ECE 666 FINAL REPORT

Nikktiha Subbiah & Vaibhav Ramachandran

May 4, 2019

Abstract

This project implements and analyses two different self-invalidation schemes, namely, dynamic self-invalidation and last-touch prediction to reduce some of the coherence latency incurred in shared memory multiprocessors. The schemes were implemented using gem5 for an MSI coherence protocol with two levels of caches. Upon successful implementation of the schemes, they were tested across five different splash benchmarks to provide a detailed analysis of the schemes alongside the results from the original paper.

1 Introduction

Processors with caches usually incur three different kinds of memory misses, namely, compulsory, conflict and capacity misses. Various optimizations have been developed to reduce these miss latency overheads. However, multicores incur an additional type of cache miss called coherence misses. This is a result of trying to maintain writes atomic and enabling some sort of consistency, it can be either sequential, weak or relaxed consistency models. In an effort to reduce the coherence invalidation overheads, various schemes have been proposed. This project talks about two such schemes - Dynamic self-invalidation and Last Touch prediction. The idea behind both schemes is to invalidate a cache block from a processor's private L1 cache before it gets an invalidation request from the directory as a result of another processor's write. Even if the goal of both the schemes is the same, the implementation is quite different.

In the Dynamic self-invalidation [1] technique, the directory dynamically identifies which blocks should be self-invalidated and provides this information to the cache on a miss. The cache controller later self-invalidates these blocks at an appropriate time. This helps decrease the number of explicit invalidations that the directory sends out without significantly increasing the number of misses. The paper talks about two different ways to identify the self-invalidation blocks and two ways the cache can self-invalidate the block. However, only one of these ways were picked for the project to perform a detailed analysis and comparison.

In the Last-Touch Prediction [2] technique, each cache block maintains a unique instruction trace that is encoded as a signature. So every time an instruction touches a cache block, the PC of the instruction gets encoded in the signature using *truncated addition*. When an invalidation comes in for the cache block, it's signature gets stored in a global signature table. Upon subsequent accesses to the cache block, the last-touch prediction begins. Every time a subsequent instruction touches the cache block, it's PC value gets encoded into the cache block's unique signature. But this time, the signature is checked against the global signature table, and if a match is found, the block self-invalidates itself. This reduces the amount of time the directory spends sending out invalidations. It also reduces time spent waiting on change of ownership of heavily write-shared cache blocks. The paper mentioned two ways of implementing the signature table, a global table that simply holds all the signatures in a single list and a per block signature table that maintains a per block signature list. However, only the global signature table was implemented for this analysis and comparison.

2 Motivation

Most systems these days are shared memory systems wherein they use a single address space even if the memory is distributed across multiple nodes. These shared memory systems use caches to replicate and migrate shared data across processors to maintain the shared memory address space. This makes coherence protocols quite complicated and hard to manage. In order to provide write atomicity, only the writer needs to have a valid copy of the data which means all the other shared copies need to be invalidated. Coherence invalidations can increase the network/bus traffic quite a bit and add to the latency overhead of shared memory systems. Reducing coherence invalidations can help improve the performance of the system and still keep it consistent. One way to reduce the number of coherence invalidations would be to predict an upcoming invalidation, in other words, predict that another core will write to the cache block in the near future and invalidate your own block before the directory sends out an invalidation request. There have been several schemes to perform these self-invalidations however, this project only talks about two of the most popular techniques as mentioned in the previous section. These schemes can help reduce the coherence overhead thus improving performance with minimal hardware overhead.

3 Dynamic Self-Invalidation

The paper[1] integrates the self-invalidation process with the coherence protocol, which means that the entire self-invalidation process is done in hardware without any software intervention. The invalidation

process has three different steps:

1. Identify a cache block that needs self-invalidation
2. Perform the invalidation
3. Send an acknowledgement for the invalidation

While most techniques couple steps 1 and 2, DSI splits the two and performs them at two different places. The identification is done in the directory while the actual invalidation is done in the private L1 cache. It is important to note that the coherence protocol used in the paper is MSI and hence all the optimizations are aligned to MSI. The paper talks about two different methods to implement each of the three steps.

3.1 Identification of self-invalidation blocks

This step is done at the directory since the directory keeps track of all the readers and writers and would have a better idea of the pattern followed by each cache block. The blocks can be picked for self-invalidation by two different technique however, the idea behind the two is the same. When the directory is servicing a miss request, it uses this extra information to predict if the block is likely to be invalidated in the future, and conveys this information to the caching node with the response. The two techniques are: additional states and version number.

In additional states, as the name suggests, the directory goes to a different state when a self-invalidation block is given to the cache. Since this technique was not the one implemented in this project, we will not be discussing the detailed implementation.

The version number technique sets aside a 4-bit number to keep track of the various versions of the cache block in the system. The version number is only updated for self-invalidation blocks which means that there can still be other cache blocks with no version number associated with them if they do not have sufficient read-write sharing. The various times when a version number would be sent to the requesting cache would be if the cache block is present in the Modified state with a different writer or if it is a write miss on a shared block. The version number is updated every time a writer requests an exclusive block. If a cache misses but there is a tag match and the block is present but in invalid state, the corresponding version number is sent with the request for the block to the directory. The directory responds with a self-invalidate block if the current version number is different from the version number of the request. If the cache controller does not provide a version number (i.e., there is not a tag match), the directory responds with a normal block. The version number can always roll back to 0 and start over if it reaches the maximum value.

3.2 Performing the Self-Invalidation

The actual invalidation happens in the cache if the cache has received a self-invalidation block from the directory. The paper talks about two techniques to do this - using a FIFO buffer or at synchronisation points. This project uses the FIFO buffer approach and hence, we will only be talking about that.
A FIFO buffer of finite-size is used to hold the addresses of all the self-invalidation blocks coming in and when the buffer gets full, the first entry from the buffer is invalidated from the cache. There can be times when the address is probably already invalidated, in that case, the buffer entry is just discarded.

3.3 Acknowledging Invalidation Messages

This section talks about if the self-invalidation by cache needs to be informed to the directory. If the cache guarantees to self-invalidate the blocks at specific times then we can eliminate both the invalidation and acknowledgement messages. This technique however, only works for invalidation at synchronization points and since this project implements the FIFO buffer technique, this optimization was not implemented.

4 Last-Touch Prediction

The paper [2] also integrates the self-invalidation process with the coherence protocol, which means that there is no software intervention. The invalidation process steps are outlined below:

1. Encoding the PC of each instruction touching a cache block.
2. On a coherence invalidation or replacement, store the unique signature in a global signature table.
3. On subsequent accesses to the cache block, again perform the encoding of PC values.
4. Simultaneously check the global signature table for a matching signature.
5. On a match, invalidate the block before a coherence invalidation or a replacement occurs.

4.1 Encoding the PC value

This step is done in the L1 D-cache for every single cache access to a cache block. Upon an initial access to a cache block that is not present in the cache, the PC value is stored in the per-block signature entry. Subsequent instructions accessing the cache block encode their PC values in the signature entry using *truncated addition*. 30 bits were used for storing the encoded trace since that was the maximum number of bits that the paper [2] mentioned.

4.2 Storing the encoded instruction trace

When a coherence invalidation, downgrade or a replacement of the cache block occurs, the encoded trace for that cache block gets stored in a global signature table. The paper [2] mentions two ways of constructing the global signature table; create a single global list that holds all the signature values of all the cache blocks or create a per cache block signature list that will hold the individual traces of each cache block that comes in. Only the global list version was implemented for ease of implementation and because the performance drop was not significant compared to the per-block signature list [2].

4.3 Subsequent accesses to the cache block

Upon subsequent accesses to the same cache block, the process is repeated once more but with one extra step. Now, when the instruction's PC value is encoded in the per-block signature, it is simultaneously checked against the global signature list for a match. This is the main feature that allows Last-Touch Prediction to provide a performance improvement.

4.4 Eagerly invalidate the cache block

When the signature for the cache block matches one present in the global signature table, it self-invalidates the cache block and performs the necessary coherence actions that would normally take place on a coherence invalidation, downgrade or replacement. This is done as soon as a match for the signature is found in the global table, instead of waiting for an invalidation to arrive.
This eager self-invalidation allows execution time to reduce by eliminating the waiting time needed for exchanging ownership, downgrading or replacing the cache block. Since the directory is already updated and the cache block is invalidated in the L1 cache, the directory can directly service any incoming requests to the block without needing to forward them to the owner.

5 Experimental Methodology

5.1 Baseline

This project was coded and simulated using the Gem5 simulator. The ISA used here is Alpha while the memory model is Ruby. Since both the papers use the MSI protocol, our baseline had to use MSI as well. The already present MESI_Two_Level model was converted into MSI. The baseline works perfectly and has been tested before implementing the two schemes. Since the directory is implemented in the L2 cache, the MSI baseline also has two levels of cache. This isn't as much of a deviation from the paper that uses a single level of cache and a directory because the L2 cache in our baseline acts as a directory

and stores data while the actual directory only acts as an interface between the L2 cache, DMA and memory.

5.2 Benchmarks

Both the papers use a combination of SPEC, SPLASH and Berkeley Split-C benchmarks. However, gem5 has only the parallel versions of SPLASH benchmarks. Hence, only those were used for the analysis.

For both the DSI and LTP studies, five different SPLASH benchmarks - namely Ocean, Barnes, FMM, Raytrace and LU were used. The choice of the benchmarks was based on availability and range of inputs since not all the benchmarks can run for over 8 processors. Since the benchmarks work on matrices or pixel values and stop at once the computation is over, the caches could not be warmed up. Also, check-pointing required atomic CPU while not all benchmarks supported Atomic CPU therefore the only way to warm up the caches was to run the programs for larger inputs sizes.

6 Results and Analysis

6.1 DSI

Multiple simulation runs were done for various values of parameters to get a detailed analysis of the scheme. The default values of the parameters are 32KB L1 cache, 2MB L2 cache and FIFO buffer size of 16. All the simulations were run to completion before using the stats. If any of the parameters are not mentioned in the graphs then the default valuse have been used.

1. The first graph compares the performance of DSI for various benchmarks with respect to the baseline. As the graph shows, DSI does not perform well for all benchmarks. The best performance improvement can be seen for FMM while Raytrace and Barnes do not see much decrease in execution time. We can also see the as the number of cores increase, the performance gets better which is not a surprising trend. One of the reason why Raytrace and Barnes do not perform well could be because of their reuse trend.

2. The second set of graphs look at the impact of FIFO Buffer size on the performance. As the graphs suggest, as the FIFO buffer size increases, the execution time decreases and for some benchmarks, a small FIFO buffer actually degrades the performance. This is because the DSI scheme is invalidating cache blocks that have a potential for reuse ahead of time which leads more misses in the processor.

3. The paper uses a single level of cache however, this project uses two levels of caches, therefore, the

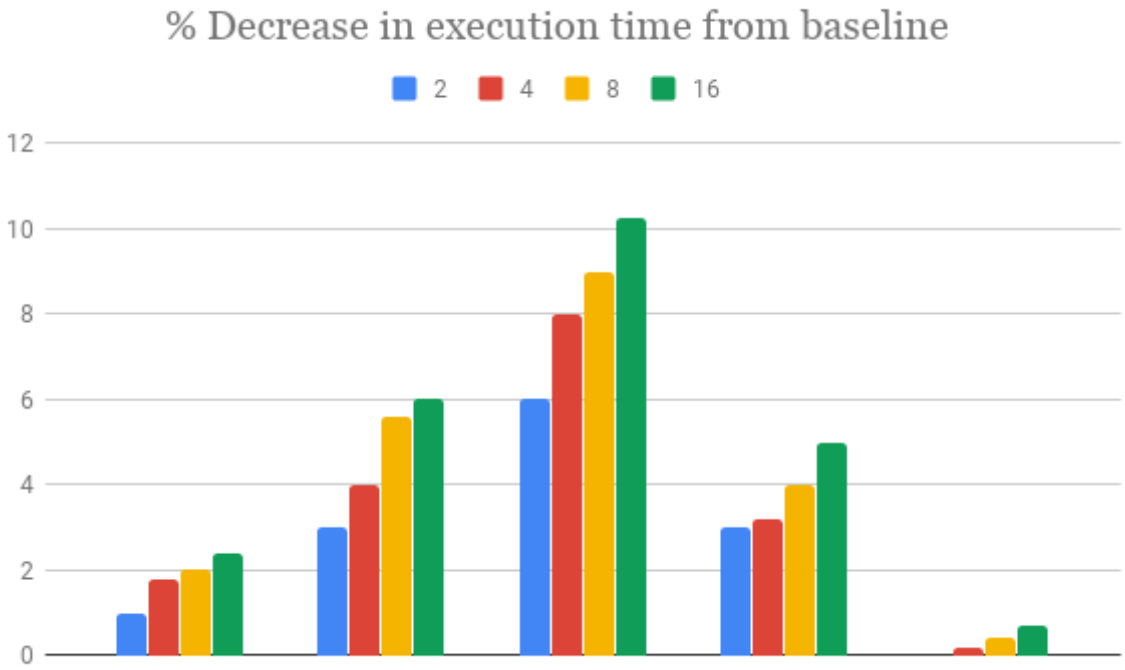
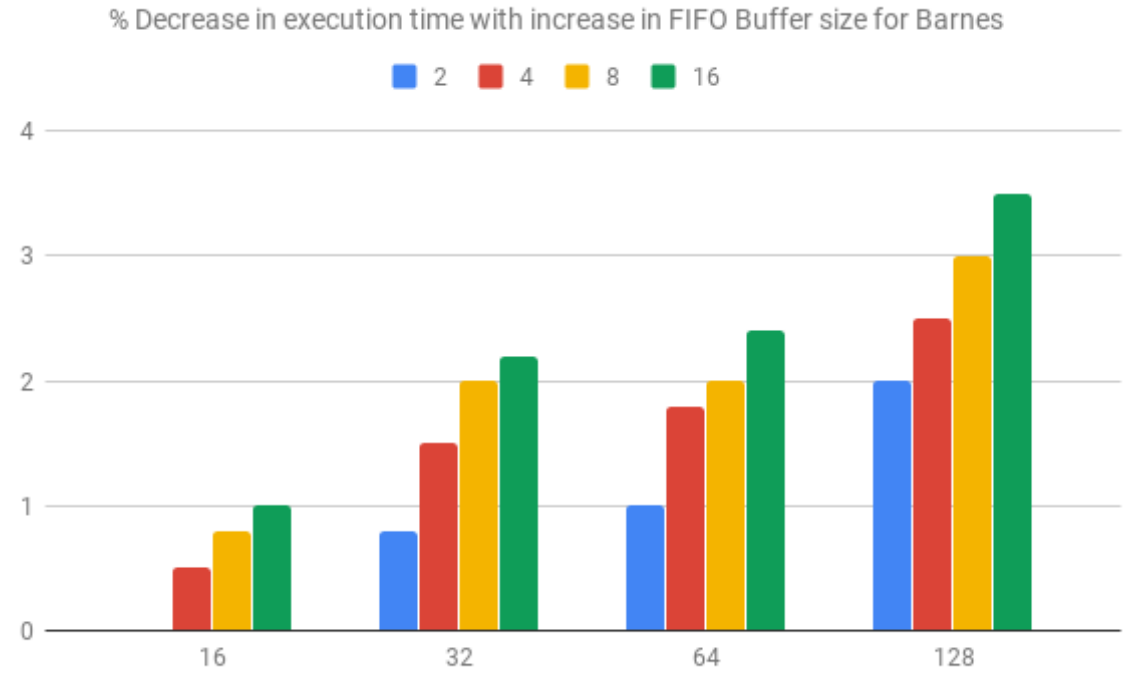
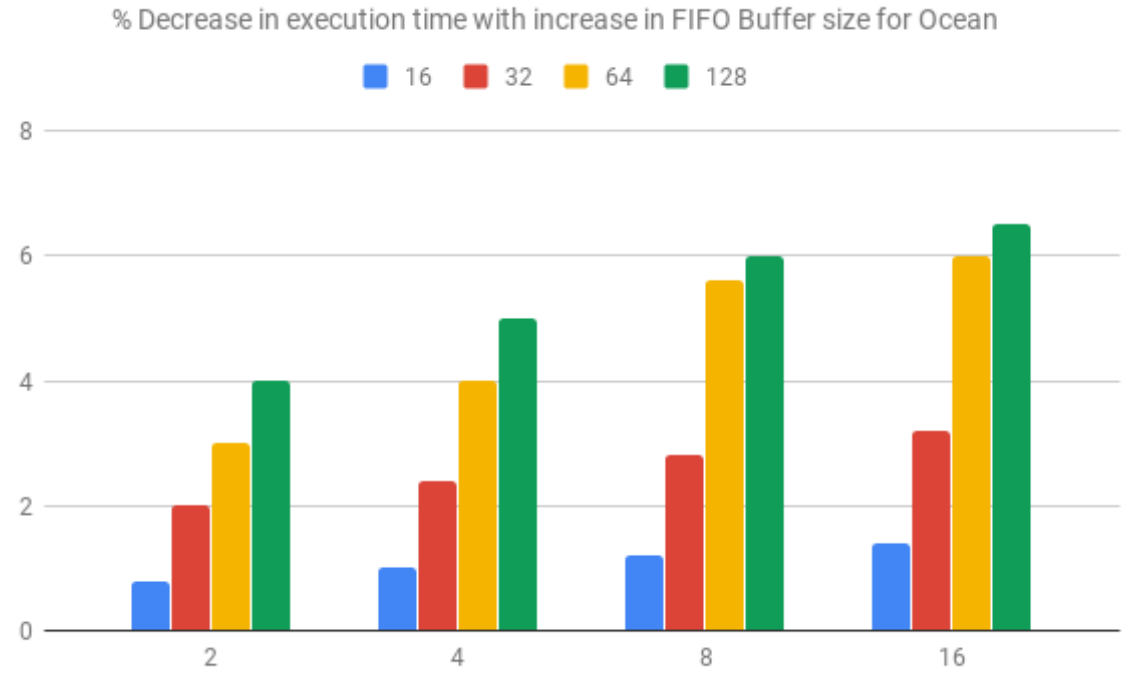
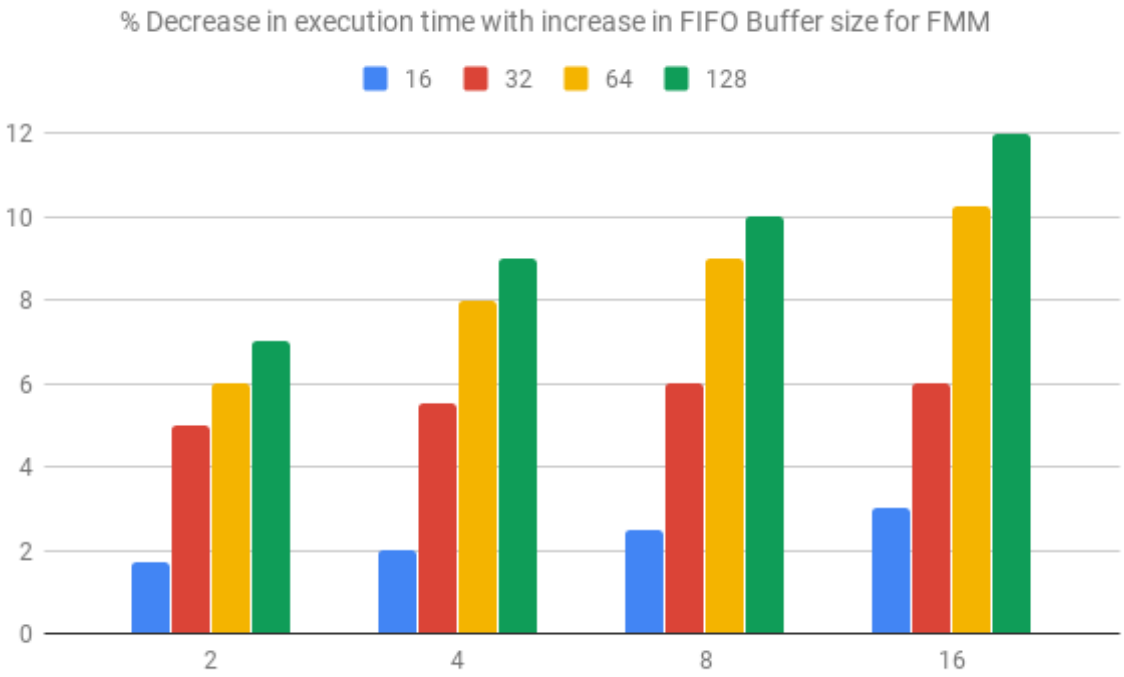


Figure 1: The order of benchmarks are left to right: Barnes, Ocean, FMM, LU, Raytrace.



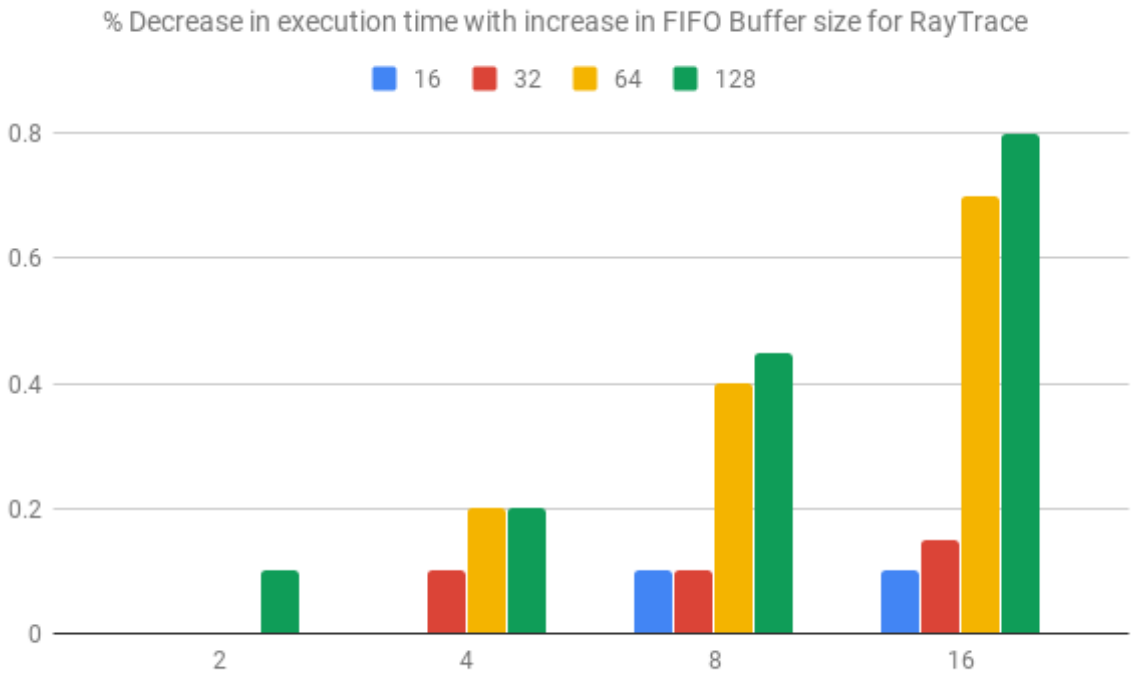
size of both the caches were changed and the execution times were analysed. Only the graphs for FMM has been shown here since there would be too many graphs otherwise and FMM was picked since it had the best performance so far. As the size of L1 cache increases, the performance gets





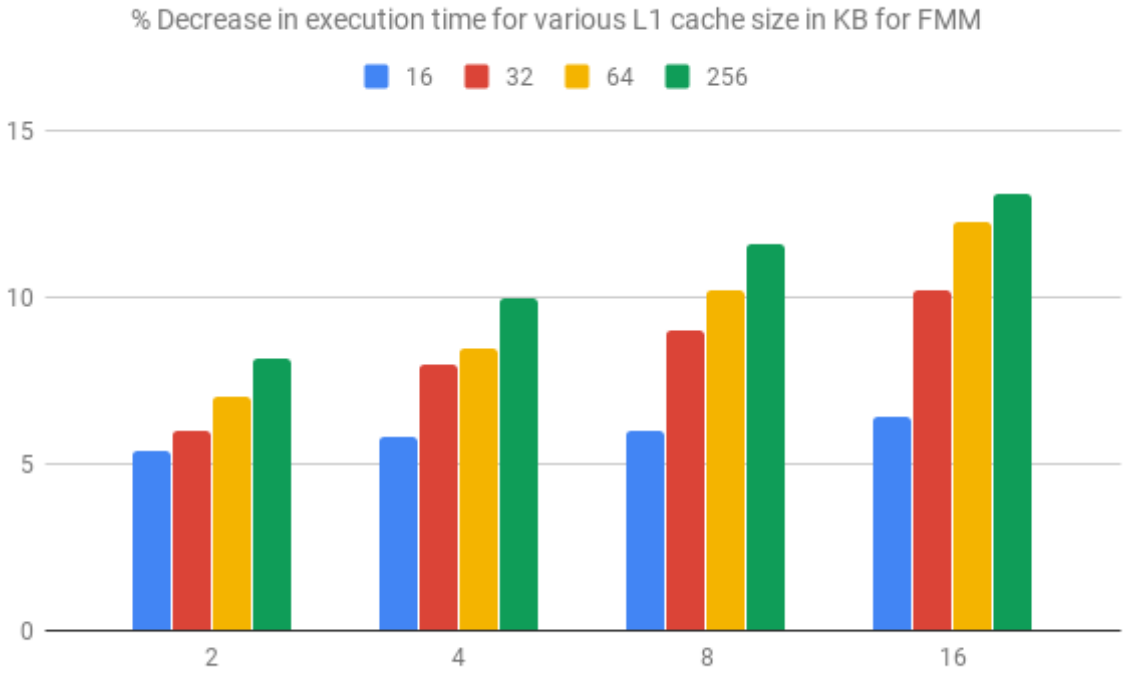
better, this is because the number capacity misses has been reduced. DSI probably does not play a huge part in this analysis until the cache size is big enough to eliminate capacity misses.

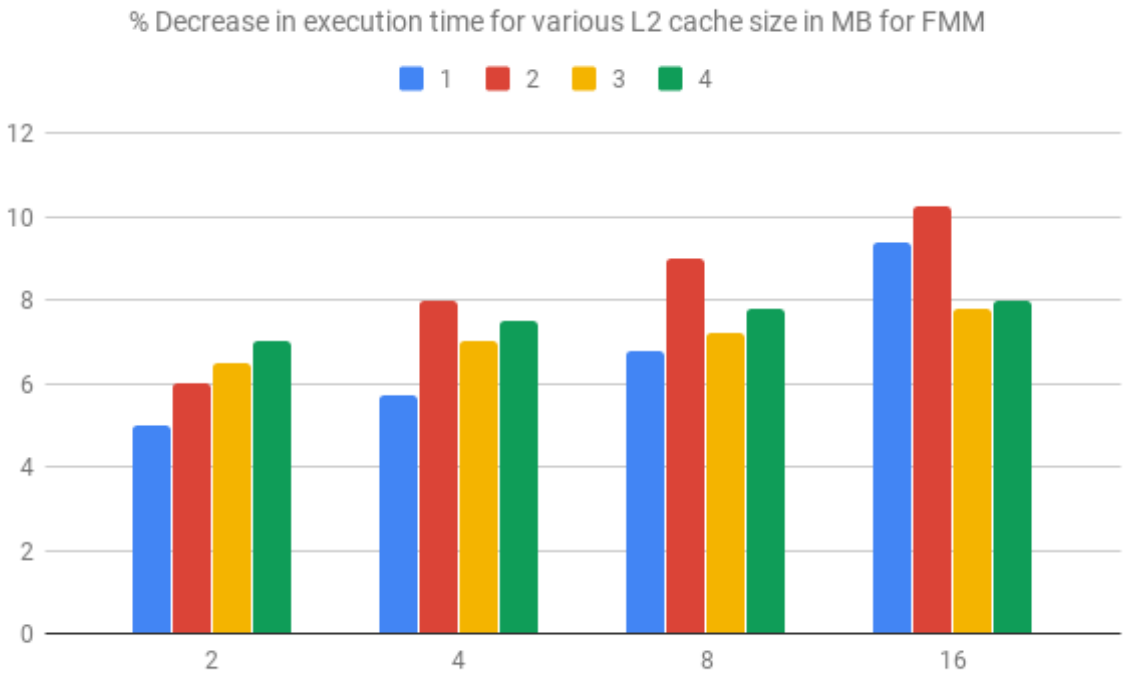
4. The L2 cache size was varied for this analysis. As it can be observed from the graph, the performance does get better with increasing cache size however, this trend has nothing to do with DSI



since the FIFO buffer is in the L1 cache. The purpose of this analysis was to provide a thorough overview with all possible parameters.

The paper uses Ocean and Barnes as the two SPLASH benchmarks and the results from this project is consistent with the results shown by the paper.





6.2 Interesting bugs that were fixed

1. There was a livelock in the design that would only show up after over 50M instructions, this meant that debug flags could not be used since the output text file would be over 25GB in size. The livelock was fixed by the method of elimination and careful checks for NACKs.

2. The lack of a strict request-response led to a deadlock in the system. Had to redo the invalidation and acknowledgement part to fix this.

6.3 LTP

Multiple simulation runs were also done for the benchmarks with different numbers of processors in order to get a detailed analysis of the performance of the improved scheme. Again, all the simulations were run to completion before grabbing the stats. The default values used in the simulations were as follows: L1 cache size: 128kB, L2 cache size: 1MB

These numbers were chosen to closely match the values that were used in the paper. The graph below compares the performance of LTP for various benchmarks with respect to the baseline numbers. As it can be seen, this implementation of LTP doesn't perform well for all the benchmarks, namely Raytrace and Barnes. Both these benchmarks did not offer much improvement in execution times when compared to the baseline. However, FMM was able to achieve a reasonable performance gain as compared to the other benchmarks. The difference in performance could be attributed to each benchmark's cache access patterns. The paper used the Barnes, Ocean and Raytrace benchmarks. Aside from the performance shown in the Raytrace benchmark, the performance trends shown by the other two benchmarks fit the trends discussed in the paper.

The paper mainly discussed about the improvements in invalidation predictions compared to other self-invalidation schemes, however since there was no mechanism currently present within Ruby to provide the data for this comparison, only a baseline performance comparison was done.

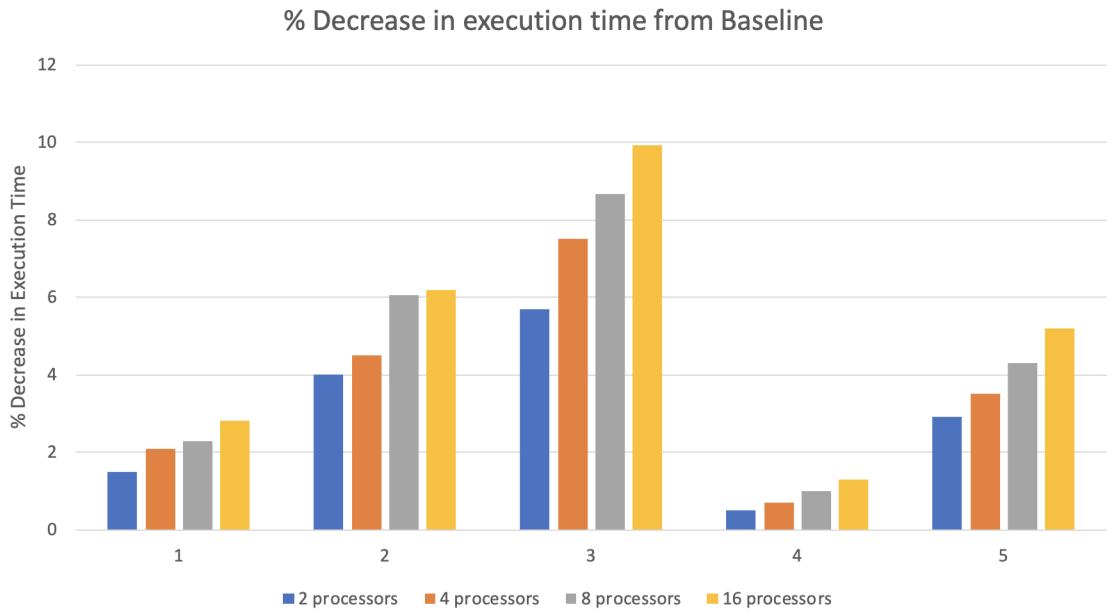


Figure 2: The order of benchmarks are left to right: Barnes, Ocean, FMM, Raytrace, LU.

7 Comparison between DSI and LTP

The LTP paper [2] mentions that on average, it achieves an 11% increase in performance compared to DSI. But as it can be seen from the graphs the two implementations performed similarly with LTP performing slightly better than DSI for most of the benchmarks.

8 Conclusion

Both implementations of self-invalidation schemes followed similar trends to the paper for a majority of the benchmarks that were run. The only benchmark without much performance improvement in either scheme was the Raytrace benchmark. While our implementations were not able to hit the high performance improvements gained in some of the other benchmarks like Ocean and Barnes, were were able to match the trends that they showed as we increased the number of processors running the programs.

9 Contribution

Nikkitha Subbiah: Dynamic Self-Invalidation

Vaibhav Ramachandran: Last-Touch Prediction

References

[1] Alvin R. Lebeck and David A. Wood *"Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors"*. ACM SIGARCH Computer Architecture News - Special Issue: Proceedings of the 22nd annual international symposium on Computer architecture (ISCA '95) Volume 23 Issue 2, May 1995.

[2] An-Chow Lai and Babak Falsafi *"Selective, Accurate and Timely Self-Invalidation using Last-Touch Prediction"*. Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA), June 2000.