MULTI-CORE EXECUTION-DRIVEN CACHE SIMULATOR

Authors: Shridhar Ganiger, Nishant Poorswani

Video Slides

1. SUMMARY:

We have implemented an execution-driven multi-core cache simulator with fast forward capabilities. This simulator supports different snooping based cache coherence schemes. It analyzes memory accesses at run time and reports various cache performance metrics. It has added features to visualize function specific cache statistics, comparison of snooping based protocols for a given program and coherence traffic.

2. BACKGROUND:

This project is an inspiration from Introduction to Computer Systems (15-213) Cache lab which is a trace-driven cache simulator for a uni-processor. Our idea is to build an execution-driven multi-core cache simulator with fast-forwarding capabilities. Caches play a vital role in determining the performance of parallel programs on a multi-core processor. One of the major challenges that multi-core processors face is the cache coherence problem. Cache is coherent when all the loads and stores to a given memory location behave correctly. There are two main methods to ensure cache coherence: a) snooping based cache coherence schemes b) directory based cache coherence schemes.

Snooping based cache coherence schemes work based on the idea that coherence-related activity is broadcast to all processors in the system. Cache controllers snoop on the bus to monitor memory operations, and react accordingly, to maintain cache coherence. There are a few snooping based cache coherence schemes like MI(VI), MSI, MESI and MOESI.

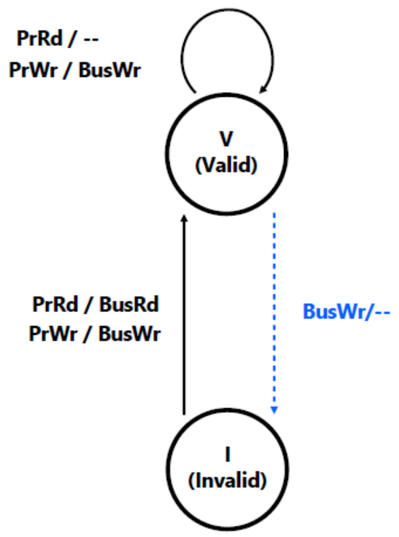


Figure 1: MI Protocol

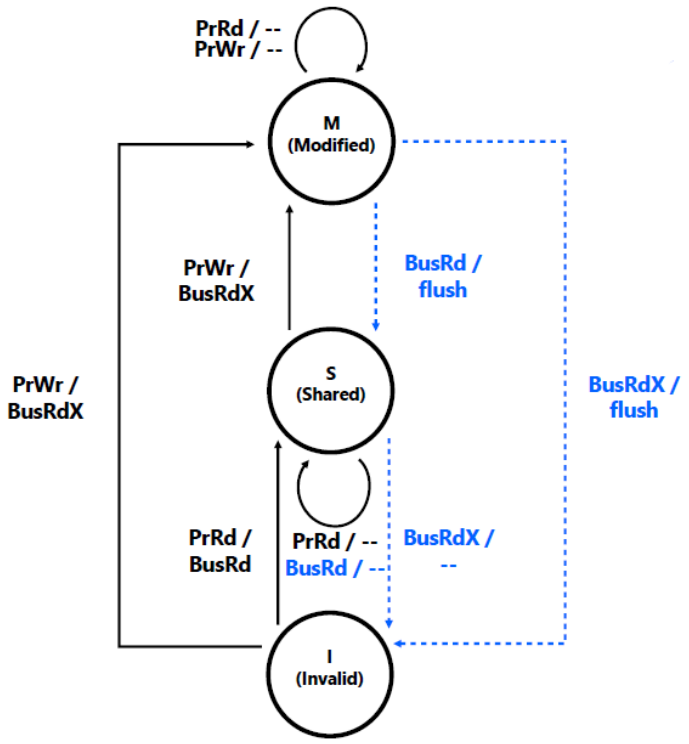


Figure 2: MSI Protocol

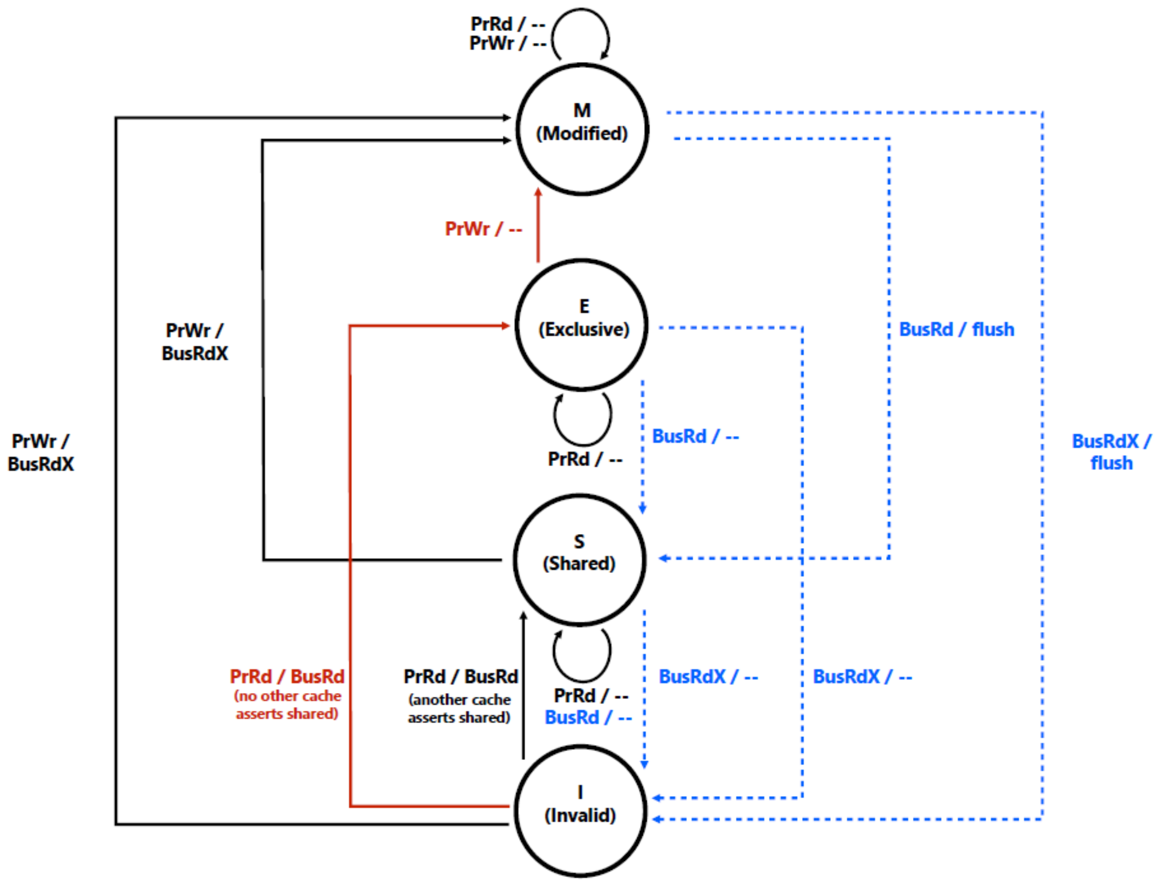


Figure 3: MESI Protocol

3. APPROACH:

- **Build a uni-processor cache simulator**

  We designed and implemented a uniprocessor cache simulator using C++ STL library. To implement a LRU policy, cache lines are implemented using lists, the recently used cache line is pushed back and the least recently used cache line is at the front of the list. The LRU line is erased whenever there is a need for an eviction. This was thoroughly tested by using the trace files used in Cache Lab (15213 - Introduction to Computer Systems).

- **Understand Pin Tool**
    - After reading through Intel Pin tool's reference manual and APIs we were able to log the memory access events in the right order at run time for a given program binary. To generate these memory events we used Pin's Instruction level granularity APIs which means that the desired callback functions would be called whenever there is a load/store.
    - After generating the memory trace files for a sample program we simulated these traces with our uni-processor cache simulator. The correctness of the results were verified against the reference simulator used in Cache Lab (15213 - Introduction to Computer Systems).
- **Execution-driven uni-processor cache simulator**
    - We initially planned to build a trace-driven cache simulator with traces generated by the Pin tool. However after speaking with Prof. Beckmann and further research, we realized that an execution-driven simulator would resolve the future memory access ordering problems for a multi-core processor.
    - For an execution-driven simulator, we needed to integrate our uni-processor cache simulator with the Pin tool. To do so, we needed to understand Pin tool's software and build architecture. This was very challenging as there wasn't enough documentation for the above. We needed to deep dive into the build infrastructure. We realized that there effectively was a hierarchy of 6 Makefiles to build the Pin tool. We

    modified the entire build architecture to remove the redundant tools and also to integrate our uni-processor cache simulator.
  - So now, whenever Pin tracks a memory access event, it triggers our uni-processor cache simulator which simulates the cache at run time. We also create a trace of these memory accesses which we use to verify the correctness of our event-driven cache simulator. The correctness was again verified using the reference cache simulator from Cache Lab.
- **Build execution-driven multi-core cache simulator and implement snooping based cache-coherence protocols (MI/MSI/MESI)**
  - For a uni-processor cache simulator the cache was implemented as a C++ class. To extend this to a multi-core cache simulator we instantiated multiple cache objects which is equal to the number of cores we intend to simulate.
  - Since we don't focus on the timing aspects of a memory hierarchy, our multi-core cache simulator assumes a point-to-point infinite bandwidth bus interconnect.
  - Each cache coherence protocol has its own protocol specific controller that handles processor action, generates bus action and handles bus action. This controller is again implemented as a C++ class and everytime a protocol is simulated an object of this class is instantiated.
  - Verifying correctness of these protocols was a major challenge since we were not trace-driven anymore. To solve this problem we initially took a sample of memory accesses and printed out the state of the cache lines to

verify their correctness. Since this was not a full proof method, we also spoke to Prof. Brian Railing regarding this. He suggested we have multiple checks in the code to verify the state correctness. We incorporated this idea with the help of multiple assert statements. This helped fish out a crucial bug in our MESI implementation.

- **Making our cache simulator feature rich**
    - Till now our cache simulator was just printing cache metrics on the terminal. We added visualization as pictures speak louder than just numbers. This was done with the help of pandas and other libraries like matplotlib in Python.
    - Another useful feature for modern day multi-threaded applications is the cache coherence traffic. This gives a good perspective on the protocol performance and performance degradation due to true/false sharing. As part of coherent traffic we log the number of bus requests, bus read exclusives and invalidations.
    - While going through the labs and using the Perf tool, we realized that we can improve speed-up substantially by optimizing the region of the code that is not cache friendly. Hence, we decided to have function specific cache performance metrics.
        - This added a new challenge since we were using Pin tool's instruction level granularity APIs. We had to re-explore Pin to see if the APIs were capable of providing a function specific event. We found that Pin also supports routine level granularity APIs. We had

    - to change our base code to incorporate this feature on top of instruction level simulation.
    - We also realized that just simulating function specific memory events will not give accurate simulation results. To solve this we came up with a Fast-forwarding technique where we warm-up our microarchitectural state (i.e. cache in our case) in preparation for accurate simulation.
  - We also have a feature to compare how different snooping based cache protocols fare for a given parallel program.
- **Creating a infrastructure to accommodate various features**

  We wrote a user-friendly shell script to take in inputs for various features and start simulation based on the user's inputs. More information is available in Section 4 (Multi-core cache simulator: Inputs/outputs)

4. Multi-core cache simulator: Inputs/Outputs

- Usage: ./cacheSim.sh [options]

-s	2^s sets
-E	E cache lines
-b	2^b bytes per cache line
-p	Type of snooping based cache coherence protocol MI/MSI/MESI/all
-n	Number of threads
-o	Specify executable to simulate

**userFuncs.in: Specify functions to simulate

    - Cache size: s, E and b determine the cache size. The user can vary these parameters and analyze how their parallel program would perform on different machines with varying cache architectures.
    - Protocol: Users can choose the protocol which they want to simulate by specifying it ahead of the -p flag. We also support all as an argument, where it compares all the protocols for a specific parallel program.
    - Number of threads: Users should specify the number of threads as an argument by using the -n flag.
    - Program Binary: Users should specify the program they want to simulate as an argument by using the -o flag.
    - Userfuncs.in: Users can specify the functions for which they want cache specific analysis.

Sample Usage: ./cacheSim.sh -s 6 -E 8 -b 6 -p MESI -n 8 -o perThreadCounter

- Outputs:
    - Total number of hits, misses and evictions
    - Per thread hits, misses and evictions
    - Per function hits, misses and evictions
    - Coherence traffic: Bus Requests, Bus Read exclusives and invalidations
    - Ability to simulate all the protocols (MI/MSI/MESI) for a specific problem and compare stats like hits, misses, evictions, bus Requests, bus Read exclusives and invalidations

NOTE: All the above outputs are available on the command line as well as through graphs.

5. **CHALLENGE:**

- Maintaining the order of memory accesses for accurate simulation
- Building an execution-driven cache simulator
- Generate cache stats for a specific function
- Verification of the correctness of implemented snooping based cache coherence schemes

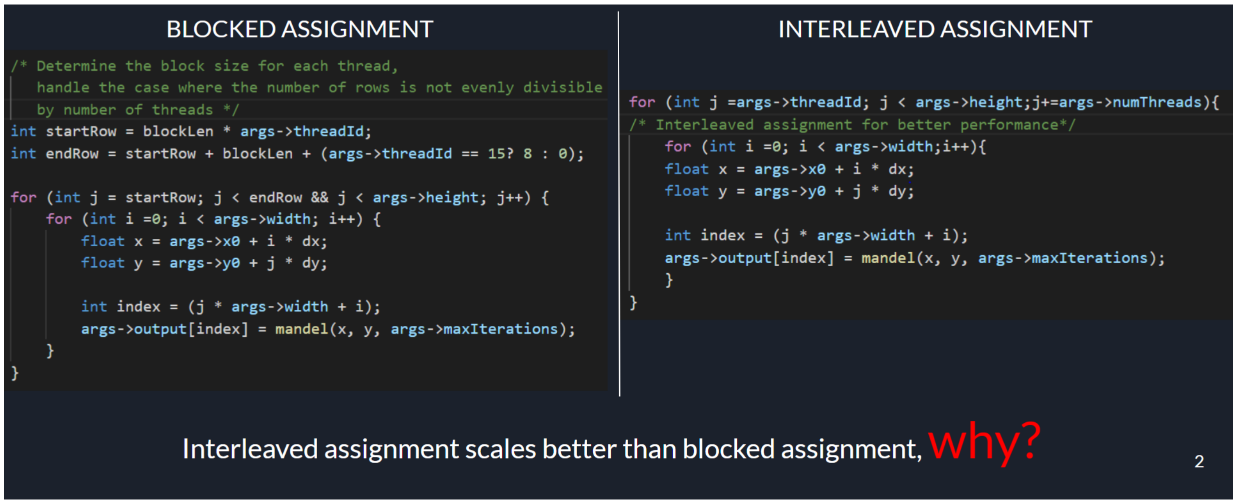
All the above challenges have been solved and explained in detail in Section 3 (APPROACH).

6. **RESULTS:**

The motivation for building a multi-core cache simulator arised from working on the assignments for the class. In each assignment, writing cache friendly code played a vital role in achieving speedup. To validate our analysis performed during the assignments, we decided to perform cache simulation on some of our assignments and course material. For all the below experiments we configured the cache parameters according to the GHC machines. (s=6, E=8 and b=6)

- Assignment 1: Mandelbrot (view 1)
    - In our Assignment 1 we had to obtain speed-up close to 8x on the Mandelbrot problem while using 8 threads. During the lab we found out that the interleaved assignment between threads performs better than the blocked assignment (Figure 1) . We used our cache simulator to analyze why?

- In Figure 2, we have one of the outputs (per thread hits/misses/evictions) compared between interleaved and blocked assignment. From the graph it is clearly visible that in the blocked assignments there was workload imbalance where certain threads were doing more work compared to the others. In the interleaved assignment the workload is evenly distributed as seen from the graphs.



BLOCKED ASSIGNMENT

```c
/* Determine the block size for each thread,
   handle the case where the number of rows is not evenly divisible
   by number of threads */
int startRow = blockLen * args->threadId;
int endRow = startRow + blockLen + (args->threadId == 15? 8 : 0);

for (int j = startRow; j < endRow && j < args->height; j++) {
    for (int i =0; i < args->width; i++) {
        float x = args->x0 + i * dx;
        float y = args->y0 + j * dy;

        int index = (j * args->width + i);
        args->output[index] = mandel(x, y, args->maxIterations);
    }
}
```

INTERLEAVED ASSIGNMENT

```c
for (int j =args->threadId; j < args->height;j+=args->numThreads){
/* Interleaved assignment for better performance*/
    for (int i =0; i < args->width;i++){
    float x = args->x0 + i * dx;
    float y = args->y0 + j * dy;

    int index = (j * args->width + i);
    args->output[index] = mandel(x, y, args->maxIterations);
    }
}
```

Interleaved assignment scales better than blocked assignment, why?

2

Figure 4

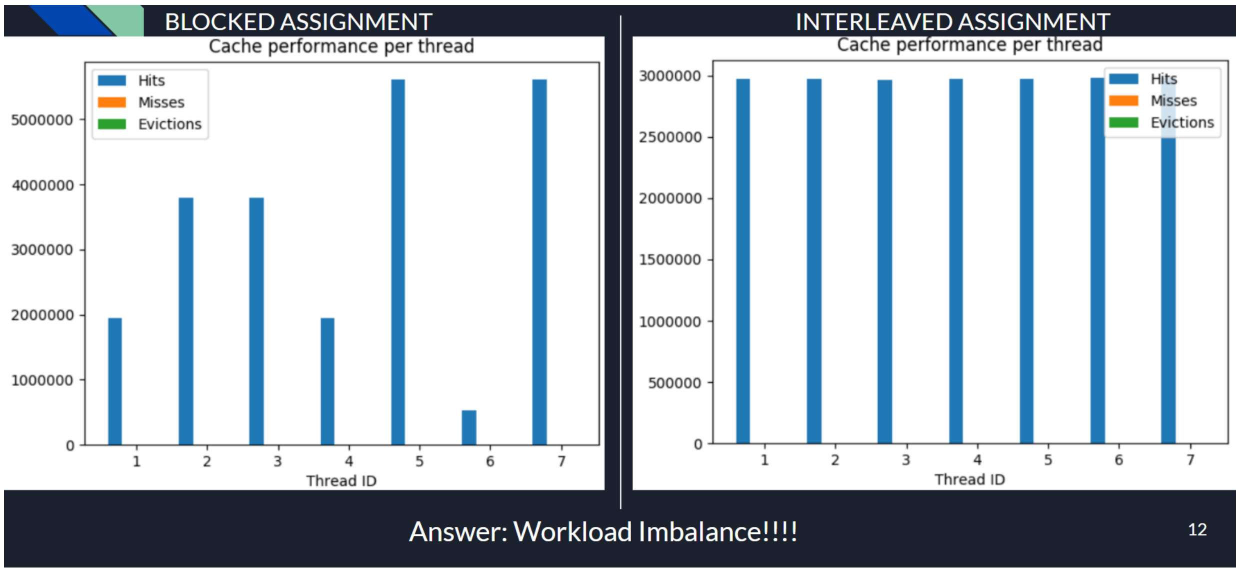
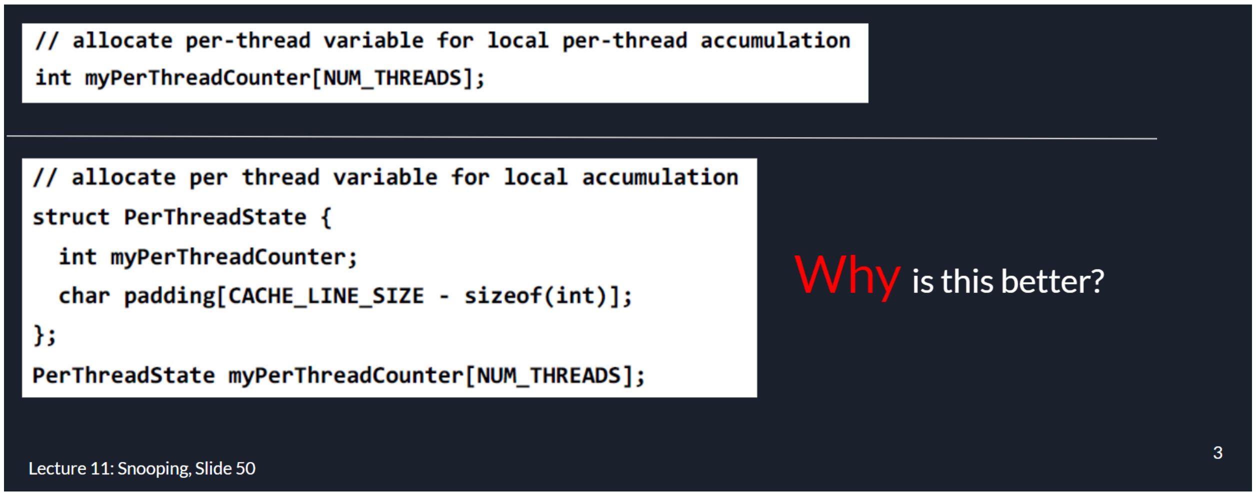


Figure 5: 8 threads, MSI protocol

- Artifactual communication via False sharing (Lecture 11)
    - In Figure 3, we have two code snippets. In the first code snippet, there is no padding added and in the second snippet padding is added. When run over a large number of iterations, the perThreadCounter in the second snippet performs way better than the first code snippet. Why?
    - This is because, in the second code snippet the padding ensures that each thread's counter is in a separate cache line. This is clearly visible from Figure 4, where there is a high number of invalidations and busRdX in the case of the first snippet compared to the second snippet.



```
// allocate per-thread variable for local per-thread accumulation
int myPerThreadCounter[NUM_THREADS];
```

```
// allocate per thread variable for local accumulation
struct PerThreadState {
  int myPerThreadCounter;
  char padding[CACHE_LINE_SIZE - sizeof(int)];
};
PerThreadState myPerThreadCounter[NUM_THREADS];
```

Why is this better?

Lecture 11: Snooping, Slide 50

3

Figure 6

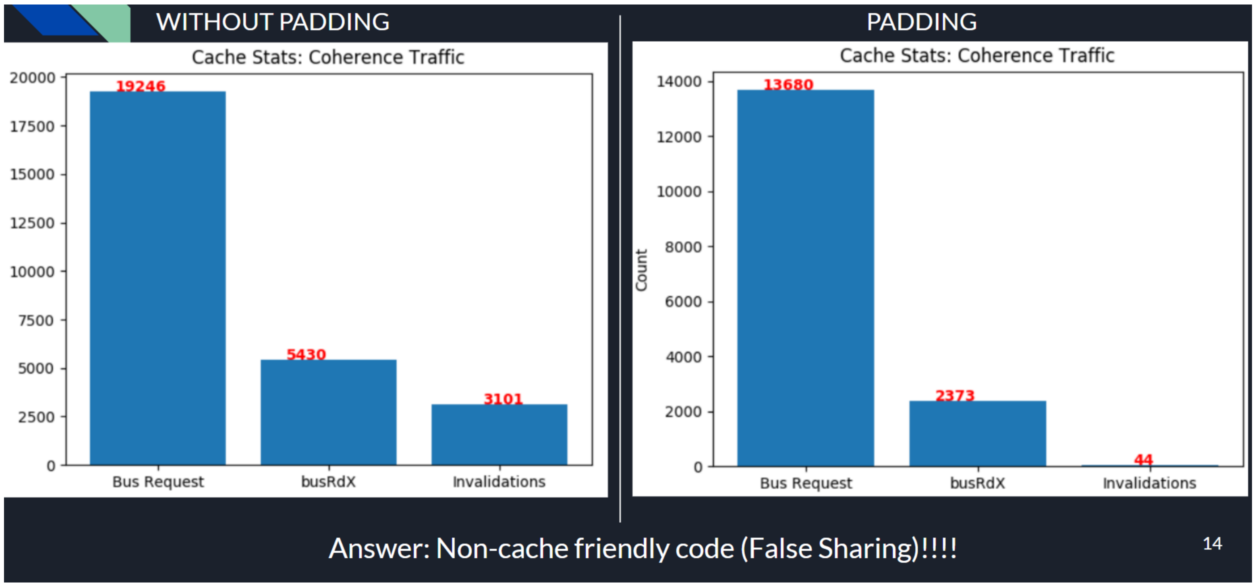
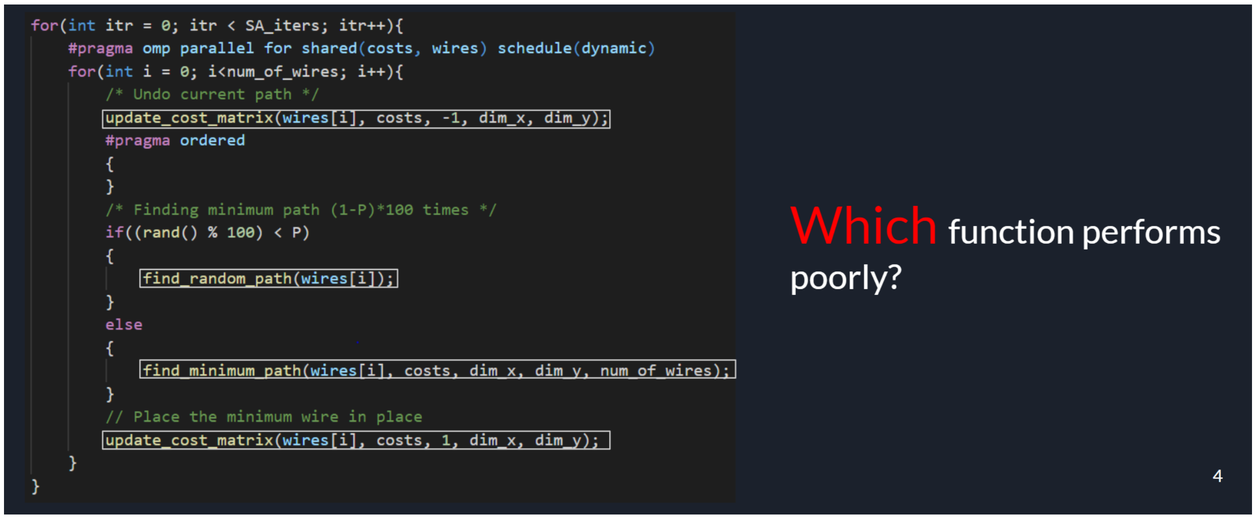


Figure 7: 4 threads, MSI protocol

- **Assignment 3: VLSI wire routing**
    - In Assignment 3, we had to perform simulated annealing for a set of wires and grid size. Our final code's main function looked something like Figure 5. One of the challenges we faced initially was to identify which function wasn't cache friendly.
    - During Assignment 3, we used the perf tool to answer this question. We found that find_minimum_path was this function. We significantly improved the miss rate for this function during the assignment. To validate this, we tested this code using our simulator. As we can see, from Figure 6, find_minimum_path is where the most number of accesses are taking place and as the code is cache friendly (from Figure 6), we were able to get close to 8x speed-up for 8 threads.

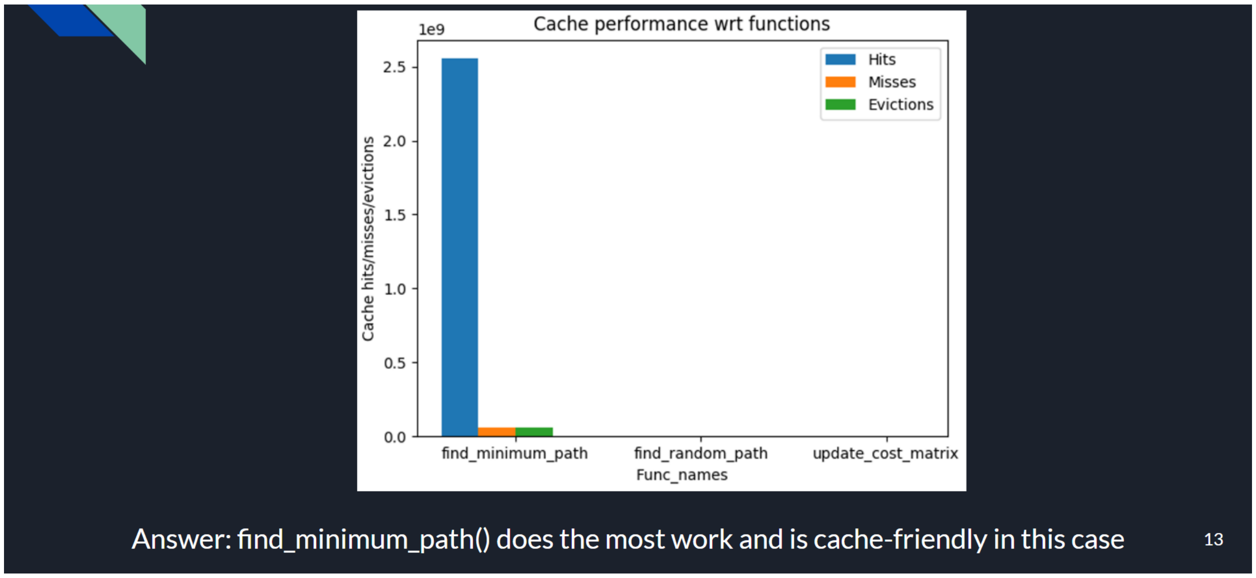


```
for(int itr = 0; itr < SA_iters; itr++){
    #pragma omp parallel for shared(costs, wires) schedule(dynamic)
    for(int i = 0; i<num_of_wires; i++){
        /* Undo current path */
        update_cost_matrix(wires[i], costs, -1, dim_x, dim_y);
        #pragma ordered
        {
        }
        /* Finding minimum path (1-P)*100 times */
        if((rand() % 100) < P)
        {
            find_random_path(wires[i]);
        }
        else
        {
            find_minimum_path(wires[i], costs, dim_x, dim_y, num_of_wires);
        }
        // Place the minimum wire in place
        update_cost_matrix(wires[i], costs, 1, dim_x, dim_y);
    }
}
```

Which function performs poorly?

4

Figure 8



Answer: find_minimum_path() does the most work and is cache-friendly in this case

13

Figure 9: 8 threads, MESI protocol

- Which protocol performs better for a given parallel program?
    - We ran the Assignment - 1: Mandelbrot code (View 1 on 8 threads) for testing this functionality. As we can see from Figure 7, MSI and MESI perform similarly for the Mandelbrot (View 1) code and is definitely better than MI in terms of coherence traffic. MI causes a large number of Bus

Requests and buRdX. As there is minimal sharing in this case, MI gets away in the case of invalidations.

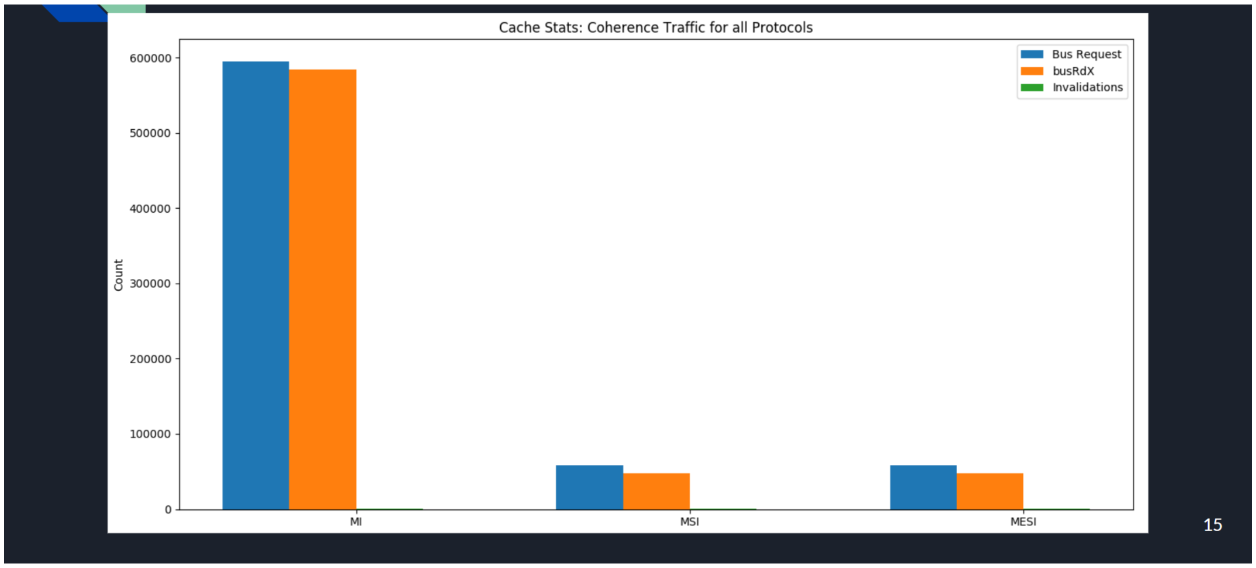


Figure 10: 8 threads

7. REFERENCES:

- Pin: Pin 3.22 User Guide (intel.com)
- Ramulator - A Processing in Memory Simulation Framework
- Fall 2022, 15618: Parallel Computer Architecture and Programming, Lecture 10, 11 and 14, Assignment 1 and 3
- Spring 2021, 15213: Introduction to Computer Systems, Cache Lab

8. WORK DISTRIBUTION:

- The entire design was brainstormed together.
- We equally divided the coding tasks and had shared debugging sessions along with peer reviews

- Slide deck and Report were done together

The total credit for this project should be distributed equally. We did more than we initially planned. (more than 125% deliverables)

9. ACKNOWLEDGEMENT:

We would like to thank Prof. Nathan Beckmann and Prof. Brian Railing for their valuable inputs. We would also like to thank Rui Chen for motivating us and pushing us to go the extra mile on the project.

10. APPENDIX:

- More graphs for MandelBrot problem produced by our simulator

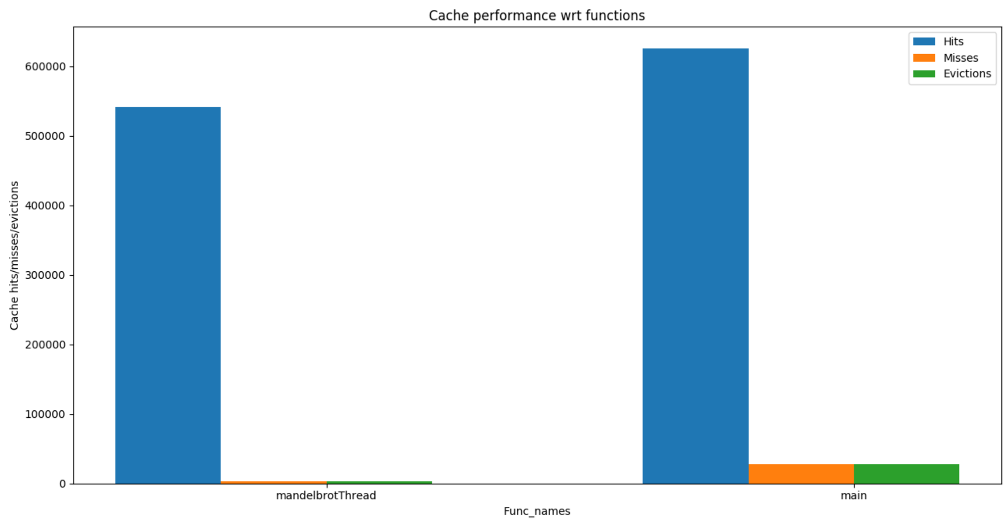


Figure 11: Function specific analysis (Blocked, MESI, 8 threads)

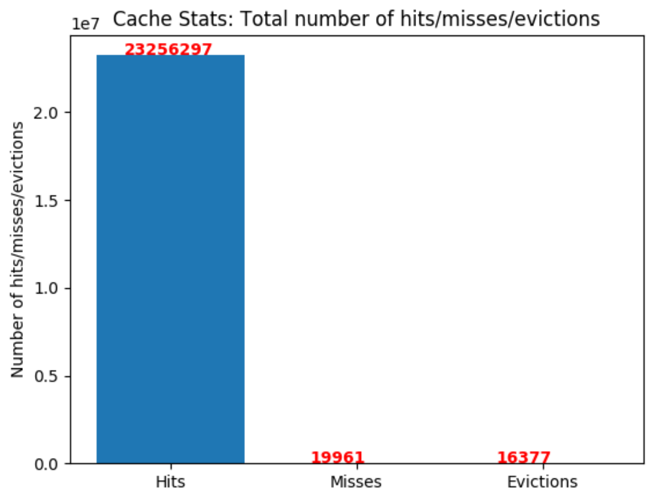


Figure 12: Total hits/misses/evictions (Blocked, MESI, 8 threads)

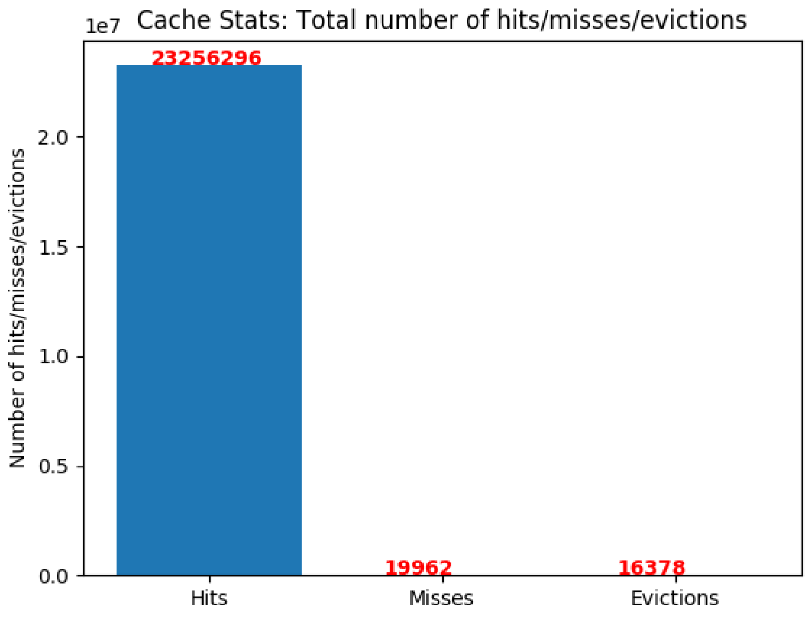


Figure 13: Coherence traffic (Blocked, MESI, 8 threads)

- More graphs for VLSI wire routing problem produced by our simulator

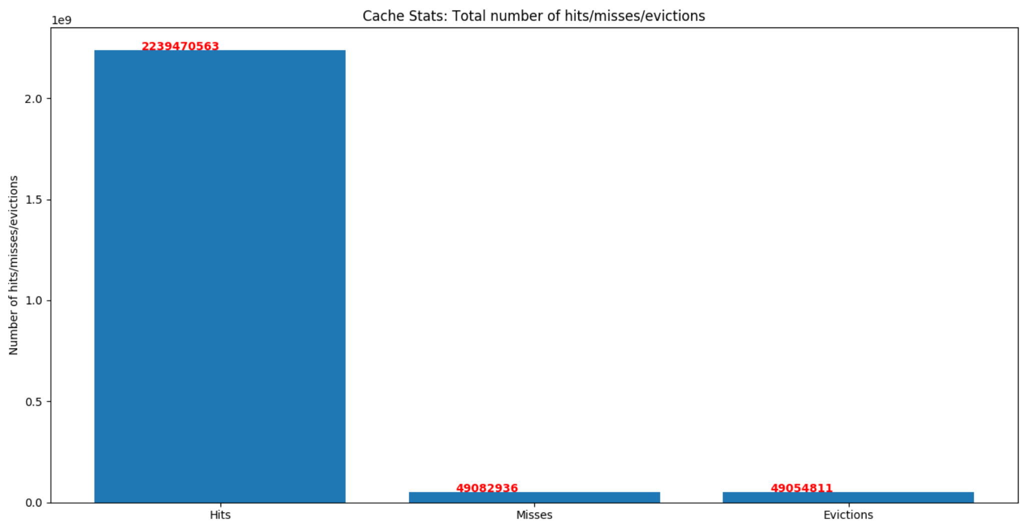


Figure 14: Total hits/misses/evictions (easy_4096, MESI, 8 threads)

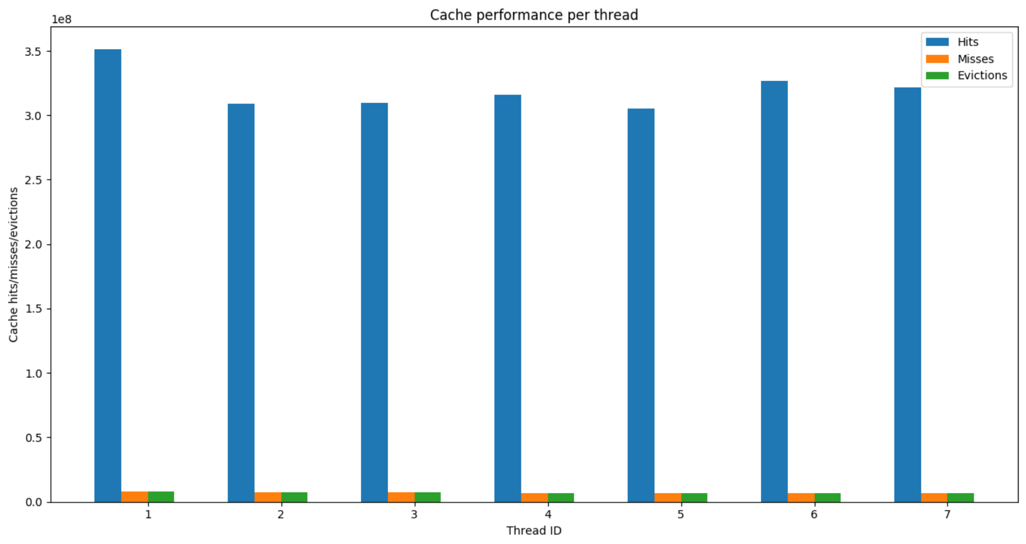


Figure 15: Per thread hits/misses/evictions (easy_4096, MESI, 8 threads)