Cache Simulator
In a Web Application Context

https://cache-simulator-jam.herokuapp.com/

Final Project, Computer Systems 5600, Summer 2018

Jenny Johns, **A**rnold Esguerra, **M**ichael Hanna (Team Space **JAM**)

Brief introduction

Students learning computer and operating systems fundamentals may have a difficult time understanding core concepts such as accessing memory, how a cache works, and the idea of multi-threading. This project is a web application that attempts to visualize some aspects of topics learned in a classroom setting while creating a tool that is fun and interactive with multiple people.

Each user can access a 'thread' and is given a 'execution path' they are responsible for completing. The numbers in memory must be accessed in order. As players interact with the memory board, the respective cache lines they interact with are dynamically loaded into the cache. A visual representation of the cache is shown, as well as a history of the most recent ten interactions. Users have the option of choosing between different eviction policies, and can see the effects of such demonstrated both visually in the memory board and in the cache. The eviction and replacement policy is unique to the cache set, which is tied to the memory region. This ensures that multiple people cannot have different eviction policies when accessing the same cache set.

As will be discussed further throughout this paper, many of the concepts we covered throughout the term get demonstrated in real time.

Through making this interactive app, we hope future users are encouraged to explore and make connections of their own into the utility of caches and thinking more in depth about core computer systems concepts.

The project incorporates the following topics:

- memory regions
- data needed to complete a program (in order)
- a cache
- eviction and replacement policies
- a recent history of actions performed on the memory region and its relation to the cache
- threads

The application is located at:

https://cache-simulator-jam.herokuapp.com

Implementation

The project was implemented on Heroku using the MEAN stack (MongoDB, Express, Angular, NodeJS).

The data (memory region, cache, history) is refreshed every one second.

The main (and only) angular component is the board.component.ts. The html file contains the markup and some logic for what is visualized. It utilizes three services: board, cacheset, and process. These three client services make http requests to the server, which in turn makes calls to the mongo database.

Accessing Memory Regions

Entering memory of region to access will send a request up the pipeline (to client service, server, and mongo database) looking for that specific region entered. If none is found, one is created and ultimately initiating the entire simulation. If two users enter the same memory region, they will see what is being accessed dynamically. Memory regions are ultimately identified by the string entered and can be any alphanumeric. By entering the same string, multiple users can access the same memory region.

Thread & Data Needed To Complete Program Execution

A thread id (which determines the data needed for program execution) is randomly initiated, but can be manually entered to retrieve the same set of instructions of data needed to be accessed.

This list of ten numbers is initiated randomly, but the same list (and how much of it has been executed as indicated by a red highlight) can be retrieved by entering the thread id in the available text box. This simulates how a process can pause its execution, the cpu can context switch to another, back, and ultimately pick up where it left off in the process.

Cache

When a piece of memory is accessed (by clicking a cell), four numbers within the same cache line are brought into the cache. What is currently in cache is represented by the white background, with the text easily visible, and also visualized in the separate section labeled as such. Behind the scenes, the data is copied over from the board, into a cache set. Every time data is accessed, if the cache set is full, it uses the specific policy set. If someone else is accessing the same memory, the cache will be refreshed on your screen every second.

There is also a history (full history available) for the cacheset if it was a miss, hit, and/or evicted. The most recent ten are shown to the user.

Achievements

We created a multiplayer online caching simulator, which allows players to act as threads attempting to complete a given 'program execution path.' This program's main application is educational, but can also be thought of as a game where players compete to finish their 'programs.'

As players compete to complete their programs, factors such as principles of locality, the usefulness of caching, the impact of differing replacement policies. become more pronounced. The hope for this program is that it could be used as a tool for future students to become more comfortable with these concepts.

Results

Although no formal testing was conducted, this program did provide an avenue to collect experiential evidence into factors relating to program execution speed. Such factors are explored more in depth in the discussion section.

Discussion

What are your conclusions and lessons learned? Not everything may have worked, but what is the key takeway from your project?

When piloting the program, it was almost uncanny how easy some of the concepts we learned in class were clearly evident factors in the completion times of the (simulated) program.

The biggest takeaway was the notion that

caching increases the program execution speed (i.e. a user's ability to find and click the right number). In a few instances, even with randomly generated program execution paths, one of the team members/threads demonstrably defeated the others in completing his given execution path first as most of his given program execution path were in the cache lines revealed by the other members/threads. Instead of having to search main memory to reveal it, this person was able to simply access it immediately.

Another key takeaway from this project was reinforcing how programs with good locality are better optimized for performance than those with low locality. When an execution path that had high locality was generated, it was much easier to complete the program. One did not have to spend time searching main memory for the required data. Conversely, with execution paths whose required data were sporadically spread across the board, the length of time it took to complete the process was consequently prolonged.

We had implemented two replacement policies for our cache simulator, 'Least Recently Used (LRU)' and a policy which randomly evicted a cache line when there was a conflict miss.

Although it was not readily apparent which was better for user performance of completing the list of ten numbers, it did shed light into how differing replacement policies could impact any given program.

Our design and structure of using a database allowed for the almost instant implementation of the idea of representing multi-threading. By ensuring that the memory region is unique and retrieving/returning the data based on that allowed for different users to access it concurrently. Setting up the server api to listen for get and post requests enforced scheduling as first-come-first-serve for updating the cache and accessing memory.

Conclusion & Future work

Given more time, we would iterate upon the groundwork we established both in establishing this simulator as an educational tool and as a game.

Educational

In terms of expanding upon the educational utility of this program, there are several areas for expansion. To list a few, one idea would be to implement preprogrammed execution paths (perhaps with example code) that demonstrated varying levels of locality. Half of the class could be given a program with low locality, the other with high. The difference in performance would expectedly be highly self-evident. Another idea which would have been interesting to see, would be to implement a lock/unlock mechanism where only one thread could access a data point at a time. We could also expand upon the amount of policies we've implemented and create differing caches (e.g. number of sets, differing degrees n-way associative caching, etc.).

Gaming

From a gaming perspective, it would have been interesting to randomize the board. When a user clicked on any given tile, the numbers needed would no longer be adjacent (perhaps this would better visualize virtual memory). Per the feedback we received in our proposal, we focused more so on the educational aspect demonstrating what was occurring within the cache. Had we not, it would have been interesting to implement a scoring mechanic based on hits, misses, and time to

complete their given pathway.

We also ultimately focused on the user visuals, and deprioritized the back-end data structures and organization. If given the opportunity, we would refactor some of the code and make a more formal and modularized model. It was admittedly an instance where I had wished we coded in C (coding for these data structures in typescript was a unique experience).

Code Organization and Structure

Given more time, and this will most likely be done, we would modularize, abstract, and clean up a lot of the code. More specifically, making different components on the client side, but also on the server side to allow for easy adding of different eviction and replacement policies.

Lessons Learned

Informally speaking, I found myself wishing to code this project in C as there were many instances where it would have been easier to make this program more robust using data structures in C.

References

Add your citations I suggest at looking at how this paper is formatted as an example. https://www.usenix.org/legacy/events/samples/frame.pdf

Expanded upon:

(n.d.). Retrieved from https://stackoverflow.com/questions/14643617/create-table-using-javascript

Appendix

Figure 1. Initially Filling the Cache

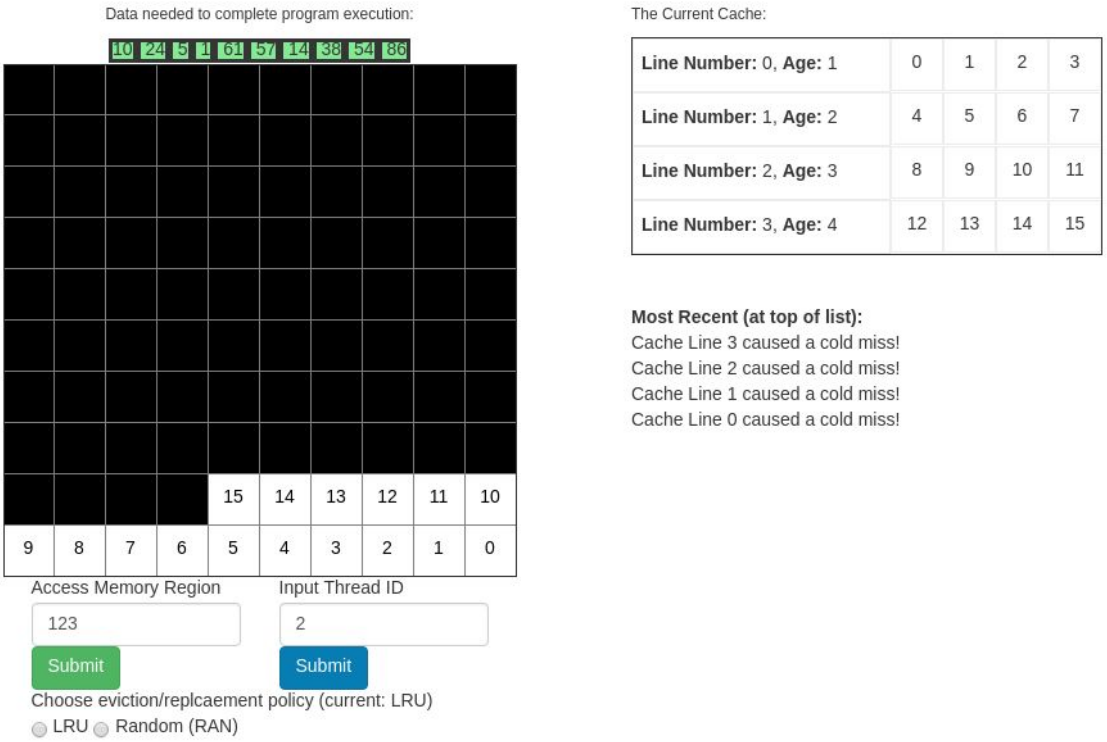

Figure 2. Accessing a number within the cache, but also selecting data needed to complete a program

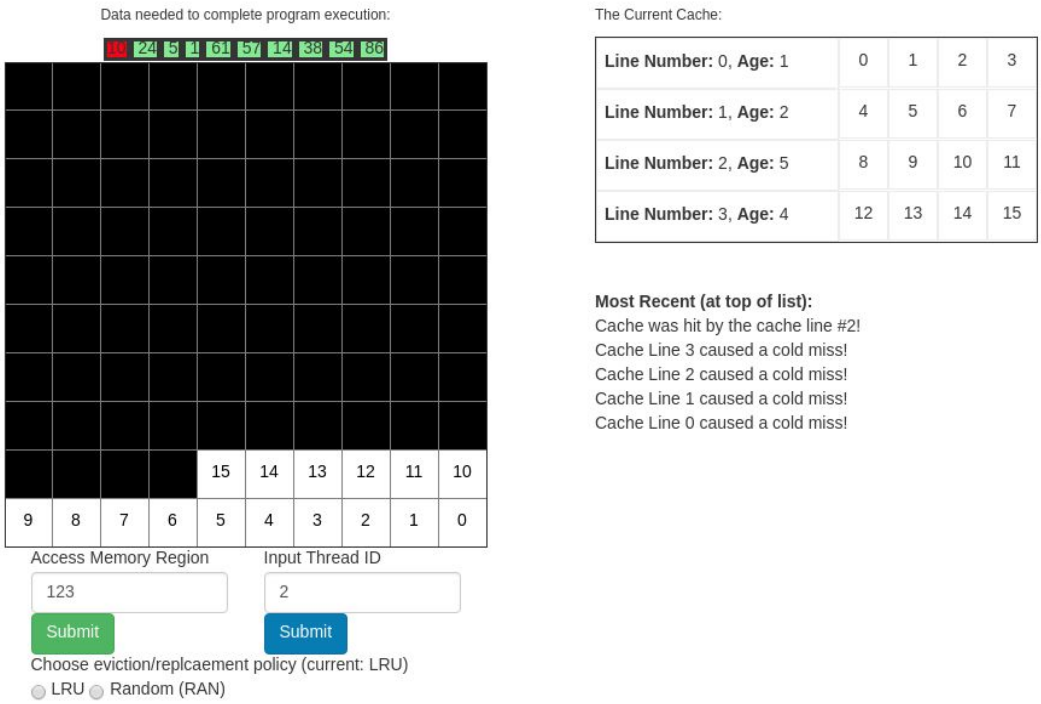

Figure 3. LRU eviction/replacement policy taking effect.

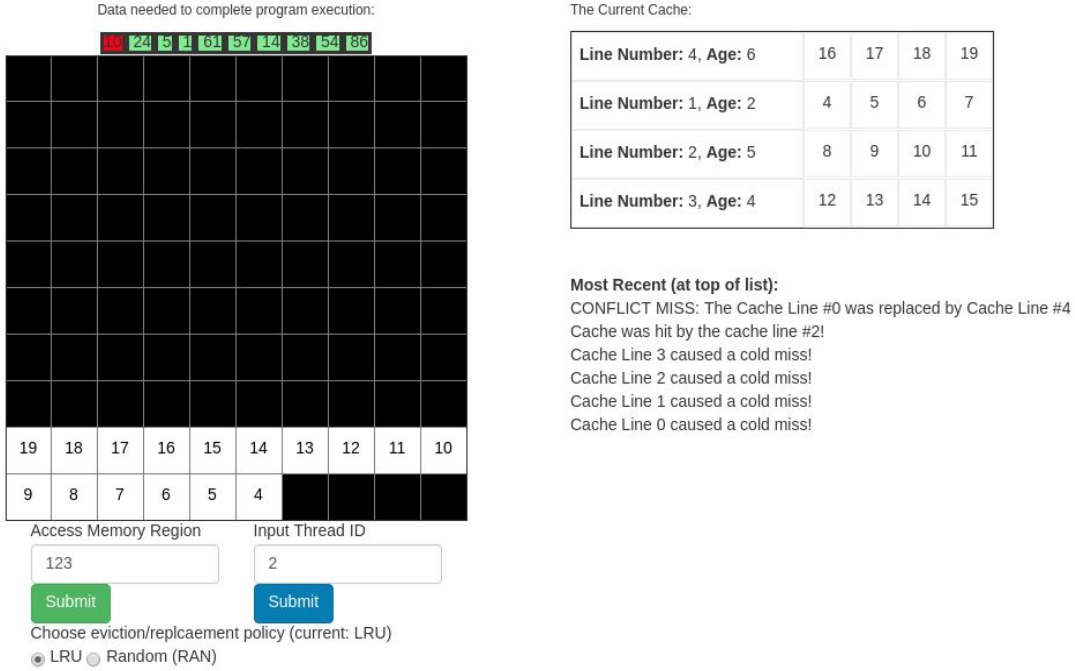

Figure 4. Changed to Random eviction/replacement policy and taking effect.