**Development of a Cache Simulator for Benchmarking and Memory Analysis**

**Abstract**

Efficient memory management is an essential element of computer systems, with an impact on application performance. This paper describes the creation of a cross-platform cache simulator that processes memory traces, benchmarks cache operations, and examines memory access patterns. The simulator includes variable cache settings such as capacity, block size, and associativity, as well as extensive error handling for incorrect input. The simulation results contain precise performance data as well as the option to save cache and memory states to a file.

1. **Introduction**

Memory structure optimization is still an important problem in computer design, notably for lowering latency and increasing throughput. Cache simulators are often used in academia and industry to test cache setups and memory access patterns. This paper presents a unique implementation of a cache simulator that is cross-platform and flexible, supporting Windows, Linux, and MacOS. The key features include:

* Processing and verifying incoming memory traces.
* Simulating cache operations (read and write).
* Creating performance measures, such as execution time and hit/miss rates

1. **Methodology**
2. **Design Overview**

The cache simulator, implemented in C, uses platform-specific libraries to ensure interoperability across operating systems. The main components consist of:

1. **Cross-Platform Compatibility:**
2. Windows-specific implementation (e.g., **gettimeofday()** replacement).
3. Linux and macOS support for **gettimeofday()** and related libraries.
4. **Parameterization:**
5. The command-line options specify cache capacity, block size, and associativity, allowing for flexibility between setups.
6. **Error Handling:**
7. Complete validation of input flags and trace files.
8. Detection and reporting of incorrect input lines in memory trace files
9. **Input Parsing and Trace Processing**

The simulator works through a memory trace file line by line. Each line denotes either a cache read or write activity. The trace processing pipeline contains:

1. **Validation:**
2. Ensure that trace lines include valid hexadecimal characters.
3. Identifying faulty addresses and instructions.
4. **Execution:**
5. Cache read: Querying the cache for the provided address and updating statistics
6. Cache write: This involves writing data to the cache and updating the main memory
7. **Performance Measurement:**
8. The simulation was benchmarked with the use of timestamps obtained through **gettimeofday().**
9. **Simulation Flow**

1. **Initialization**

**-** Both the cache and the man memory structures are allocated memory

**-** Configuring the cache according to input settings

**2. Trace Execution**

**-** Analyzing every command and carrying out the necessary cache actions

**3. Post-Processing**

**-** The cache’s final state is written to main memory.

**-** Producing optional output files and performance metrics

1. **Cross-Platform Challenges and Solutions**

* **Windows**
* Due to the lack of gettimeofday(), FILETIME and conversion to UNIX timestamps had to be used in a customized implementation.
* **Memory Management**
* Buffers are dynamically allocated and resized (Windows implementation of getline()).

**3. Experience in Developing the code**

**3.1. How did you structure the code and why?**

Since we used C to develop our cache simulator, we were slightly restricted in how we structured our code. Whereas, in Java, you have object-oriented programming and can utilize classes and custom objects to better organize and develop, in C we had to utilize structs heavily in order to have a medium to store the state of memory and the cache. There is a memory block struct which is initialized as a pointer array of X size where X is (memory capacity / memory block size). There is a cache struct which contains information on the capacity, block size, associativity, and a pointer array which is initialized to the number of total cache lines. There is a struct to handle all of the cache statistics as well. When our program is run, the main function must initialize all of these structs and pass them with a pointer reference to any functions so that they can change the actual contents of the cache, memory, and statistics. We split our functions logically based on whether they were used by both the “read” and the “write” cache functionality. If something was used by both, it received its own function. If it was used by one and not the other, it was implemented into one of the other functions on the pipeline for that process. Overall, we utilized a single main file which handled the logic for the program parameters and reading trace files and another IO library which contained all of the logic for the cache and memory themselves. That made sense as a logical split for us.

**3.2 What were the challenges in simulating a cache and memory?**

The majority of the problems that we faced with simulating a cache and memory came upon us because of using C in general. We wanted to make our simulator capable of being run on any OS, so we had to implement logic for handling compilation in different environments which caused a lot of pain on our part. There are just general linking issues with different OS’ as well that we had to implement solutions for, like having to link the math library in CMake on Linux but not MacOS, or the getline() and gettimeofday() functions not being available on Windows so having to implement those functions directly. One of the biggest challenges we faced with the cache and memory simulation itself was address partitioning and developing a method to determine the lengths of the tag, set, and offset then retrieving them. We decide to go with determining the length of these addresses based on set-associativity from our book then bit-shifting different sizes to retrieve the actual information with less hassle. We could have also created bitmaps based on the lengths at the very beginning of the program, but we only had that thought after already developing the sufficient method of bit-shifting. Although, the bitmap method would provide better application performance, so it is something to think about as an enhancement. There were some smaller problems with pointer references in functions, but that is a C-specific issue and we figured out that, as long as you have a pointer you don’t need to reference a pointer to that pointer in the function parameters in order to allow a function to make permanent changes to that pointer.

**3.3 Describe how you partitioned the 32-bit addresses into tag, index, and offset, when given a particular configuration**

We used information from the book to determine the set, word, and tag lengths of the addressing scheme first. Set length = log2(number of cache sets), word length = log2(cache block size / 4), tag AND set length = 32 – word length, tag length = tag and set length – set length. After determining the length, we bit shift the address to determine the values like this: tag = address >> (32 – tag length), set = (set << tag length) >> (tag length + word length), word = (address << tag and set length) >> tag and set length. This method worked a lot of the time, but due to how C handles bit-shifting a 32-bit integer by 32 bits, we had to implement a check when setting the word data to see if it is 0 length and if so, set word to 0. This is because shifting a 32-bit integer by 32 bits in C just returns the initial value that you started with for some weird reason. C is a strange language.

**4. Rather than running all possible combinations of cache parameters, you will develop a plan that tests a partial set of combinations. Describe your plan and justify it.**

**5. Results and Discussion**

**5.1. Cache Simulation Performance**

A memory trace file of any size is processed by the simulator, and important metrics include:

* Instructions handled in total.
* Hit and miss rates for caches.
* Time of execution in milliseconds.

**5.2 Provide a table of the statistics you collected from each of the five best configurations in rank order (best first) and specify the criteria for choosing the best.**

**5.2. Error Handling Robustness**

The simulator showed strong edge case handling, including:

* Inexistent input files
* Erroneous trace lines that include incorrect addresses or instructions.

**5.3. Cross-Platform Functionality**

The design’s portability was confirmed when the implementation worked well on Linux, macOS, and Windows.

**6. Discuss factors we did not consider in these experiments that architects must consider before selecting a specific cache configuration, and how these factors would influence the selection. You should consider potential cache access time (i.e., hit time), the die area that the cache might occupy, etc.**

**7. Conclusion**

The program created a cross-platform cache simulator that is versatile, fast, and reliable for examining memory trace data. The simulator’s architecture promotes compatibility, parameterization, and error handling, making it an important tool for memory hierarchy research and education. Future development will focus on expanding the simulator to include multi-level cache designs and adding performance statistic visualization tools.