PerVERT: Performance Visualization and Error Remediation Toolkit

Niels Joubert*
Stanford University

Eric Schkufza†
Stanford University

Figure 1: *The PerVERT tool*

Abstract

Performance tuning is an important step in the development large software systems. Examples include web-servers which routinely handle thousands of simultaneous content requests, and petaflop supercomputers which perform physical simulations that span tens of thousands of cpu cores. As improvements in clock frequency slow and hardware trends continue towards increased parallelism, the runtime performance of these and similar systems will become ever more a function of memory efficiency. Unfortunately, the ability to effectively reason about this phenomenon using existing tools such as valgrind [Nethercote and Seward 2007], gprof [Graham et al. 2004], or gdb [Stallman and Pesch 1991], through a text-based interface, is limited, and tedious at best.

We present PerVERT, an instrumentation framework for logging a process's virtual memory traffic and a visualization suite for reasoning about common memory performance bugs: Are memory accesses organized coherently in both spatial and temporal dimensions? To what extent do these patterns differ based on program inputs or changes in source code?

Keywords: performance visualization, JIT, compiler, instrumentation

Links: DL PDF

1 Introduction

Users and developers alike care about the performance of computer programs. It's commonly understood that good performance can often be the distinguishing factor in the usability of a code. This is especially true is the scientific and financial simulation communities, where long-running codes have to share computing power on clusters of machines.

As computers become faster and processor count increase, performance becomes more and more a function of memory efficiency. Memory speeds are not increasing at the same rate as computing power, and the cost of communicating data by touching memory is becoming the primary factor in degrading performance. For this reason, tuning the memory access patterns of a code is the primary way of increasing performance of the same algorithm. This is an important distinguishment - given the same correct code, performance is increased by using smarter datastructures with smarter access patterns to them.

Performance tuning is still considered a "black art" due to the

*e-mail:njoubert@cs.stanford.edu
†e-mail:eschkufz@cs.stanford.edu

opaqueness of performance: while code directly expresses computation, the time it takes for a computation is a second order effect of the code. This is especially true for accessing memory, where the memory hierarchy is completely invisible to the user.

Improving the performance of code means minimizing the time it takes to compute a result. For this reason, people measure performance in terms of events per amount of time. To be able to improve the performance of code, we must get insight into this currently-opaque operations of memory accesses, and understand how they relate to the code.

Most debugging tools are built to help with the correctness of code, which is a property of the computations done on the program's state. This naturally leads to a control-flow-centric view of the program: what is the transformations and are they performing the correct behavior on their input data? Debugging works well when the tool breaks the program into small parts that can be understood and checked for correctness locally.

Performance does not nicely follow this model, since performance is a global phenomenon - the layout of data directly affects the accesses performed by other parts of the code. Performance concerns also cuts through the normal barriers between third party libraries and your own code. It's thus important to understand the memory accesses of your program using a *data-centric* approach: looking at memory accesses as events happening on data regardless of where it comes from. This needs to happen both on a global level to support understanding these concerns and on a local events level to tie events back to the code performing them, so that the programmer can change their code.

The massive amounts of data, chaotic global changes in this dataset, and the natural spatial layout of a physical memory system leads naturally to a visual representation of this data. To that end, this paper presents a visual performance tuning tool that presents global structure, local events, and aggregate statistics of a code's memory performance, with the intent of helping the user identify problems, find solutions, and implement these solutions.

The remainder of this paper is structured as follows. In section 2 we present goals for the design and use of this system. In section 3 we present the visual and interaction design of this system, and how we anticipate developers using this system. In section 4 we show the inner workings of this system and how we supported our design while meeting our design goals. Lastly we discuss related and future work.

2 Design Goals

2.1 Binary Instrumentation

Performance is an aggregate property of a software system. It depends on every layer of code which is executed, including those that the user often has no insight into: libraries and system calls. This barrier to optimization could be substantially lowered by a toolset that could instrument both. Additionally, by instrumenting code at the binary level, the resulting transparency can extent not only through all layers of the code, but through to the choice of programming language as well.

2.2 Remote Analysis

For many high performance software systems, development and production environments are completely distinct. Development often takes place remotely, and code is run on exotic hardware that does not even support a graphical environment. Running analysis directly on the backend and shipping the results to a thin visualization client would enable the user to debug performance in the same environment that he develops in. Additionally, divorcing data analysis from visualization would allow multiple developers to inspect the same performance log simultaneously.

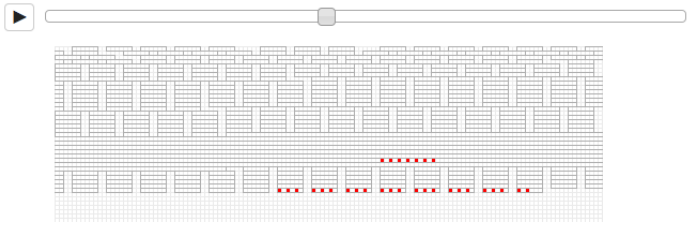

Figure 2: *Memory Map display showing global structure of memory sized to the width of cache lines, with local events marked by color as reads (green) or writes (red).*

2.3 On-Demand Analysis

High performance software systems generate enormous amounts of profiling data. Aggregating and visualizing that data statically would be intractable, both in terms of computational effor and storage requirement. This complexity could be overcome both through caching and a combination of statically computing only what can be stored efficiently and computing all other analyses on demand.

3 Design

3.1 Memory Map

We want to support understanding memory accesses in a global sense. To that end, we present a program's virtual memory space as a spatial layout of memory. Memory addresses are physically arranged in caches, where a cache line contains on the order of hundreds of individual memory locations, and subsequent accesses within a cache line have different performance characteristics than accesses that cross cache lines. This physical property of the system gives a clue to the correct visual design of the memory layout - by directly representing all of memory segmented into cache lines, there's a good impedance match between memory events of a program and the physical accesses that occur. We thus present the entire memory space as a grid of memory locations, where each row corresponds to a single cache line.

In figure 2 we show this grid. We now want to visualize memory events - actions a program performs that impacts the memory system. As we previously stated, performance tuning attempts to minimize the amount of time a code takes to run, and memory events also happen over time. This leads us to display events over time using animation.

We visualize four different types of events: memory region allocation, memory region deallocation, reads and writes.

Region allocation and deallocation affects the structure of memory, thus we present it using structural visual cues. Allocation regions are shown by highlighting those areas of the memory grid, and deallocations causes these highlights to go away.

Reads and Writes causes memory traffic to happen to a certain address. We represent this visually by highlighting the grid cell corresponding to the memory address, in red for writes and in green for reads. By distinguishing colors, it's possible to visually see copies

Figure 3: *Fireworks with alpha blending shown on memory map, highlighting memory events as they occur and overlap*

between locations happening, which is a strong indicator to optimize the performance of code by doing zero copies.

Memory performance is at a maximum when accesses are both temporally and spatially local. We want to provide a visual cue to help the user judge the spatial locality of memory access over a period of time. To support this, we show a short history of memory accesses by highlighting not just the current but the last 50 cells accessed. By showing history trails we provide a visual cue to how chaotic memory accesses are, so the user can visually judge whether accesses are happening primarily in the same cache line or jumping between cache lines. This has a direct influence on the performance of their code, so it highlights problems with code as well as hinting at possible solutions.

Given this framework we have to somehow highlight the current access and provide a strong visual cue for what's currently happening. With no highlighting of the current event, the currently described display is difficult to follow since accesses happen on individual cells of a pixel grid. By enlarging the grid slightly we make it easier to see individual accesses - a fair tradeoff against seeing less of total memory at one time.

We highlight current accesses by using a technique we call "fireworks" - highlighting a large block of memory centered on the current access, and animating this access to zoom down onto the cell it touches, shown in figure 4. This gives a very strong visual cue to the local changes occuring without sacrificing our view of global structure.

A single memory location can be accessed multiple times using both reads and writes following one another. We want to visually distinguish points in our history trail that have many accesses to them versus those that have a single access. Similarly, we want to visually distinguish fireworks that happen on top of one another depending on the type of accesses - reads or writes. We use a technique called "alpha blending" to change the color of a cell as different types of accesses happen on it. For example, a cell that is accessed by alternating reads and writes now becomes fireworks of greens and reds, and with their overlap colored yellow. In the same way, a cell that is accessed by mostly reading and sometimes writing will be colored mostly green (reads) with a small amount of red (writes). This display gives visual cues to the access patterns of an individual cell in the same way that history trails gives access patterns to different memory cells.

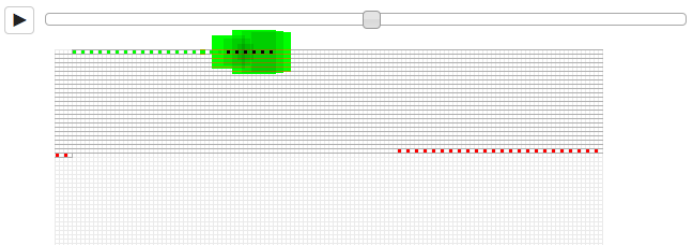

Figure 4: *Interaction happens primarily through the play/pause button, stepping the code with the keyboard, or scrubbing the time slider.*

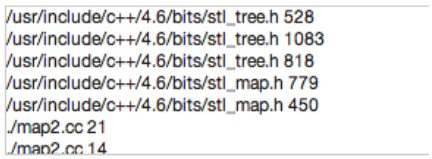

Figure 5: *Code Context window showing the stacktrace of an event.*

3.1.1 Interaction

The first level of interaction with this memory map is through watching an animation play back. The set of visual cues we've described leads to interesting events being very obvious, and begging further inspection. We provide a time slider to show progress over time and interacting with the current point in time.

We provide controls to pause the animation, and scrub through time or step forwards and backwards in time. By scrubbing an event you can roll up into a higher level view of quickly moving through accesses to compare memory behavior over larger chunks of time. If a locally interesting event happens, stepping over it gives time to mentally reason about the behavior.

This implementation of the memory map is primarily focused on identifying problems. Once a problem area has been identified, PerVERT can be paused at this point in time program's execution, and the aggregate statistics presented next can be inspected for this event.

3.2 Code Context

Every event in a memory trace is caused by a single instruction with an associated context: the stacktrace which produced it. This information is extremely valuable, and shown in figure 5. It allows the user to associate behavior in the memory map visualization (both good and bad) with the source code that produced it. Accordingly, we display stack traces for every frame in the memory map visualization, including both user code and library code.

3.3 Context Access Patterns

Once a stack context is identified as problematic, a user may wish to further inspect its behavior. One particular method for doing so, which is often useful, is to examine its stride: the address distance between the accesses that it produces. PerVERT provides two complementary visualizations for doing so.

A histogram view (figure 6) aggregates strides for a context over the entire run of the program. This view provides insight into its caching behavior. Values are binned in powers of 2 bytes up to the

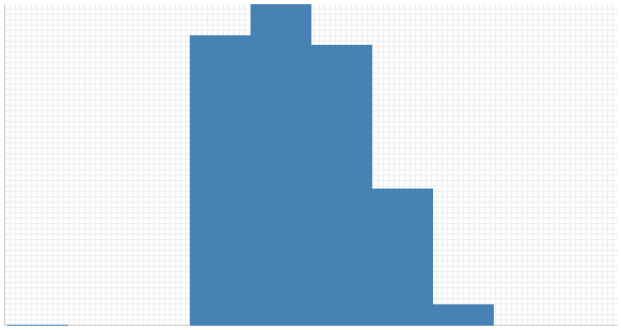

Figure 6: *Histogram showing memory access strides*

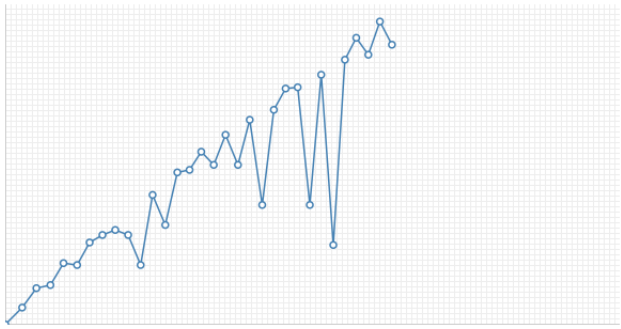

Figure 7: *Line graph showing memory accesses for a context over all time*

length of a cache line. Contexts that spend the majority of their time taking large strides, likely often result in cache misses.

A line graph view (figure 7) shows strides over all time. This view allows the user to distinguish between a context which is consistently problematic, and one that exhibits poor striding behavior over only a small segment of a program's execution.

Inspecting both these views gives an overall insight into the overall caching behavior of a program and suggests an approach for addressing non-performant code. Either change the striding pattern of the code, or modify its data layout.

4 Implementation

PerVERT consists of five separate components as suggested by Figure 8: a binary instrumentation tool for recording memory traces from program executions, an analysis engine for indexing those traces, a backend data server for performing on demand analyses on top of those indices, an HTTP JSON server for serving that data, and a front end for visualizing that data to the user. These components are described in detail below.

4.1 Binary Instrumentation

The binary instrumentation component of the PerVERT pipeline is a plugin for Intel's pintool [Luk et al. 2005]: a language agnostic JIT instrumentation framework for X86 object files. Because of this, PerVERT can be run object code irrespective of its original source language.

PerVERT produces a memory trace by intercepting each of a program's instructions, just before they are executed. Reads and writes are found by extracting the arguments from opcodes which are

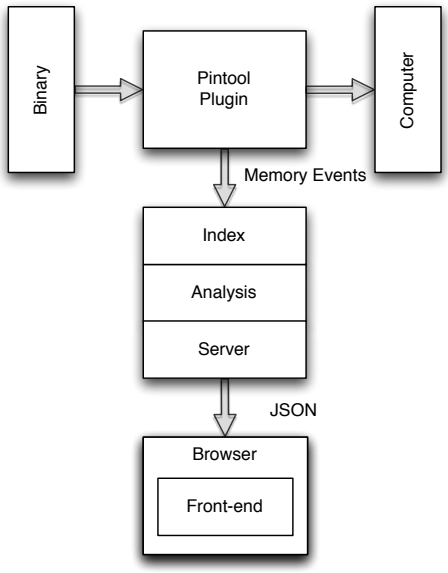

Figure 8: *The implementation of PerVERT.*

known to touch memory, wheras calls to malloc and free are found by identifying jumps to addresses which correspond to those functions in the symbol table. Reads and writes to addresses that do not appear within malloc'ed regions of the heap are ignored. The resulting trace is written out to a file, and each element is annotated with the stack context in which it occurred.

4.2 Backend

The PerVERT backend is designed to handle massive amounts of data. It is no uncommon for even a modestly sized program to generate in excess of one million memory events. The PerVERT backend builds a static index over each memory trace that it produces, and performs on demand analysis when necessary in response to user interactions. To minimize the amount of time spent transmitting the resulting data, PerVERT's analysis engine and data server are unified. We describe this architecture below.

4.2.1 Index

The PerVERT analysis engine is responsible for building static indices over memory traces. This includes tracking memory events by type: read, write, malloc, or free, as well as by context. For any event, the analysis engine records the stack context that produced the event, as well as pointers to all other events produced by that context. The PerVERT analysis engine also indexes a program's debugging symbol library so that it can decode hexademical stack traces to lists of file-line pairs.

4.2.2 Server

The analysis engine performs a set of operations whenever a new binary is traced. Once this analysis is complete, the data has to be made available in a form that's quickly transmitted and easily consumed by the front-end. As the user explores a program, some analysis has to happen dynamically, so events on the front-end has to trigger new analyses. Since we want the front-end to be divorced from the analysis engine, so that analysis can happen close to the code while the front-end can run on a different machine, we implement a client-server infrastructure where the data server is con-

tained inside the analysis engine.

The server itself takes the form of a HTTP JSON server. The server publishes a set of paths that returns information to the front-end. Each path is associated with a different type of information, and takes parameters to specify the exact version of the information we're interested in. For example, the memory map publishes a path that computes the set of memory events and regions at a specific point in time for a specific executable. The front-end can thus request exactly and only the information it needs to display to the user, and the back-end does not need to track any state.

The server itself is engineered in the "Rack"-style[Neukirchen] of web servers, where multiple layers of middleware are plugged together to provide a full stack web server written as a set of modules.

The server runs presistently in the background, even between traces of executables. This makes it possible to capture multiple traces of the same executable for future analysis, and capture multiple different executables so that multiple people can use the tool on the same machine.

Since the API is completely stateless, multiple users and view the same program at the same time, and all data transferred can be cached on either side. This allows for complete flexibility in when analysis happen and how data is caches for the most efficient view of the data.

4.3 Frontend

The visualization is built as a HTML5 ajax application. All user interaction initiates asynchronous requests for textual data from the back-end. This information is then rendered as visualizations using a suite of toolkits.

Any communication to the backend is stateless. To support a highly interactive experience, all back-end data is cached. The user can now perform quick comparisons between data by flipping back and forth without having to request data from the back-end. It also lowers the stress on the back-end server.

The memory map requires drawing a large space with custom visualizations. Since performance of this view is critical, we implement it by drawing pixels on a HTML Canvas element. We stack several of these canvas elements to avoid having to redraw all the layers every frame. The alpha-blending of different accesses is implemented as a standard Alpha Compositing algorithm on the pixel level.

Graphs of aggregate statistics are built using D3, and drawn on-demand by events firing from the user's interaction with the time slider, or as time animated. These graphs each fire their own data requests to the caching layer, and can thus easily be swapped out for different displays.

5 Example Applications

5.1 Algorithm

To evaluate PerVERT, we built a test suite of algorithmic variants for a same simple routine which one would expect to find a scientific computation kernel. (1) Create a container of objects on the heap. (2) Write values to each of those objects. (3) Read values from each of those objects. (4) Free the objects and the container.

The variants differ only in the type of container used.

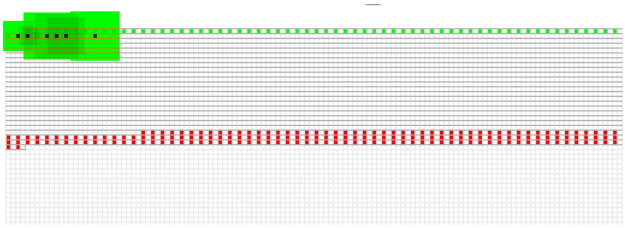

Figure 9: *C Array example*

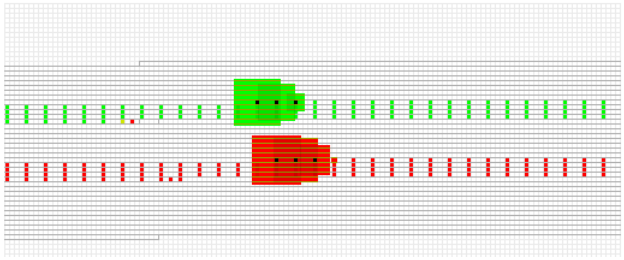

Figure 10: *C++ STL vector example*

5.2 Variants

5.2.1 C Array

For this example we inserted and read a set of integers into and from a C array, shown in figure 9.

In the C++ Array view, the animation reveals two distinct sections of the code's behavior. Initially a linear progression of writes to memory occur as values gets inserted into the array. Once this is complete, a second linear pass over the array reads values back.

5.2.2 C++ STL Vector

For this example we inserted and read a set of integers into and from a C++ vector, shown in figure 10.

The vector starts off as a short 4-element array. As values are written to this array, the dynamic resizing of the array and the cost associated with that can immediately be seen: after 4 writes, a new array is allocated and all the previous values are copied to this longer array. This pattern of inserts, followed by enlarge and copy, now makes up the first section of the array. The same behavior of the array's second section can still be seen, where a linear read of the entire final array happens.

5.2.3 C++ STL Map

For this example we inserted and read a set of integers into and from a C++ map, shown in figure 11.

The map approach has similarities to the STL vector, where lots of resizing and copying can be seen. As the animation is viewed, every insert starts with a traversal of multiple cache lines before it finds the part of the map that stores the bucket for the given item. Once the insert happens, multiple writes occur as it inserts and sorts the bucket list. Reads out of the map also is a significantly more complicated algorithm now - multiple cache lines are traversed to find the appropriate bucket, the bucket is searched for the item, and the item is returned.

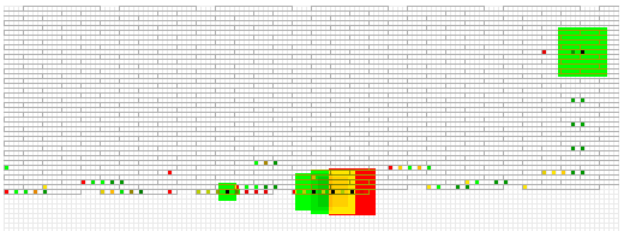

Figure 11: *C++ STL map example*

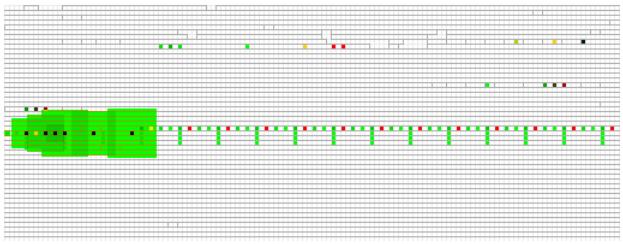

Figure 12: *Haskell example*

5.2.4 Haskell List

For this example we inserted and read a set of integers into and from a haskell list, shown in figure 12.

Since we are language agnostic, we used the tool to inspect haskell code that performs the same operation. In this case there is a significant amount of startup accesses happening, which appear to initialize the haskell environment. A linear array still gets built, but bookkeeping happens at the beginning and end of every access. This display makes it very obvious where haskell incurs an overhead above the equivalent C++ implementation.

6 Related Work

Visualizing Memory Graphs[Zimmermann and Zeller 2001], made the same observation that you want to drill down into memory structures, but they took a very different approach: they care about structures being built rather than than access to these structures. While we find this to be a useful inspection for debugging purposes, it does not directly show performance. It's a great example of memory visualization for debugging rather than performance.

KCacheGrind allows you to visualize the static structure of your code - similar to our stack traces - and also attaches performance events to it. While we take a very visual approach to this, KCacheGrind bubbles up statistics directly related to cache performance. This is useful for identifying issues, but we claim that our approach gives a direct visualization of the reasons behind the cache performance.

7 Future Work

Future work for this project could proceed in two complementary directions: one in which we explore new methods for visualizing and interacting with data, and another in which we explore new methods for data collection and analysis.

7.1 Visualization and Interaction

Comparing alternative implementations of the same program is a difficult problem. Even a small change in access patterns or choice of data structures can render two traces nearly unrelatable. This could be mitigated somewhat by exploring techniques for canoncalizing memory event streams and identifying patterns that are invariant between two streams. Another complication worth exploring is prohibitively large memory event streams. Future work could examine techniques for aggregating more data into a smaller part of the memory map, user-defined methods for filtering data based on events of interest, and tools for bookmarking those events either manually or automatically.

7.2 Data Collection and Analysis

PerVERT currently visualizes a program's use of the virtual address space, but says nothing about physical memory. However because performance is truly a function of physical memory usage, it would be useful to visualize that space as well. One way of doing so without being invasive would be to build in a cache simulator. Other areas that could be improved include support for analyzing incomplete data, or for intelligently sampling from prohibitively large memory event traces.

8 Conclusion

In this paper we presented PerVERT, an instrumentation framework for logging a process's virtual memory traffic and a visualization suite for reasoning about common memory performance bugs. We described the design goals that lead us to its current form, and through several small prototypical examples, demonstrated the effectiveness of its current design. In its current state, PerVERT functions well as a final class project. However, it is our intention to continue development as described above, and to evolve its design into a mature research tool.

References

GRAHAM, S. L., KESSLER, P. B., AND MCKUSICK, M. K. 2004. gprof: a call graph execution profiler. *SIGPLAN Not. 39* (April), 49–57.

LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, ACM, New York, NY, USA, PLDI '05, 190–200.

NETHERCOTE, N., AND SEWARD, J. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, ACM, New York, NY, USA, PLDI '07, 89–100.

NEUKIRCHEN, C. Rack: a ruby webserver interface.

STALLMAN, R., AND PESCH, R. 1991. *Using GDB: a guide to the GNU source-level debugger*. Free software foundation.

ZIMMERMANN, T., AND ZELLER, A. 2001. Visualizing memory graphs.