Visual tools for debugging data structure performance

Coordinator: Kostas Zoumpatianos
Advisor: Stratos Idreos

Background and Motivation

While processing power of household CPUs has increased extremely rapidly in recent years, the speed of access to storage hasn't increased at the same rate. As a result of this, data movement is often the major bottleneck In the world of databases and data intensive systems. Consequently, designing appropriate data structures that minimize data accesses is one of the key concerns of the field.

Validating a new design is relatively straightforward: a well designed benchmark will quickly point to a speed / throughput improvement or lack thereof. On the other hand, it is rather more difficult to determine the reasons behind **why** a certain data structure implementation performs the way it does. An oft-encountered problem is that a design would perform well theoretically, but the real world implementation doesn't live up to that standard. Given the highly complex interactions of the different components of a modern computer architecture e.g. different data and instruction caches and fallback rules between those, prefetching, pipelining, TLB, SIMD or regular parallelism, it can be difficult to guess where exactly the problem is. The inspiration for this project came from a situation where the author was trying to debug why a trie implementation was slow.

Goals

Existing tools provide incidental and non-detailed metrics e.g. from the point of view of the hardware (L1 / L2 / LL cache hit ratios) or the operating system (CPU usage, load averages, thread usage) or the larger system in which the data structure operates (number of operations per second). None of these directly address our core concern: memory accesses to the data structure itself. I would like a mechanism that understands our data structure and provides metrics from the data structure's point of view.

Moreover, I would like a tool that works with real, existing data structure implementations, with minimal modification. Finally, I want a tool that avoids interfering with the performance characteristics of the software being measured.

Related Work

First and foremost, without an implementation, analytical models are great tool to reason about the expected properties of a hypothetical data structure. However, the work of modelling is difficult and thought-intensive. Careful decisions need to be done around which details are modelled and which are to be abstracted over. It is easy to make false assumptions. Also, as the conditions and scenarios become more complex (e.g. taking into account complex query and data distributions), so do the models, to the point where a prototype implementation might be simpler to create.

Given an implementation, there are empirical tools that are often brought up: for example, tracing mechanisms that use hardware performance counters like perf and dtrace, but the metrics aren't very detailed (e.g. number of L1 cache hits/misses). This gives a general idea of what might be happening. Memory / heap profilers like valgrind are generally geared towards monitoring allocation sites, number of allocations, overall memory consumption, and the detection of memory leaks. None of these tools get to the heart of the matter, which is getting detailed records of memory accesses.

For memory accesses specifically, IMem/memTrace is a very interesting prototype tool specifically tailored to trace memory accesses with minimal overhead. This could potentially serve as the tracing component of the debugger, but has not been maintained since. MemDB is a more comprehensive tool with the closest overall goals to our project, but focusing more on profiling memory sharing in multithreaded applications. It is also similarly unmaintained. Both of these tools depend on an old versions of libraries which depend on an old versions of GCC, which makes it a frustrating experience to deal with.

These tools, however, provided the inspiration for using Dynamic Binary Instrumentation frameworks for doing memory access tracing. Dynamic Binary Instrumentation involves analyzing the behavior of a running program by injecting instrumentation code alongside the regular program code. The injected code can be thought of as an event handler that performs whatever action is necessary to gather whatever metric is necessary. However, since this happens at the machine code level, DBI tools are agnostic to the underlying language and frameworks of the code. DBI tools are used in the security industry for analyzing malware. Another example is Valgrind, which is usually thought of as a profiler, but underlying it is a dynamic binary analysis framework, albeit a heavyweight one that comes with a dramatic performance hit. Newer tools focus on reducing the performance overhead and the side-effects of instrumentation. There are two main open-source lightweight DBI frameworks: Intel PIN tools and DynamoRIO which provide similar features. I chose the latter because it was well documented, simple to build across a variety of platforms, and openly and actively developed.

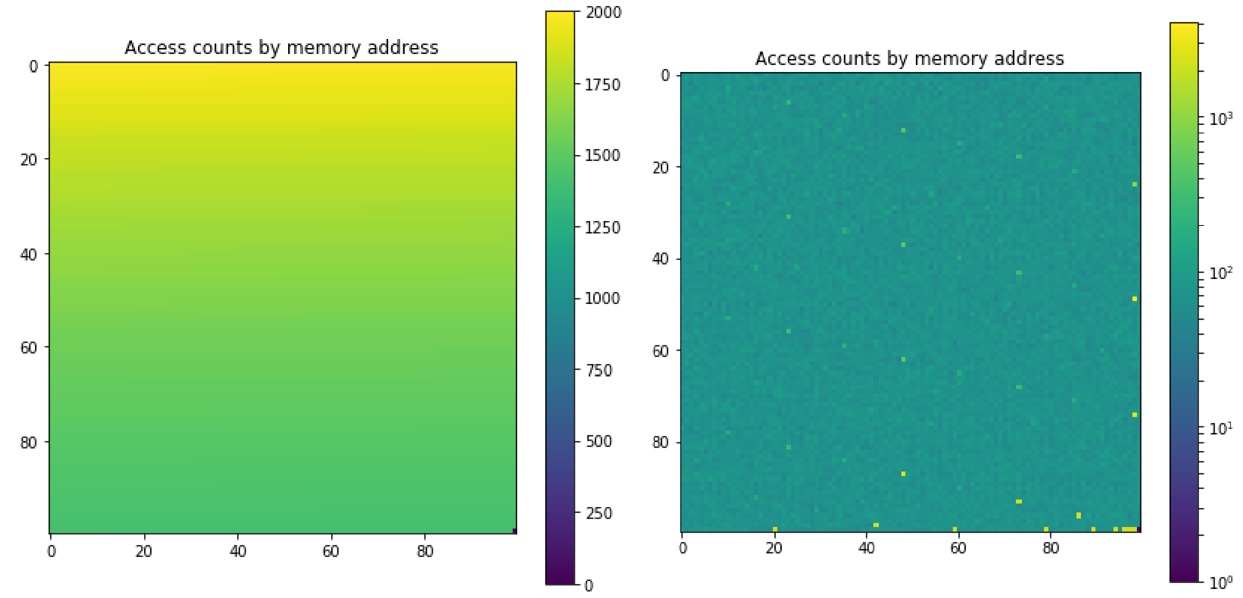
Results

I have built a prototype to test the viability of this idea.

First, the memory trace of a running executable is dumped with DynamoRIO's memtrace_x86_binary instrument. This trace can be thought of as a log of all memory accesses. Each entry contains information on the type of access (read/write), virtual memory address, the size of the access, and other more tangential information. This file can currently reach several gigabytes and hundreds of millions of entries for a program that runs a little over a second.

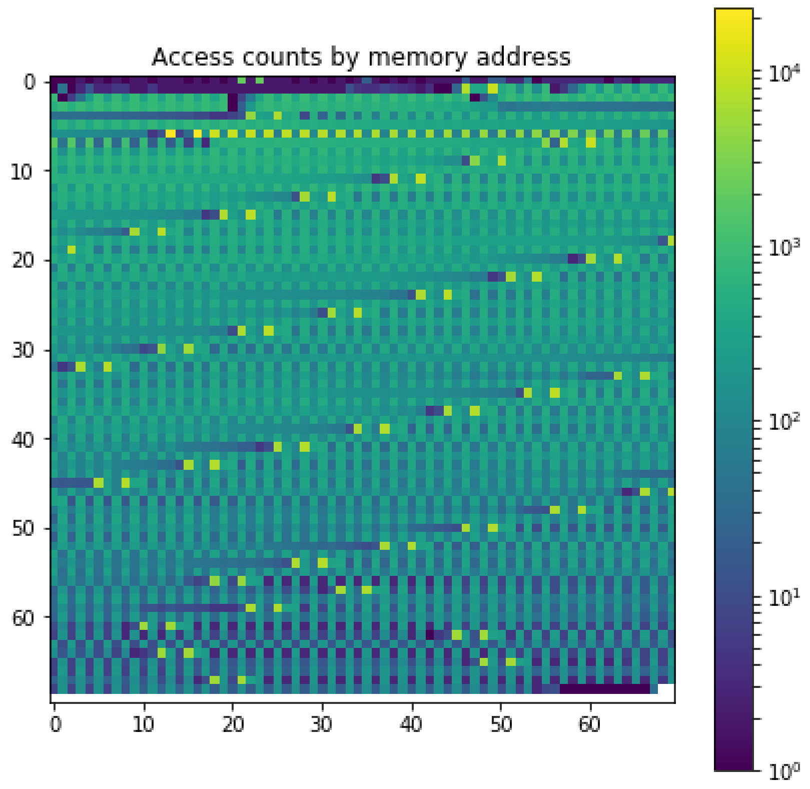
This file is then processed using a python script, parsing the binary format and dropping records from the trace that aren't relevant. This then allows us to create visualizations of our memory space.

One such visualization is a heatmap: ignoring temporal data, I focused on the memory region that contains the data structure, to notice specific access patterns. Here is a heatmap that shows memory accesses in two different data structures:



Memory addresses are shown as one contiguous block that wraps around the edge: the top left is the first address, and the bottom right is the last.

These heatmaps compare two very common methods in databases to search for values: scans and binary searches. The graph on the left shows a series of scans through an unsorted array. The gradient-like nature of the reflects the fact that every search starts scanning at the beginning of the array, but then some searches stop early when they find their result. The graph on the right shows a series of binary searches over a sorted array of the same values: the accesses appear to be concentrated in a small number of evenly distributed points (the midpoints that the binary search uses), but also there are a lot less accesses overall, which are fundamental benefits of a binary search. As one can see, even this simple graph provides important information about the behavior of the data structures without any information a priori. Below is another similar heatmap for a B-tree where one can see the different nodes and the sequential scans within the nodes:



Another option for visualization that we experimented with includes the temporal aspect of the data: we can "play back" the access log, plotting it in regular intervals and changing the color scale as we go:

- Scan animation: https://drive.google.com/open?id=1P3vfutDH8av0ApdjDoanByMnP_nKL7lY
- Binary search animation: https://drive.google.com/file/d/19rbx4eSLfHPfbVPA6YINER2kqpK3dpEJ/view?usp=sharing (You always see the midpoints as highest, and since the accesses are uniformly distributed, as more results land on all the regions of the array the "background" starts to increase all together)

Vision and Future Work

I envision a final tool to be split into three main parts: the tracer, the preprocessor and the visualizer. The tracer is a very lightweight DynamoRIO instrument that dumps relevant memory traces from an executable with as little performance overhead and configuration as possible. The preprocessor is a tool to take the binary trace dumps, cleans them up and converts them into a more easily processable format that's ready to analyze. The visualizer then takes a clean trace and visualizes it in a multitude of informative ways. This can be implemented as a command line tool but also as a library so that it can be used alongside more complex exploratory data analysis tools, e.g. inside a Jupyter notebook.

For the tracer:

- The current tracer worked well but the size of the trace dumps pose a problem - it makes the analysis cumbersome:
    - A sampling approach might work well, however this might interfere with some of the more granular analysis
    - Starting and stopping the trace to only record relevant parts of the program can be implemented through DynamoRIO's annotation framework
    - A compressed tracing format might be helpful - e.g. long sequential writes can be recorded as a single range record instead of thousands of records. We might also simply record offsets rather than full addresses.
- Alternatively, we might be able to reuse the dump format from DynamoRIO's cache simulation framework, which wouldn't cut down on size but would allow the trace to also be run through their cache simulation tool.

The preprocessor:

- Allow filtering the trace down and tagging the trace's relevant regions by a pre-specified set of address ranges, which can be dumped during program execution.
- Improve data structure detection: currently we use a very simple clustering method that looks at minimum distances between accesses to detect relevant-looking address ranges if none are given. We can probably use some better algorithms for this.

For the visualizer:

- Provide more over-time animations
- Allow for visualizing multiple memory regions at the same time
    - For memory regions that have multiple connected regions (e.g. BTrees), allow the user to input a list of connections, so we can draw them like the data structure itself.
- Experiment with the temporal aspect of data accesses:
    - We could generate an access sequence network, which would show how often which region gets accessed after every other region, which might help tease out

    dependencies between memory regions and point to items that should be placed closer together in memory.

- A big open question is how we should handle changing data structure shapes over time - is there a possibility to track this? In a large memory space it's rare that new allocations happen in the same virtual memory address as the old one, so perhaps it's only a matter of marking "removed" regions appropriately. The annotation framework could be helpful in this, or alternately we could track de-allocations with DynamoRIO.