MemViz: A Memory and Cache Performance Visualizer

Glenn Smith, Max Thomas

Motivation

Good cache performance and memory access characteristics are vital to writing a high-performance program, yet there has never been an easy way to analyze and evaluate these properties. Current tools tend to simply present cache performance data as a raw statistic, which may be enough to indicate a problem but does nothing to contributing to the solution. We decided to create a tool that permits users to create a trace of all the memory accesses made by an arbitrary program, and then visualize said traces as a more easily interpretable, configurable bitmap. The hope with this project was to provide a profiling tool that identifies the location of the problem, in addition to the existence of the problem.

Related Work

Memory and cache visualization tools exist already, but none that we found performed both functions on a process-level scale. Inoue et al. [2] presented a dot plot visualization of memory on macOS systems and used it to compare various methods of forensic analysis to see if they accurately recorded system memory. They ranked a memory imager on four criteria: completeness, correctness, speed, and interference. Their tool visualized the differences between memory and forensic snapshots in a dotplot matrix, comparing similarity of bytes and coloring where errors exist. This formed a roughly diagonal matrix of enormous size, so they sampled it down into a 2048x2048 image. They also generated dotplots comparing live memory with hibernation files, discovering many cases of caching files in memory while copying, using their imaging tool to show that hibernation files are not good representations of system memory and should not be used for forensic snapshots.

For visualizing cache accesses over time in small programs, Choudhury et al. [3] presented a method using concentric circles. Their tool created an animation of memory accesses, mapping main memory and the L2 and L1 caches into concentric circles, with the processor represented as a dot in the center. Memory accesses are then visualized as propagating out from the processor through the layers of the cache. Their tool, however, was mostly tested on trivial programs, simulating an unrealistically small cache. This is because it was designed for educational purposes, to teach students about what kinds of program behaviors cause bad cache performance, as opposed to being used as a cache performance profiling tool.

On a much larger scale, Bosch et al. [4] used a complete machine simulator to track when memory accesses used various caches in parallel applications and visualized the results using the visualization tool Rivet. They focused on multiprocessor systems using shared memory, running a case study of the Argus rendering system, a parallel graphics rendering library optimized for shared memory multiprocessors, with the SimOS machine simulator. Using the Rivet visualization

environment they created MView, a tool for displaying application memory behavior, and PView, a tool for displaying scheduling, exception handling, and other process attributes. They used these tools on the Argus renderer and were able to discover unexpected idle times in scheduling and investigate parallel performance.

Our analysis tool is based off the work done by the DynamoRIO team [1] and their sample code for DrCacheSim. DynamoRIO is a library that permits the creation of program analysis tools that function by inserting arbitrary instrumentation into processes at runtime. This permits recording and analysis of pretty much anything a program does, from memory accesses to instruction use. DynamoRIO ships with, among other things, a tool called DrCacheSim, which is designed to simulate the cache performance of a program at runtime. While their tool analyzes the same scope of cache accesses as ours does, and for roughly the same purpose, it is entirely terminal-based and only produces broad statistics for cache performance across the whole program. We expanded it to record specific address data that we could import into a GUI, allowing users to examine cache performance at an arbitrary resolution.

Design & Evolution

Memory access & Cache Performance Tracker

From the beginning, the performance tracker was intended to use the DynamoRIO library [1] to insert and manage the instrumentation needed to record memory access and cache data. The initial plan was to write a DynamoRIO plugin from scratch, which was quickly abandoned due to the high level of difficulty of writing plugins. Instead, the first iteration of the memory tracker was created by modifying the memtrace_x86 plugin that came packaged with DynamoRIO. Using this, we were able to create initial test traces of memory accesses.

This proved insufficient for our purposes, since it was incapable of recording cache access data and ignored memory accesses in the instruction segment. To that end, we switched to using a modified version of the DrCacheSim tool, leaving the cache simulator and framework intact but completely rewriting the statistics gathering portions to record and then write out a memory and cache trace. This approach proved sufficient to gather the recorded data, although it did necessitate running the analysis tool through a shell script due to the way DynamoRIO handles file paths.

The file format also went through several iterations. The basic structure (Figure 1), which changed remarkably little, was composed of several sections. Each section was composed of an

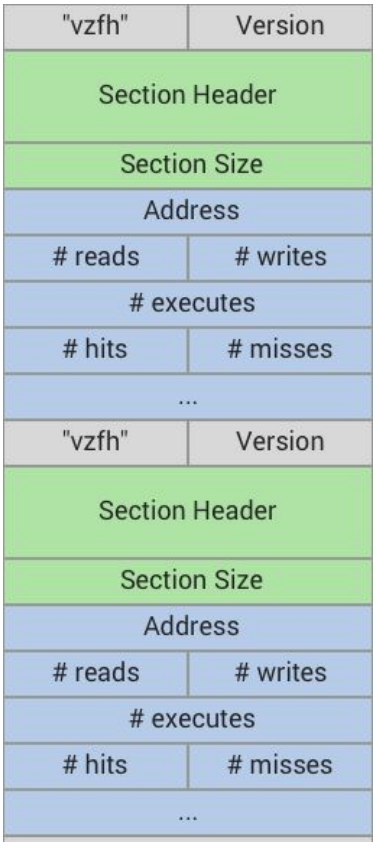

Figure 1: Final file format diagram

initial 16 or 32-byte section header, followed by some number of subsections, each composed of a 16-byte subsection header and some number of 16 or 32-byte blocks, each containing data about some address. As the tool evolved, the data collected (as well as the manner in which that data was collected) changed, and the file format changed along with it. The final version has 2 sections. One of them contains data collected from traces of the L1 process 0 data cache, and the other contains data collected from the L1 instruction cache.

The final tool takes as arguments an absolute (from root) or relative path to a binary, and a relative path to an output file. It then runs our modified tool (named DrCacheTrace) on the provided binary, recording the output to the requested file. The user can then store the trace file for later use, or open it in the trace file visualizer.

Trace File Visualizer

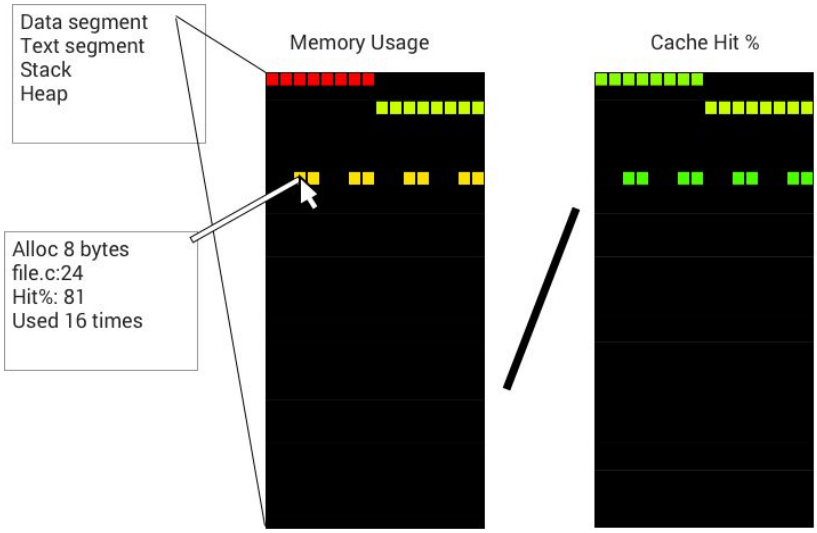

Figure 2: Initial mockup of the visualization

The initial design (Figure 2) planned to show memory in many lines of a few bytes each, providing information on any byte the user hovered over. We intended to show allocation size, cache hit rate, usage counts, and potentially source file information (if available). The pixels would use hue to represent how often the bytes were accessed, having red pixels represent least used and green pixels represent most used. We planned to include the option to use the cache hit rate for the hue value instead. Additionally, there was going to be a menu with a list of memory segments, from which the user could select one to show. The memory regions from other segments would be hidden, allowing users to focus more closely on the category of data they were investigating.

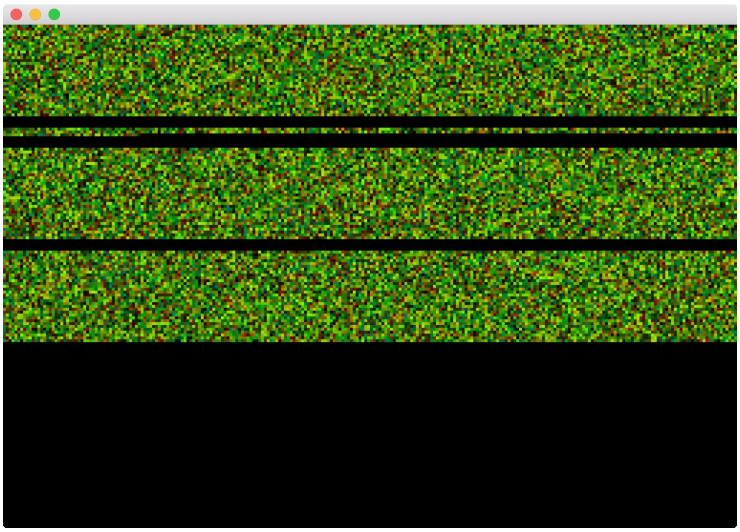

Figure 3: openFrameworks-based renderer of random test data

Initial mockups of the UI design in openFrameworks (Figure 3) indicated that our previous plan had wildly underestimated the size of the data. Even for reasonable amounts of memory the window was filled almost completely at any size greater than a few pixels per byte. Additionally, while openFrameworks was useful for quick testing and writing basic drawing algorithms in C++, we quickly realized that it does not have the mature GUI capabilities that we needed. Because of this, we migrated the code to use the Qt framework instead. Even though Qt required some strange setup and a total rewrite, the visual GUI designer and extensive library support was a very acceptable trade to make. Qt also allowed us to make scrolling regions, so we could work around the large data sizes.

Over time we realized that many of our planned features were out of the scope of the project, due either to the difficulty of collecting the information or the difficulty of storing the information in a usably compact, meaningful format. We underestimated the amount of data that would be produced even by trivial programs. The smallest programs we tested produced a trace file that was over 1 MB, and the largest program we tested (curl) produced a file that was over 700 MB. The difficulty lies in how much information needs to be stored. With our final format storing 32 bytes per address, we had to limit our scope to prevent files from becoming even larger.

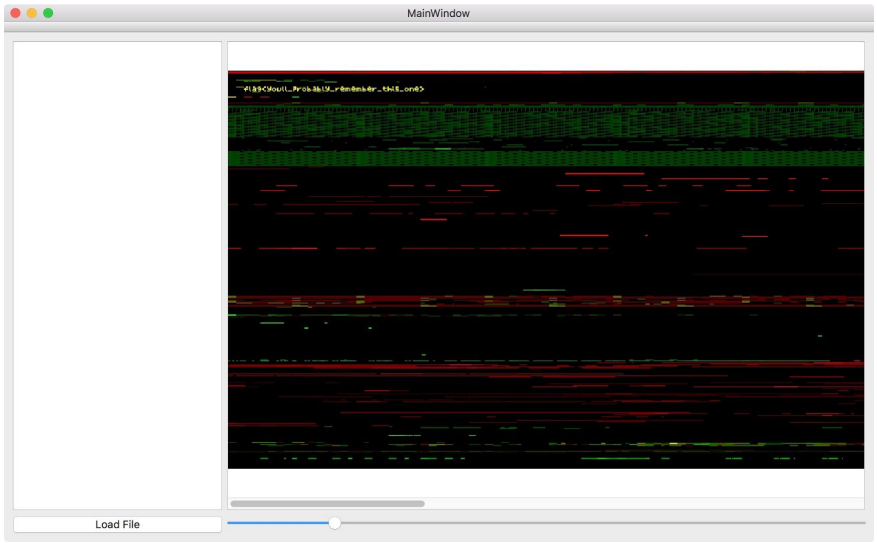

Figure 4: Early Qt-based renderer

Early revisions of the Qt-based interface (Figure 4) included a scroll area to house the large visualization, a zoom slider to allow users to scale the image for a closer look. As the design progressed, we added the ability for users to change which variables were represented on the visual axes of hue and brightness. The final set of choices includes Read to Access Ratio, Write to Access Ratio, Execute to Access Ratio, Hit to Access Ratio, Access count relative to the most accessed byte, and constant value. The final tool that we created (Figure 5) also includes a list of pages that users can choose to hide, allowing them to focus more closely on the data that matters. We added an information panel that gives statistics for the byte the user is currently hovered over. Currently, the statistics shown include read/write/execute counts as well as cache hit and miss counts.

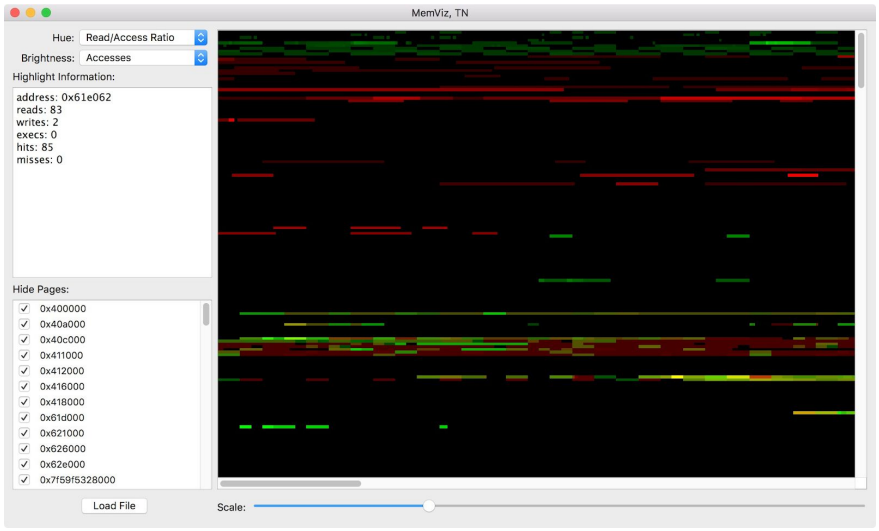

Figure 5: Final renderer

User Feedback

Questions

The two questions we asked were:

1. Are we collecting too much data? Too little? Any suggestions on other things to look at?
2. The visualization [at the time, see Figure 4] is somewhat unintuitive. How can we make it easier to use?

Answers

The feedback we received from our demonstration was mostly positive, despite most of the class not being part of the intended audience for the tool. Most people were confused by the tool's purpose and recommended improvements to the UI, which at the time was lacking most of the options the final version has. Some people recommended better inspection of the information, as we had not yet implemented that panel. Multiple people suggested having the tool visualize data over time and be temporally filterable. While this would be useful and interesting, it is outside the scope of our project. The recommendations that were given for the analysis tool were either redundant with the suggestions given for the UI, or too difficult to implement within the remaining available time. Several of them have ended up the future work section.

Formal User Study

Description

To formally run a user study for this, we would find a group of experts in the systems-level programming area. Due to the niche appeal of this tool, a test with a general audience would not be feasible, as they would not have the required background knowledge to understand the data visualized by the tool. We would attempt to find as many people as possible, regardless of age or other factors, as it may be difficult to locate sufficiently many qualified individuals. A decent sample size would be at least 20 people. We would conduct the experiments ourselves, again due to potential difficulties finding qualified people.

We would be attempting to answer our research question: can people use the tool that we made to find areas of poor cache performance? The hypothesis would assert that yes, our tool does allow them to do this. To test this, we would provide the participants with our tool, a set of test programs, and a set of questions about those test programs. The test programs would specifically include: (1) a basic "hello world" program to help familiarize the participants with the tool, (2) a program designed to miss the cache as often as possible, (3) a larger program with more complicated memory access characteristics.

Before beginning the test, we would describe how our tool works, being sure to mention its ability to display various aspects of memory access, including access type and cache performance, and its ability to show data for individual addresses. Participants would then have 20 minutes to try and answer these questions: (A) which regions of memory were accessed the most, (B) which regions are the stack, the heap, the program binary, and system libraries, (C) where they could improve the performance of the test program. We would ask them to identify regions with good and bad cache performance, which we would be able to compare with expected results. After the participants have completed the tasks, we would ask them if they thought the tool was (A) Effective, and (B) Useful.

After doing all the studies, we would use the results to test our hypothesis. We would check if a statistically significant number of people were able to correctly identify regions and if a significant number of them considered the tool effective and usable. The intent would be to show whether or not our tool is capable of being used for program memory analysis. For this, we would need data showing that skilled programmers were able to complete successfully the problems we gave them and found the tool effective. As most programmers are generally unaware of memory traces, having positive results would inform them (and us) that this technique both exists and is effective at analyzing programs.

Script

Thank you for coming to our study. Just as a reminder, you are going to be testing a software tool that analyzes and visualizes the memory accesses that programs make. You are free to leave at any time if you do not wish to continue, and you will be compensated for your time. There are no significant health risks associated with this tool and we are not going to collect any personally identifying information. Let us introduce what you will be doing...

Indicate laptop on table running tool.

The tool shows the memory as a bitmap, where each pixel is associated with a memory address that was used by the program. You can change the coloring scheme with the dropdown menus. Additionally, hovering the mouse over a pixel will display specific values for number of reads, number of writes, number of executes, number of cache hits, and number of cache misses in a text field to the left of the bitmap. To focus better on the parts of memory you want to investigate, you can hide segments by unchecking the corresponding item on the segment list. We've got an example loaded already; try playing around with the options for a few minutes to get an understanding of what you can do.

Have a sample file preloaded and let them investigate for maximum 5 minutes or until they are ready to continue.

Now that you know how to use the tool, we'd like you to use it to answer some questions.

Hand sheet of paper with questions.

You'll find the necessary files on the computer's desktop. We'll give you 20 minutes.

Sit back and observe them answering questions. Take notes on where they stop to think, if they make any sounds, or if they ask for help. Don't answer their questions. Also record how long it takes them to do each part, and if they stop to change any responses.

20 minutes later…

Do you think this tool was usable? *Write their response.* Was it useful? *Write response*.
Thank you for your time.

Future Work

If we had more time to refine the tools we created, we would add both more data collection and more interface tools. The additional data includes L1 data cache data for processes other than the main process in multiprocess programs, and higher levels of the cache, such as L2 and L3. Additionally, we would use zlib (or something similar) to compress the file format, which currently has rather low entropy but large size. We would add the ability for users to annotate the visualization, highlighting and selecting points of interest. We would attempt to record which instruction addresses accessed which bytes of memory to allow users to determine which lines of their code were responsible for the data. Also, we would record which segments the various regions of memory were associated with and allow the user to show only the heap, stack, or other regions by name instead of by address.

Contributions

Max worked on the DynamoRIO cache trace tool, and then on the UI once the tool was in a satisfactory state. This involved learning to work with the DynamoRIO library, analyzing and repurposing the DrCacheSim tool, and designing the file format used to store the traces recorded by the tool. Max also helped with bug-hunting for various problems during UI development. Glenn designed the UI and implemented the visualization renderer. He also created the region hiding, file format parsing and analyzing code, and prototyped the visualization in openFrameworks before writing the majority of the Qt renderer.

References

[1] Bruening, Derek L. “Efficient, Transparent, and Comprehensive Runtime Code Manipulation.” Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 2004.

[2] Inoue, Hajime, et al. “Visualization in Testing a Volatile Memory Forensic Tool.” Digital Investigation, vol. 8, 2011, doi:10.1016/j.diin.2011.05.006.

[3] Choudhury, A. N. M. Imroz, and Paul Rosen. “Abstract Visualization of Runtime Memory Behavior.” 2011 6th International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT), 2011, doi:10.1109/vissof.2011.6069452.

[4] Bosch, R., et al. “Performance Analysis and Visualization of Parallel Systems Using SimOS and Rivet: a Case Study.” Proceedings Sixth International Symposium on High-Performance Computer Architecture. HPCA-6 (Cat. No.PR00550), doi:10.1109/hpca.2000.824365.

The motivation and audience for your visualization.
What are the goals you hope to achieve with your visualization?
What was your research question and hypothesis? OK

Summary of related work, data sources, algorithms, and other inspiration or background material. (Use proper bibliographic format in the report.) ok

Visualization Design Evolution. What was your initial idea for the visualization, what your initial plan to present, explore, or discover with the data? How did the design change as you worked on the visualization? Include intermediate screenshots and examples of the visualization process. OK

What feedback did you get on the design from other people? (How well do these people match your target audience?) How did your visualization design or implementation change based on this feedback? How could this visualization continue to evolve beyond this course project? OK

Description of the core features/contributions and technical implementation details/challenges. What visualization packages or other pre-existing code did you leverage? What algorithms or data structures did you implement? OK

For team projects: Who did what? (Note: Both presentation and report writeup should be a joint effort). OK