)
BusRd / ReadData
Eviction (> 1 sharer) / --

Figure 2: The state transitions of a directory

- **S** (shared) - the cache is one of at least one other sharer of the line and the line is clean
- **I** (invalid) - the block is not valid in the cache and cannot be read/written

where the states in MSI are a subset of those in MESI, etc. The cache lines transition between these states in response to events from the processor, as well as events received on the interconnect from the directories. The state transition diagrams for the coherence protocols we implemented are presented below (the interconnect transactions are explained in tables 2 and 3).

MSI requires two interconnect messages for the common case of reading an address and then writing to it (e.g. incrementing a variable). MESI helps to reduce this inefficiency by adding the `E` state, which means that if a processors wants to do a read-write and no other processor is currently sharing the data, it will get an exclusive copy of the data for the read which can then be upgraded to a modified copy for free without any interconnect traffic.

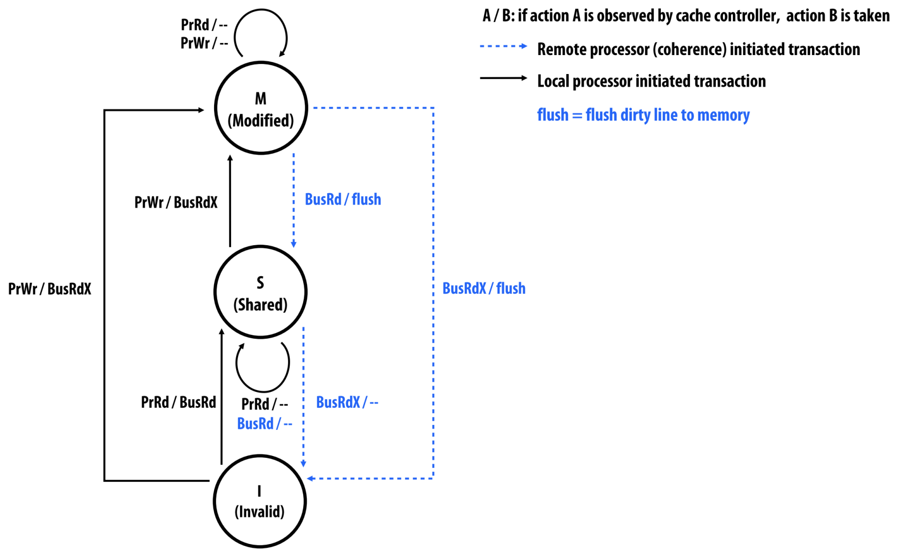

A / B: if action A is observed by cache controller, action B is taken
Remote processor (coherence) initiated transaction
Local processor initiated transaction
flush = flush dirty line to memory
PrRd / --
PrWr / --
M (Modified)
PrWr / BusRdX
BusRd / flush
S (Shared)
PrWr / BusRdX
BusRdX / flush
PrRd / BusRd
PrRd / --
BusRd / --
BusRdX / --
I (Invalid)

Figure 3: The state transitions in MSI (source)

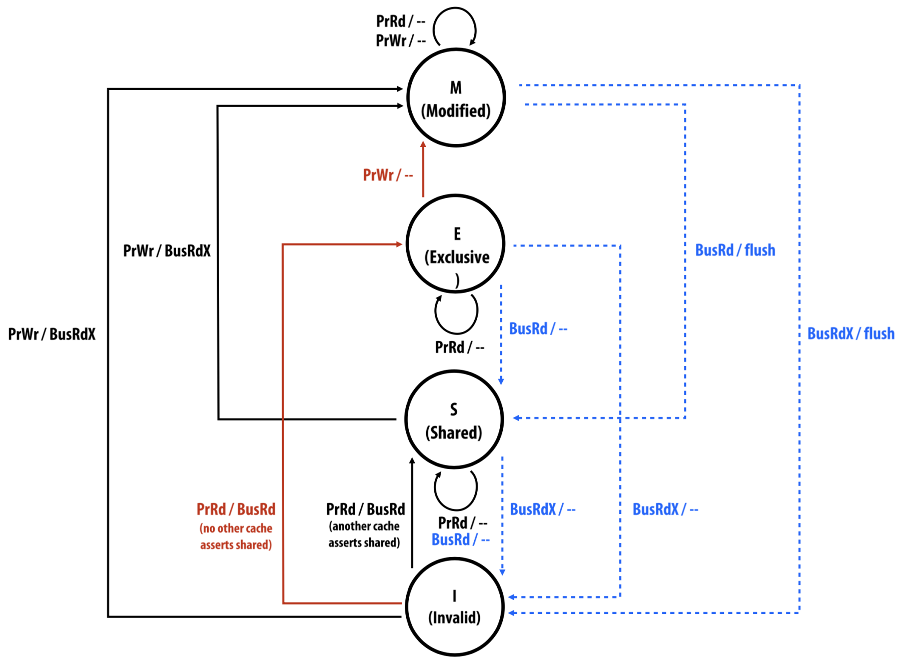


Figure 4: The state transitions in MESI (source)

The MOESI protocol operates very similarly to the MESI protocol, with the addition of the `O` state. This protocol reduces the number of memory writes necessary, as now when another CPU wants to read a dirty line, a flush is not necessary as the data is now sent over the interconnect directly from the owning cache.

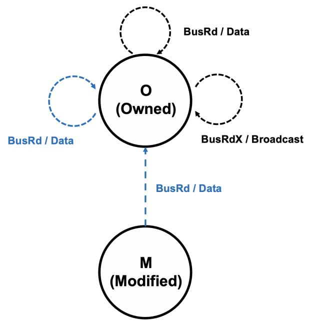


Figure 5: The new state transitions in MOESI (other transitions remain the same as MESI)

2.4 Locks

In order to evaluate the cache simulator, we run the simulator on a large number of traces generated from running parallel programs. We chose to analyse several lock types, as locks serve as a fundamental aspect of synchronizing parallel programs and they exhibit interesting cache coherence behaviour as their memory can be highly contended. We chose to implement the following types of locks to look for differences in their performance in various cache protocols:

- **test-and-set:** This lock relies on the atomic test and set instruction. The lock will spin on the test and set instruction, checking repeatedly whether a variable is in a desired *unlocked* state and trying to set it to be *locked* if so.

- **test-and-test-and-set:** This lock improves on the test-and-set lock by reducing the traffic on the interconnect by spinnning on a test while the lock is locked, then only attempting to acquire the lock with test-and-set when it knows the lock has been released.

- **ticket lock:** The ticket lock provides an improvement over the first two lock types for two reasons - fairness (by guaranteeing that locks are granted in the order they are requested) and fewer invalidations. It results in fewer invalidations since only a read is required to acquire the lock, instead of a write in the test-and-test-and-set case, which means the only invalidations come from the release of the lock. It is implemented by maintaining a ticket number that each thread increments when it acquires the lock, and a serving number which is incremented when a thread releases the lock. A thread then acquires the lock when its ticket number is equal to the serving number.

- **array lock:** The array based lock is similar to the ticket lock in that it maintains fairness, while providing a further decrease in the amount of interconnect traffic. To do so, the lock uses an array where a thread acquires the lock by waiting on an element ofthe array to be set to the unlocked state, and released the lock by setting the next element to the unlocked state. This lock can be further improved by having each array element padded to be on separate cache lines, which prevents false sharing. The drawback of this lock is the space requirement, which is linear in the number of threads.

3 Approach

3.1 Pintool

We used Intel's `pin` program to instrument the binaries of several parallel programs we had written in order to generate memory access traces of the programs. `pin` allows you to insert callbacks into a compiled binary which run whenever certain instructions are executed. In our case, we used `pin`'s built in `INS_MemoryOperandIsRead` and `INS_MemoryOperandIsWrite` functions to insert a callback after every memory read and write from the programs under analysis.

Our pintool creates a trace that looks like the following:

```
[CPU] [R/W] [Address] [NUMA Node]
1 R 0x7ffef17ecea8 0
2 W 0x7ffef17ecea0 1
3 W 0x7ffef17ece90 0
4 W 0x7ffef17ece88 0
```

We make the simplifying assumption here that each thread is running on a different physical CPU, which obviously is not always the case but it allows us to better analyse the programs we write since we get a more even distribution of work between the CPU cores. We found when generating traces that even programs with high amounts of parallelism ran on only a few physical processors.

We also used a simple probabilistic method to decide which NUMA node an address resided on. When running a program through pin, we will periodically read and parse the Linux proc filesystem's numa maps. This provides information on various regions of memory and how many numa nodes they are mapped to. This information is not precise however; it will only indicate the number of pages in the region that belong to each node (e.g. if the region has 20 pages, it may indicate there are 8 on node 0 and 12 on node 1). Whenever we encounter an address on a previously unseen page, we probabilistically assign that page to one node from the given region based on the ratios. This information is saved so that future accesses to the same page are given its numa node consistently.

3.2 Usage

Our simulator is run with the following parameters:

```
-t <tracefile>: name of memory trace to replay
-p <processors>: number of processors used to generate trace
-n <numa nodes>: number of NUMA nodes used to generate trace
-m <MSI | MESI | MOESI>: the cache protocol to use
-s <s>: number of set index bits (number of sets = 2^s)
-E <E>: associativity (number of lines per set)
-b <b>: number of block bits (number of blocks = 2^b)
-v: verbose output that displays trace info
-a: display aggregate stats
-A: display aggregate stats not including processor 0
-i: display individual stats (i.e.) per cache, per NUMA node
-h: help
```

The user specifies how many processors and NUMA nodes the trace was generated with, as well as the protocol to use and the properties of the cache (i.e.e how many blocks per set, the size of the blocks and how many sets per cache). These default to `s = 6`, `b = 6`, `E = 8`, giving an 8-way associative 32kB cache which is the configuration of an Intel L1 cache.

The user can also specify the output of the simulator using the `-Aaiv` flags. The `-v` flag gives a verbose output which prints the interconnect events that are happening on a per memory access basis. The `-a -i` flags print stats after running the program on an aggregate or per cache level. We noticed that for the smaller programs that it was feasible to generate pin traces for, thread 0 had far more activity than the other threads since it was being used to set up the program and create other threads, etc., so we included the `-A` flag to give aggregate stats without the first thread's stats being included. These flags can be used together to get the entire output of the program.

3.3 Stats and Latencies

The entire list of stats the simulator collects are shown below:

```
struct Stats {
  size_t hits_;                 // cache hits
  size_t misses_;               // cache misses
  size_t flushes_;              // flushes from cache to memory
  size_t evictions_;            // evictions from cache
  size_t dirty_evictions_;      // evictions of dirty lines from cache
  size_t invalidations_;        // invalidations due to coherence
  size_t local_interconnect_;   // intra-NUMA node interconnect messages
  size_t global_interconnect_;  // inter-NUMA node interconnect messages
  size_t memory_reads_;         // reads of main memory
  size_t memory_writes_;        // writes to main memory
}
```

Our simulator also has a simple timing model, in which it uses fixed latencies for different events. The default latencies are shown in table 1, but these are adjustable by editing the `latencies.h` file which defines these as constants. The latencies we used are based off latencies we found online for commonly used interconnects and memory systems. The NUMA distance on `andrew` is 2, so we set the global interconnect latency to be twice the local.

Event	**Latency**
Cache Access Latency	$1ns$
Memory Access Latency	$100ns$
Local Interconnect Latency	$1ns$
Global Interconnect Latency	$2ns$

Table 1: Default Latencies

The combination of these latencies and the stats we collect allows us to get rough estimates of how long the memory accesses of programs should take. We present the latency reports as part of the output when the user runs the program with stat reporting enabled.

3.4 The Architecture

We used C++ to write the simulator for a few reasons. Firstly, the traces are very large (∼100MB+) so we needed a performant language to get through the traces in a reasonable amount of time. Secondly, we are modeling a lot of physical objects such as caches and directories, so it allowed us to encapsulate these objects into classes which made them a lot easier to work with than if we had used C. It also meant we could reduce repetition in our code by doing things like having an abstract cache block class which we could inherit from to model other block types. The simulator is divided into 5 main classes - `NUMANode`, `Cache`, `Directory`, `Interconnect`, and `CacheBlock`.

Given arguments about how many processors are in the node and what coherence protocol is being used, a `NUMANode` is responsible for setting up a `Directory`, `Interconnect`, and one or more `Cache`s. The main function reads the trace file and sends the reads/writes to the relevant `NUMANode` which passes them on to the correct cache. It is also responsible for aggregating stats from all entities in the node once the trace has been ran.

The `Interconnect` class holds the logic for communicating between `Cache`s and their `Directory`, as well as between different `NUMANode`s. It keeps track of how many messages have been sent from a `Cache` to a `Directory` and vice versa, as well as how many messages have been sent between NUMA nodes which allows us to account for the different latencies of the inter- vs intra-node messages. The full list of interconnect messages the simulator supports are shown below in tables 2 and 3.

Message Type	Data Sent	Reason
BusRd	Address	Cache wants to read line at the given address
BusRdX	Address	Cache wants to write line at the given address
Data	Cache line	Cache is responding to directory request to fetch data at some address (usually because the cache has a exclusive modified copy and another cache wants to read it)
Eviction	Address	Cache has evicted the data at the given address and thus is no longer sharing it
Broadcast	Address, cache line	In MOESI, when a cache writes to a line it owns it must broadcast the data to all caches that are sharing that line, so notify the directory to do so

Table 2: Cache → Directory Messages

Message Type	Data Sent	Reason
ReadData	Exclusive bit, cache line	Send a cache back a cache line after it asked to read it, and tell the cache if it has an exclusive copy of it
WriteData	Cache line	Send a cache back a cache line after it asked to write it
Fetch	Address	Request that the cache sends its data and transitions to a sharing state (or in MOESI, just send data and remain in owning state)
Invalidate	Address	Invalidate a line in a sharing cache after a different cache got exclusive access to the line

Table 3: Directory → Cache Messages

We model a `Cache` as follows:

```
struct Set {
  Set(int associativity, Protocol protocol);
  std::vector<CacheBlock*> blocks_;
};

class Cache {
 public:
  Cache(int id, int numa_node, int s, int E, int b, Protocol protocol=MESI);
 private:
  std::vector<Set> sets_;
}
```

Where the relevant `CacheBlock` type is constructed to make up the `Set` based on what protocol is specified (i.e. MSI/MESI/MOESI), and $|\texttt{sets\_}| = 2^{\texttt{S}}$, $|\texttt{blocks\_}| = \texttt{E}$. The `Cache` class is responsible for servicing read/write requests to a given address, updating the state of the `CacheBlock`s and communicating with the `Directory` when necessary to send coherence messages.

The `Directory` maintains the current state of every line of memory in the `NUMANode`. It responds to coherence messages from `Caches` by updating its internal state and sending more coherence messages to other caches across the system. For example, upon receiving a `BusRdX`, it would send an `Invalidate` message to all other caches sharing that line, so they have to re-request a fresh copy of the line on their next read.

```
struct DirectoryLine {
  DirectoryState state_;         // U, S, EM
  std::vector<bool> presence_;
  int owner_;                    // in MOESI, the cache with the block in O state
};

// defines a mapping { memory line address -> DirectoryLine }
class Directory {
 private:
  std::unordered_map<unsigned long, DirectoryLine *> directory_;
};
```

`CacheBlock` is an abstract class from which the three block types we implemented - `MSIBlock`, `MESIBlock`, and `MOESIBlock`, inherit from. These classes are responsible for storing the state of the block based on what protocol they implement, and correctly transitioning between states after being read/written. After reads/writes, they return the interconnect message that should be sent to the `Directory` based on what their new state is. They keep stats on the events that have happened, such as hits, misses, evictions, invalidations etc.

3.5 Programs

3.5.1 Locks

In order to implement the lock programs, we took advantage of the atomic types from the C++ standard library. This provided an easy way to use atomic operations such as increment and exchange without the need to explicitly write assembly code.

The test-and-set and test-and-test-and-set locks are implemented using `atomic_exchange` function. This is essentially a test-and-set operation that we can perform on the lock variable. In the implementation of the ticket lock, two counters are used – one to indicate the next ticket and another to indicate the ticket currently being *served*. Atomic operations of load, store, and increment are used to implement this as a thread needs to increment to set the next ticket, load the current ticket as it waits, and store the new current ticket when it unlocks.

The array based lock is implemented with an array size of 128. This number was chosen arbitrarily, but is sufficient for supporting a reasonable large number of threads. In a typical array based lock implementation, an atomic circular increment is used (i.e. perform `x := (x + 1) % n` atomically). This is both difficult to implement and expensive to run. Instead, we use an atomic fetch and increment followed by a non-atomic modulo. All later indexing into the array likewise uses a modulo. Since the index into the array becomes ever increasing, this can lead to correctness issues after billions of uses of the lock. However, this version was sufficient for our tests and easier to implement. We have an additional implementation of the lock where each array element is aligned and on separate cache lines.

3.5.2 Evaluation

We wrote a simple program to evaluate the locks. In the program, threads repeatedly acquire the lock, increment a shared variable, and release the lock. In order to artificially create more contention, we have each thread sleep for one second while holding the lock. In order to balance the amount of time it took to run the traces and obtaining more data, we chose for each thread to make 10 increments. We ran this program with each lock type with 2, 4, 8, 16, and 32 threads.

To evaluate with more realistic programs, we also implemented a simple concurrent binary search tree which allows for insertions only. The general approach for this was to lock two nodes of the tree at a time as it traversed down to find where to insert. To balance run time, amount of data, and managing very large file sizes, we had each thread make 100 insertions of random integers. This was done with each lock type and 8 threads.

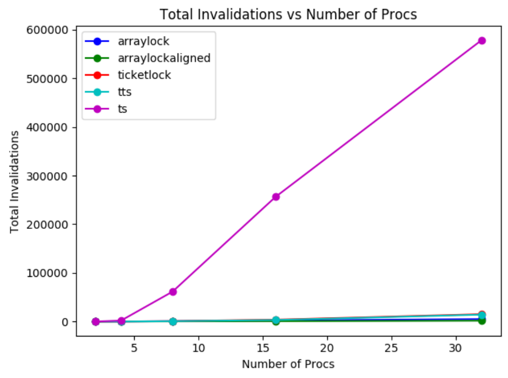
The main thread in both of these programs was just responsible for spawning and joining threads and did not perform the locking/unlocking computation done by the children threads. This was intentional, as the work done by thread 0 to set up the program was substantially more expensive than the simple operations every other thread was performing with the locks, which lead to the stats collected about the locks being diluted greatly by the large number of operations in setup. We added the `-A` flag in order to isolate the stats about the lock performance.

4 Results

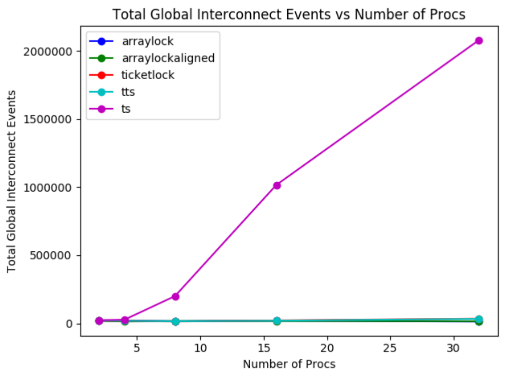
We wrote a script to generate plots for all of the metrics we collected, comparing lock types as well as cache coherence protocols. The most interesting findings are presented below.

4.1 Test-and-Set Locks

One of the most striking findings was how poorly regular test-and-set locks performed relative to the other locks. The cause for this difference in performance is due to the cache coherence behaviour of the lock. Since it tries to perform a test-and-set every time it tries to see if the lock is available (requiring an exclusive copy of the line), there is an invalidation on every lock acquisition attempt. This is seen in figure 6 which shows that the invalidations and interconnect traffic between NUMA nodes is orders of magnitude higher than all other locks. This resulted in up to $\sim$10x slowdown on larger processor counts (see figure 8) for test and set locks compared to the next slowest lock, as the invalidations meant that main memory had to be accessed frequently which is expensive compared to cache accesses.



(a) Invalidations



(b) Interconnect

Figure 6: Test-and-Set Performance (MESI)

4.2 Highest Performance

Firstly, it's worth noting the difference in the number of reads/writes necessary to implement each lock, because this varies between the locks, with the simpler test-and-set based locks performing $\sim 20\%$ fewer instrutions than the arraylock and ticketlock (see figure 7). Our simulation showed that this made a big difference, with more cache-performant locks having slower estimated times due to this overhead.

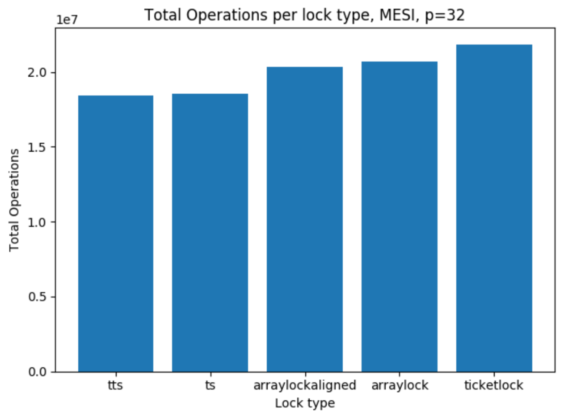
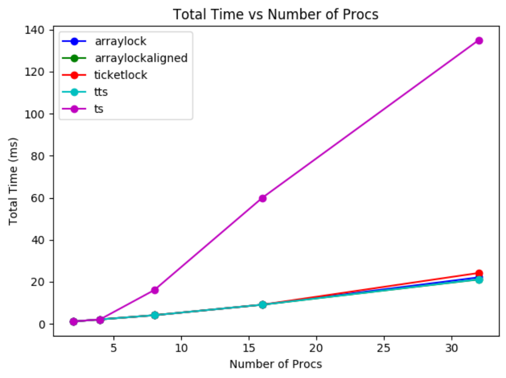
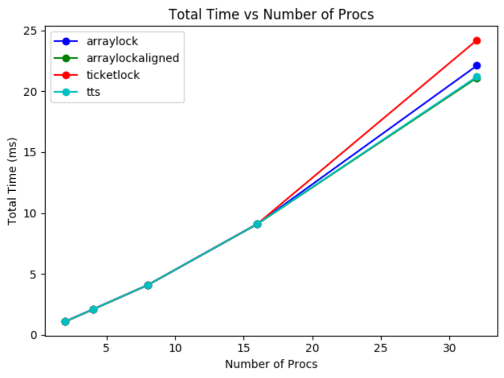


Figure 7: Total operations of each lock type

From our comparisons, the fastest lock was the aligned arraylock. Figure 8 below shows the estimated runtimes the simulator calculated for each of the programs, which is based off the stats collected and our estimated latencies.



(a) With `ts_lock`

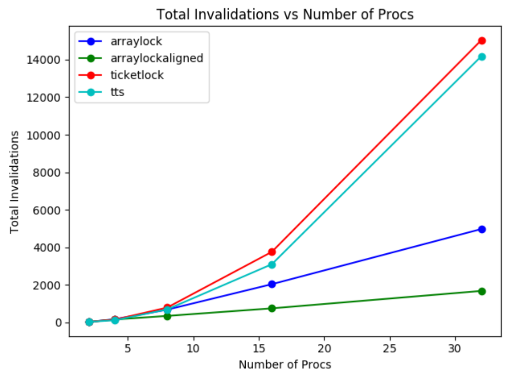


(b) Without `ts_lock`

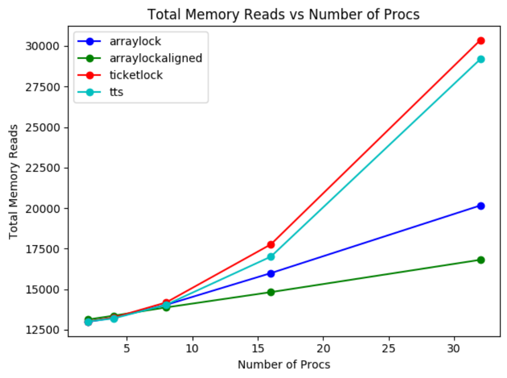
Figure 8: Total Estimated Program Time (MESI)

The aligned arraylock far outperformed any other locks in terms of having fewer memory reads due to missing infrequently in the cache. It also had the fewest number of invalidations due to the fact that it has $O(1)$ invalidations per cache per lock release while ticketlock has $O(P)$ and test-and-test-and-set has $O(P^2)$. It's also interesting to see the implications of false sharing as demonstrated by the difference between the aligned and unaligned arraylocks, with the unaligned lock having $\sim$2x as many invalidations and memory reads as its aligned counterpart. The difference in performance in these areas is illustrated below in figure 9.

The test-and-test-and-set lock (tts lock) performed second best even though its cache performance was far worse than the unaligned arraylock, which is due to the fact that it requires fewer memory operations and thus less time to acquire the lock (see figure 7. The ticketlock suffers from both relatively many operations as well as poor cache performance, so it performs the worst. An interesting thing to note is that while theoretically the tts lock has $O(P^2)$ invalidations and interconnect traffic, we saw $O(P)$ in our test programs. This was due to the fact that upon the release of the lock, all copies of the lock are invalidated. Since threads require an exclusive ownership of the line to do the test-and-set, at most one cache's line is invalidated per test-and-set attempt since all threads try to test-and-set at the same time.



(a) Invalidations



(b) Memory Accesses

Figure 9: Performance Characteristics of Arraylock (MESI)

4.3 Cache Coherence Protocols

We also compared performance of the cache coherence protocols we implemented (MSI, MESI, MOESI) and found some interesting results. The difference between MSI and MESI was most clearly exhibited by the aligned arraylock, where the benefits of the exclusive state were seen. As figure 10 shows, the lock had a constant number of fewer reads regardless of the number of processors. The reason for this is when a thread wants to lock the lock, it must increment the index, which is a read-write operation. In MSI, this results in a `BusRd`, during which memory is read, and then a `BusRdX`, during which memory is read again before sending the `WriteData` message. However, in MSI, the processor can get an exclusive copy of the line after the first `BusRd` and not have to send a `BusRdX` the second time.

For the less cache-friendly locks such as ticketlock, we observed noticeable differences between MOESI and the two other protocols. As shown in figure 11a), there was greater interconnect traffic with MOESI than the other protocols, which is explained by figure 11b), which shows at higher processor counts, MOESI is giving a 50% reduction in memory accesses. The reason for this is that the owning cache is serving memory lines to the other caches instead of flushing to memory, which

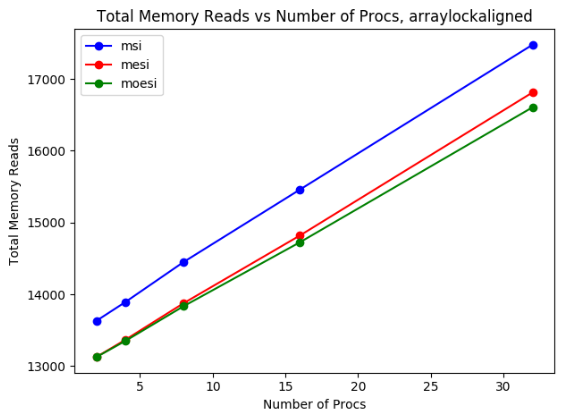
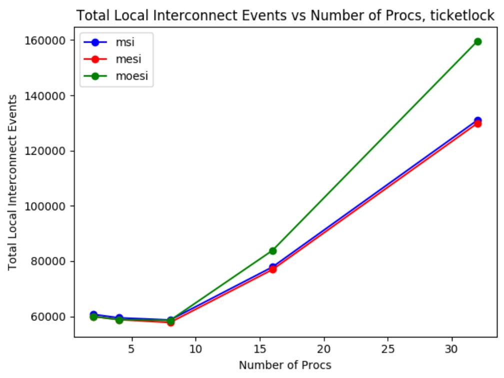
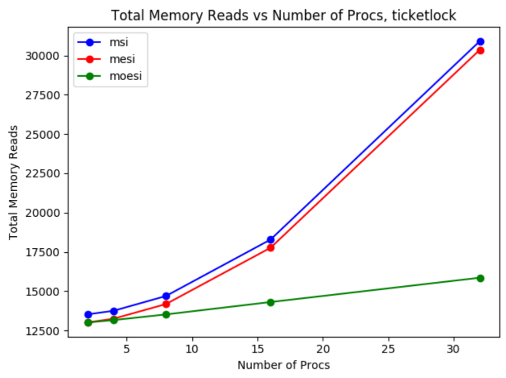


Figure 10: MESI performs better than MSI on `arraylock_aligned`

causes the interconnect traffic to increase. Since interconnect events are far cheaper than memory accesses (100x cheaper in our model), this shows the benefits of the MOESI protocol.



(a) Interconnect Traffic



(b) Memory Reads

Figure 11: Performance of MOESI vs MESI/MSI

We were also able to observe this behaviour in other programs. In the binary search tree program with the test-and-set lock, we were able to see that the MOESI protocol led to substantially fewer memory accesses compared to other protocols (figure 12), as the owning cache can again serve the data instead of going to memory which leads to a speedup.

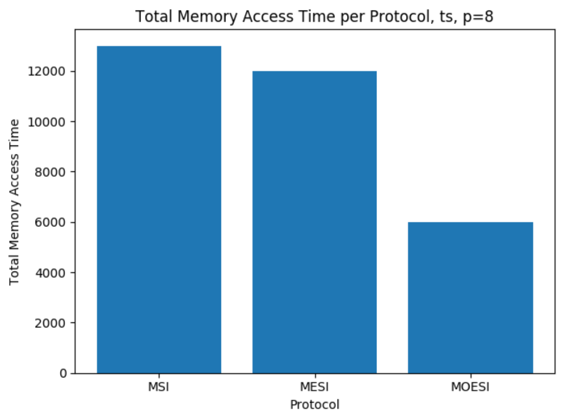


Figure 12: MOESI outperforms other protocols with `ts` on BST program

5 Conclusion

Our findings above illustrate that effectiveness of our cache simulator in analysing parallel programs and different cache coherence protocols. Through the statistics provided by the simulator, we found that aligned arraylocks far outperform the other lock types we implementedin terms of cache performance, at the cost of higher memory utilization. However, our simulator estimates that test-and-test-and-set locks operate at roughly the same overall latency as the arraylock due to lower cost of acquiring/releasing the lock. Our findings also demonstrate the benefits of the MOESI cache protocol over the MESI and MSI protocols, due to the great reduction in main memory accesses under certain workloads.

Overall, we feel the project allowed us to explore interesting ideas discussed in class in more depth, and gave us a greater understanding of the importance of writing cache-friendly code, especially on multiprocessor shared address space machines.

6 References

http://www.cs.cmu.edu/afs/cs/academic/class/15418-f20/public/lectures/10_cachecoherence1.pdf

http://www.cs.cmu.edu/afs/cs/academic/class/15418-f20/public/lectures/11_directorycoherence.pdf http://www.cs.cmu.edu/afs/cs/academic/class/15418-f20/public/lectures/17_synchronization.pdf

https://en.wikipedia.org/wiki/Directory-based_coherence

https://people.eecs.berkeley.edu/~pattrsn/252F96/Lecture18.pdf

D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta and J. Hennessy, ”The directory-based cache coherence protocol for the DASH multiprocessor,” [1990] Proceedings. The 17th Annual International Symposium on Computer Architecture, Seattle, WA, USA, 1990, pp. 148-159, doi: 10.1109/ISCA.1990.134520.

7 Work Distribution

7.1 Task Breakdown

Task	**Performed By**
Making the proposal doc	Sam
Writing the entire simulator	Sam
Making the checkpoint doc	Sam
Writing a script to generate all traces and run simulations on each	Sam
Writing the pin tool	Brian
Writing lock implementations and parallel BST	Brian
Writing a script to generate plots for simulator output	Brian and Sam
Writing the final doc	Brian and Sam
Making final presentation	Sam

7.2 Final Distribution

Sam: x%

Brian: y%