TraceVis: An Execution Trace Visualization Tool

James Roberts
NVIDIA Corportation
12331 Riata Trace Pkwy
Austin, TX 78727
Email: jroberts@nvidia.com

Craig Zilles
Dept. of Computer Science
University of Illinois at Urbana-Champaign
201 N. Goodwin Ave.
Urbana, IL 61801
Email: zilles@cs.uiuc.edu

***Abstract*— This paper describes TraceVis, a tool for graphically exploring application behavior as it relates to its execution on a microprocessor. Because the human mind can process enormous amounts of visual data–readily identifying trends and patterns–such a tool facilitates performance analysis by allowing raw data to be inspected rather than summaries of pre-selected characteristics. To this end, TraceVis has been designed to enable interactive navigation of many kinds data useful for studying performance problems including: relating the execution to disassembly and high-level language code, annotating the program trace with events (*e.g.*, branch mispredictions and cache misses), and tracing instruction dependencies. In addition, TraceVis provides mechanisms for searching in traces and annotating traces (to bookmark areas of interest or mark regions previously characterized) to allow an analyst to focus their attention on the desired behaviors and regions. Along with exhibiting TraceVis's features, this paper demonstrates how these features can be used in conjunction to analyze the performance of a simulated microprocessor's execution. TraceVis is available in source form for non-commercial use [1].**

I. Introduction

Modern microprocessors have become complex to the point that an intuitive analysis of a program's execution is no longer possible through mere inspection. Innovations such as pipelining, out-of-order execution, memory hierarchies and prefetching have created concurrency and non-determinism in the microarchitecture that obfuscate analysis of even the simplest program. The complexity of modern machines justifies (at least in part) the shift toward qualitative computer architecture research, which has led to the development of a number of high-quality, publicly-available simulation infrastructures.

While these simulators enable collecting performance estimates of novel (micro)architectures, few offer much support into understanding the sources of the changes in performance. Program behavior is often known only in terms of broad generalizations (*i.e.*, summary statistics like average fetch time, IPC, hit rates, etc.). As a result, too many research papers stop short of fully explaining how their proposals impact program execution. We believe this is in part due to a lack of tools that enable the inspection of execution behavior.

A well known alternative to naive aggregation is the use of visualization. Visualization finesses the need for aggregation by exploiting the human's innate ability to efficiently process tremendous amounts of visual information and to identify patterns and anomalies in that information. While a number of high-quality commercially-available tools exist for visualization of parallel program execution (*e.g.*, VAMPIR [2] and PARAVER [3]) and some microprocessor vendors have developed in-house tools (*e.g.*, [4]), the quality of publicly-available visualization tools is not nearly comparable to that of architecture simulators.

To address this need, we have developed TraceVis, a flexible and functionality-rich visualization tool for analyzing microarchitectural simulation data. In designing TraceVis, we had the following eight goals:

1) **Easy access to high-level information.** Coarse-grained information should be available with minimal effort.
2) **Access to increasingly low-level information on demand.** Upon identifying a point of interest at a high-level, users should be able to work down to levels of increasing detail in an intuitive manner.
3) **Interactiveness.** Users should be able to manipulate views and obtain data with minimal latency. Furthermore, the tool should remain responsive regardless of the size of the trace and/or the granularity at which it is being viewed.
4) **Search-ability.** Given a set of user-specified parameters, the tool should be able to locate regions of the trace that match those criteria.
5) **Ability to detect patterns, phases and anomalies.** The tool should aid users in distinguishing regions of similar behavior from those of distinct behavior.
6) **Ability to annotate traces with persistent information.** Users should be able to specify and associate information with a trace as a means for tracking progress, identifying points of interest and conveying their interpretations to collaborators.
7) **Flexibility in usage and extensibility to other systems.** The tool should be general enough that it is capable of visualizing a wide range of microarchitectures.
8) **Visualize everything.** Minimize the need for users to read text-based or numeric results.

This paper demonstrates the features of TraceVis in two ways. First, we briefly overview some of its basic functionality (in Section II). Then, in Section III, we demonstrate a detailed usage scenario that exhibits how these features can be used in performance analysis. Section IV discusses a few implementation issues. Section V provides a brief survey of related work. Lastly, Section VI provides a conclusion.

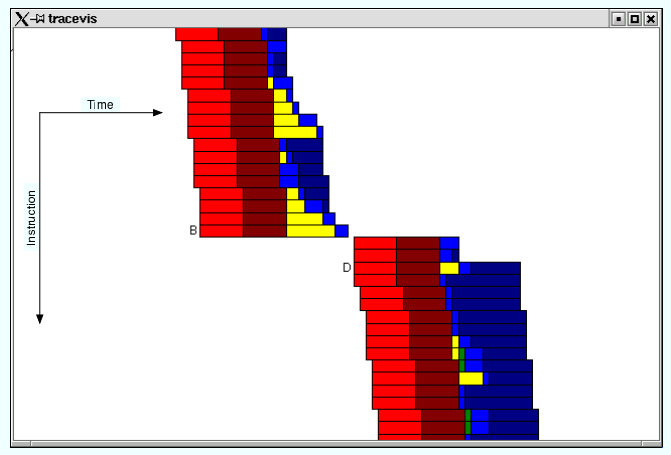

Fig. 1. The basic trace graphing functionality of TraceVis. *Instructions are drawn as horizontal bars and subdivided into color-coded regions each representing a pipeline stage. The 'B' and 'D' denote a branch misprediction and a data cache miss, respectively, for the instruction to its right.*

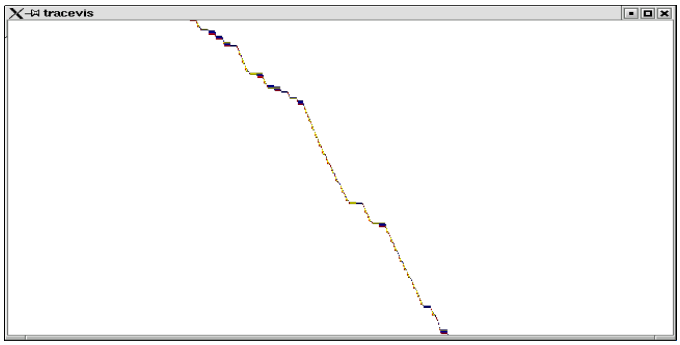

Fig. 2. TraceVis zoomed-out. *TraceVis is capable of rendering at arbitrary levels of zoom. Here 15k instructions are rendered in the image.*

II. Features

In this section we introduce and discuss the key features of TraceVis. We start with the most basic features and proceed to more involved features. Whenever possible we attempt to furnish actual screen shots of the tool to better illustrate its functionality.

A. Basic Trace Graphing

The focal point for TraceVis's functionality is its trace graphing ability. The trace graphing feature enables users to visualize execution traces as instructions flowing through the processor pipeline. Figure 1 shows a sample trace graphing from TraceVis. In the graph, instructions are represented by rectangles whose width denotes the lifetime of the instruction. The instruction bars are sub-divided into color-coded regions which represent different stages in the instruction's lifetime (*e.g.*, fetch-time, decode-time, etc.). The instructions are arranged along the y-axis in program order and along the x-axis according to cycle-time of the instruction's events. All graphed instructions are non-speculative; that is, TraceVis does not display bad-path execution arising from branch misprediction or other misspeculated events.

This representation is similar to the notation used in architecture texts (*e.g.*, [5]) and other pipeline visualization tools [4], [6]. This intuitive representation allows users to quickly determine how instructions flowed through the execution pipeline and how instructions interacted with one another in the course of their lifetimes.

As shown in Figure 1, TraceVis can annotate instructions with a set of events that relate to the instruction. Example events include: branch mispredictions (B), instruction cache misses (I), data cache misses (D) and load/store ordering violation pairs (L,S).

Navigation about the trace (in 2D) can be performed via either keyboard or mouse control. Keyboard control allows users to quickly page up and down a trace. When paging up or down, TraceVis automatically adjusts the x-offset of the trace so that the trace body remains centered in the frame. Mouse-based navigation allows users to interactively wander through the trace by dragging the trace in any direction.

TraceVis is also capable of arbitrary levels of zoom. Figure 2 shows the same trace from Figure 1 pulled back to a much lower level of zoom. From this vantage point it is possible to view large scale program behavior and to quickly survey the trace for regions of particularly interesting phenomena (*e.g.*, low IPC). Interactive zooming then allows users to zoom-in on points of interest and diagnose its cause.

Due to limited pixel space and processing capability, TraceVis reduces the level of detail of its trace rendering as the view is zoomed-out. For instance, beyond a certain level of zoom, the individual event annotations are no longer rendered (in Section II-E we will discuss methods for reclaiming information about these events at this level of zoom).

Also, when zoomed-out beyond a certain zoom-level threshold TraceVis begins rendering only a sampled subset of the instructions within the visible range (*i.e.*, a fixed number of instructions per scan line). This effective sampling of the trace data is necessary in order to maintain interactive frame rates, especially when rendering regions that may include millions of instructions. In practice, we have found that the sub-sampling of the trace data is largely imperceptible at these low levels of zoom.

B. Searching

TraceVis includes a mechanism for searching a trace for a specific instruction address, event (*e.g.*, branch misprediction, cache miss, etc.) or any combination of the two. Figure 3 shows the search feature. The window on the left is where the user specifies the search criteria. Upon initiating a search, TraceVis grays out all the non-matching instructions and moves the first matching instruction to the top of the trace window. Repeating the search function advances the trace to the next matching instruction.

C. Static Code Correlation

TraceVis includes functionality for correlating instructions in the dynamic trace back to the low- and high-level source code. The reverse operation (*i.e.*, correlating static code to

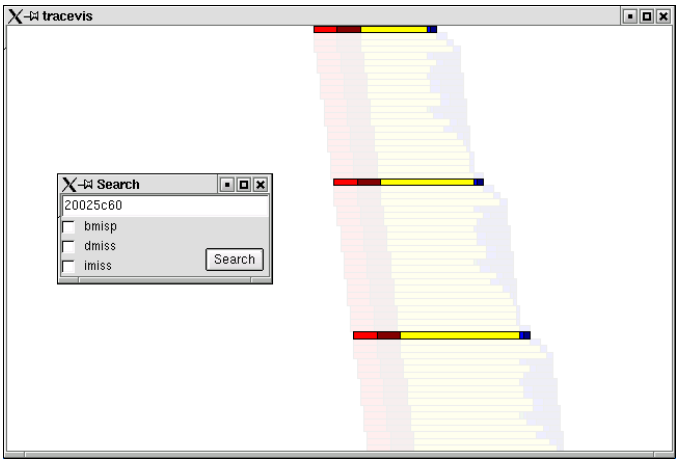

Fig. 3. TraceVis searching for matching instructions. *The first matching instruction is moved to the top. Non-matching instructions are grayed-out.*

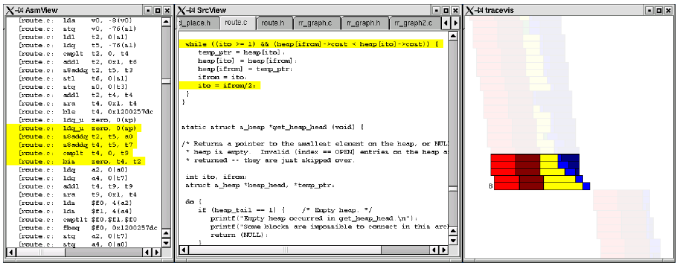

Fig. 4. Dynamic-to-static code correlation. *Highlighted assembly, C source code, and dynamic instructions for the same set of static instructions.*

the dynamic trace) is also possible. Figure 4 shows TraceVis correlating dynamic trace instructions back to both assembly code and C source code.

To use the dynamic-to-static code correlation feature, the user selects a region of the dynamic trace using a bounding box. TraceVis then highlights all lines of high- and low-level source code that match the address of any selected instruction. Since the highlighted regions of text may not be contiguous or even on screen, TraceVis incorporates functionality to allow the user to jump among the highlighted regions of text using keyboard controls.

Correlating static code to the dynamic trace works in a largely symmetric fashion. The user selects lines of source code on either the high-level or low-level views. TraceVis then grays out all but the matching regions in the trace.

D. Static Code Coloring

Figure 5 shows the static code coloring capability of TraceVis. This feature operates in a very similar fashion to the static-to-dynamic code correlation mechanism. The user selects a region of static code and specifies a color for that region. TraceVis then colors regions of the dynamic trace accordingly. Conversely, the user can also select regions of the dynamic trace and request that all static instructions in that region be colored.

The coloring feature differs from the correlating feature in

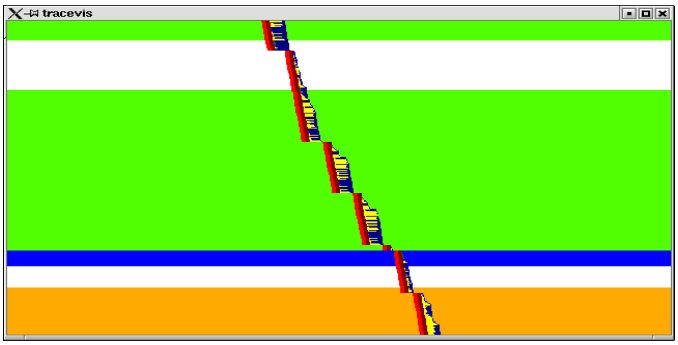

Fig. 5. Static code coloring. *Instructions within a colored region share a common set of instruction address. Uncolored regions contain instructions with addresses that are not in any of the designated sets.*

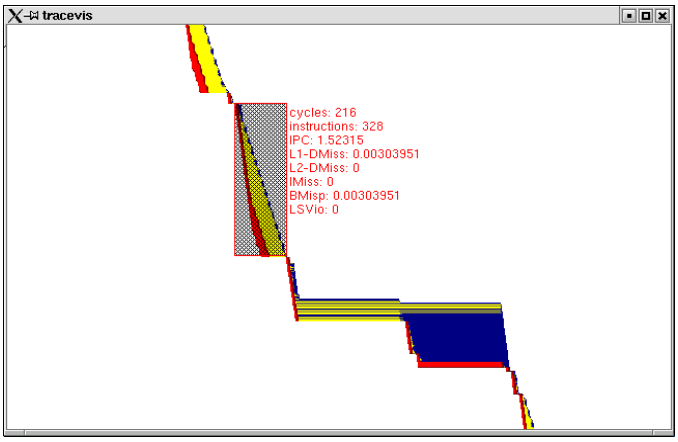

Fig. 6. Regional Statistics. *The statistics pertain to the selected region. The figures shown are fractions, not percentages.*

that: 1) multiple colored regions can co-exist on the same trace, and 2) the coloring feature is persistent across TraceVis sessions; that is, the regions and colors are stored to disk for later re-use. On the whole, the correlation mechanism is meant to be a light-weight operation for quick analysis, and the code coloring mechanism is meant to be for less ephemeral annotation.

Region coloring is useful for tracking progress of a user's code exploration. That is, once a section of code has been investigated, the user can color that region to denote that the region (and by extension, all other regions of the same static code) have already been explored. In the same vein, TraceVis provides support for bookmarking so that regions of interest can be annotated. In Section II-E we will discuss how code coloring can be used in conjunction with statistics graphing for the purpose of tracking program phases.

E. Aggregating Statistics

Since TraceVis offers arbitrary levels of zoom, at some point it becomes valuable to summarize data such as cache misses or branch mispredictions into relative rates. TraceVis provides two features for aggregating statistics: region statistics and statistics graphing.

The region statistics feature allows the user to select a

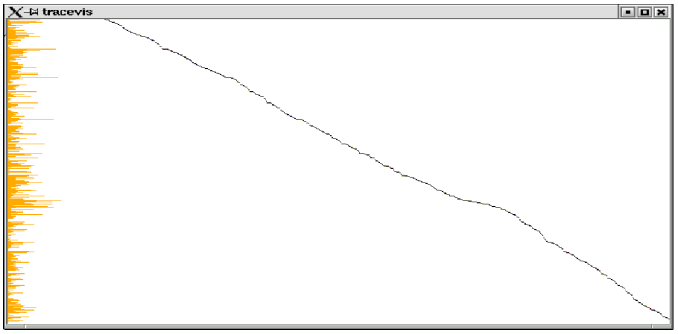

Fig. 7. L2 miss rate visualization. *The histogram depicts the instantaneous L2 miss rate.*

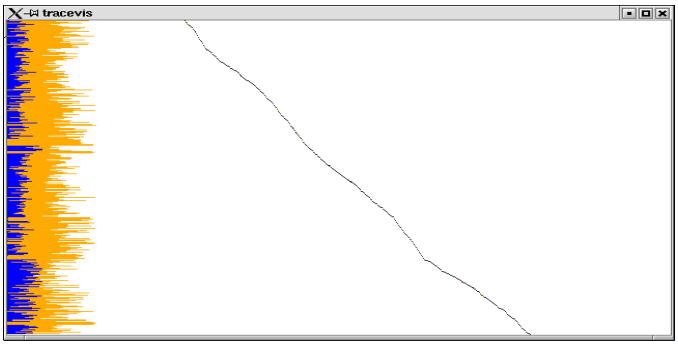

Fig. 8. Code composition visualization. *The colored bars in the histogram reflect the frequency of instructions from the correspondingly colored regions of static code.*

portion of the trace and compute summary information for that portion of the trace. As shown in Figure 6, the region is annotated with the number of cycles and instructions encompassed, the average IPC, and the event rates for the available events. Event rates are reported in number of events per instruction.

The statistics graphing feature plots the frequency of one event type as a histogram along side the program trace. For example, Figure 7 shows TraceVis graphing the relative frequency of L2 cache misses. Each line of the histogram corresponds to the average frequency of L2 cache misses for all instructions represented on that scan line. The statistics graphing feature is particularly valuable when the trace is zoomed out to the a point where individual event annotations (such as those shown in Figure 1) would be too small to be legible.

In addition to plotting event frequencies, TraceVis can plot trace composition with respect to colored static regions, as shown in Figure 8. For each scan line, the corresponding line on the histogram shows the relative makeup of all the instructions represented by that scan line (in terms of the colored regions of static code). For example, a histogram line that shows equal lengths of the two colors implies that the instructions represented on that scan line are composed of an equal amount of static instructions from the two colored regions.

In order to maintain interactive frame rates, statistics graphing currently utilizes a statistical sampling method for obtaining its results. Again, while this methodology adds imprecision to the visualization, we believe the accuracy is sufficient for the purposes of this tool.

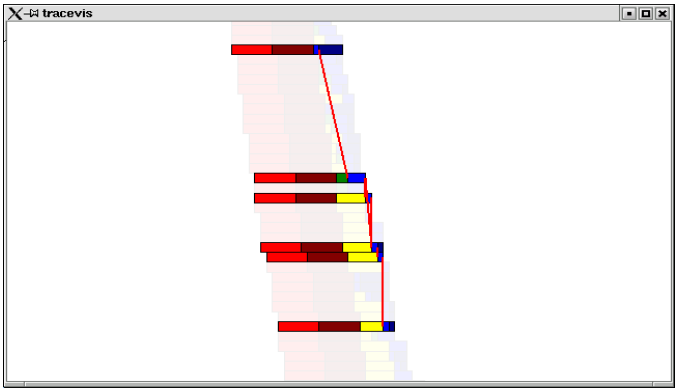

Fig. 9. Dependence Chain Visualization. *The lines between instructions identify the data dependence chain originating from the bottom-most instruction.*

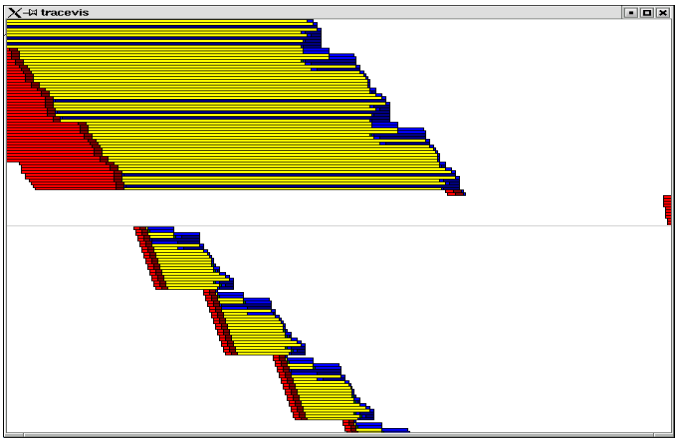

Fig. 10. Multiple Core Visualization. *Two program executions shown simultaneously.*

F. Dependence Graphing

Figure 9 shows the dependence graphing functionality of TraceVis. To use the feature, the user selects a dynamic instruction and requests a dependence graph for that instruction. Then TraceVis graphs out the backward dependence information and grays out the non-involved instructions. Since dependence information potentially extends far back into the trace, the depth of the dependence tree traced is capped by a user-definable setting.

G. Multi-core Traces

Lastly, TraceVis includes support for for visualizing multiple traces simultaneously, which is useful for analyzing parallel or multi-threaded workloads on chip multiprocessors or multi-threaded processors. Figure 10 shows the TraceVis screen split between two separate program traces. The two views are synchronized in such a way that motion or zoom adjustments in either view is automatically mirrored by an commensurate adjustment of the other.

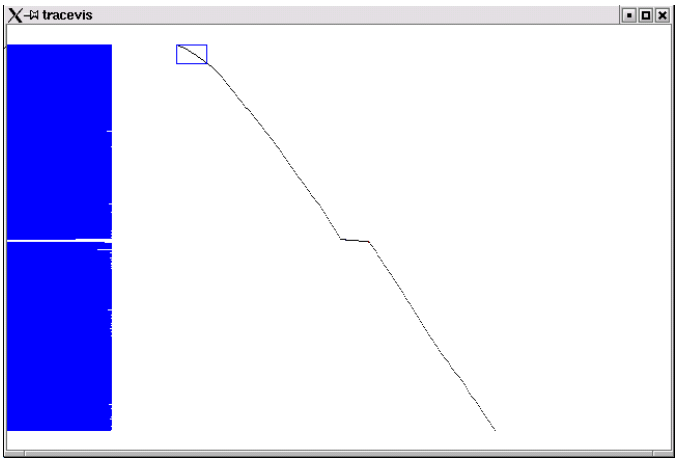

Fig. 11. Visualization of a 10M instruction-long trace of `vpr`. *This trace was captured several billion instructions into the benchmark's execution. The trace with static instruction composition with respect to the boxed region at the trace's beginning; the nearly solid histogram on the left shows that most of the trace uses the same instructions as the beginning of the trace.*

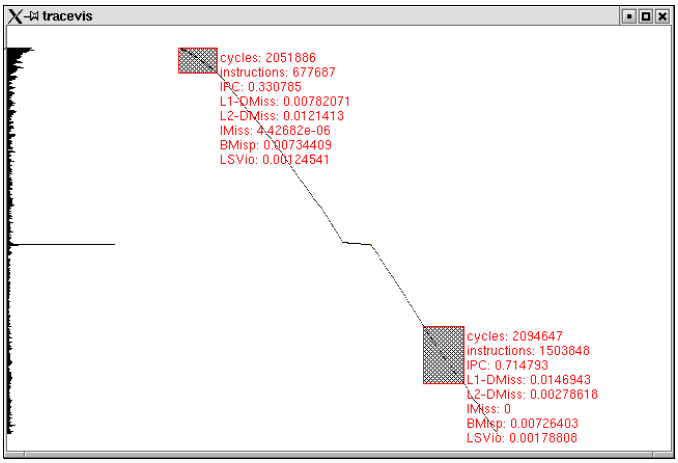

Fig. 12. L2 data cache miss rates and regional statistics (using fractions, not percentages). *Both the histogram of L2 misses and the regional statistics suggest that the warm-up code incurred more performance-degrading events than the steady-state regions.*

III. Use Cases

In this section, we present a sample usage of TraceVis, demonstrating how its features enable an interactive analysis of an execution. For the sake of exposition, we have selected a scenario of modest complexity, but the methodology applies to more complicated examples, as well. We start at a high level and then work down to progressively lower levels of detail.

A. Full-Trace Visualization

Figure 11 shows TraceVis visualizing the full length of a 10M instruction-long trace of the SPEC2000 benchmark `vpr`. This particular trace was recorded several billion instructions into the benchmark's execution. At this high-level view, rendering individual instructions is not feasible or even useful. Rather, instructions are aggregated and rendered as a single summary instruction for each scan line. With 10M instructions and roughly 500 scan lines, it follows that each scan line contains summary information for roughly 20k instructions.

Despite the coarse granularity of the information on this graph, we can gain several key insights into the application's overall behavior. Most obviously, we can gauge the application's IPC (*i.e.*, rate of execution) by observing the slope of the graph's curve. Based on this IPC analysis, we can resolve three distinct phases in the trace's execution: 1) a decreasingly shallow-sloped region at the trace's beginning, 2) a very shallow-sloped and distinctly demarcated region in the trace's center, and 3) the nearly uniform-sloped region that comprises those parts of the trace not in the other two regions.

B. Warm-up Phase

First we investigate the shallow-sloped region at the start of the trace. Here we suspect that the low IPC of this region is the result of not "warming up" the simulated machine before starting trace collection. That is, the low IPC in this region is not a function of the code being executed, but rather is due to training of processor structures such as the branch predictor and the cache (*i.e.*, compulsory cache misses).

Using TraceVis, we can validate our theory by showing that a) the static code being executed in the initial phase is the same as the code executed in the rest of the trace, and b) that performance-degrading events (*i.e.*, branch misprediction rate, cache miss rate, etc.) occur at higher rates in the initial phase than in other regions executing the same static code.

First we prove that the code in the warm-up phase is the same code that is executed in the region of higher IPC. Figure 11 shows TraceVis graphing the composition of the execution trace with respect to colored regions of the static code. The box around the top left portion of the trace is the region from which the instruction addresses were selected. That is, we selected that region of the dynamic trace and colored all the static instructions within the region. Upon graphing out the composition based on our set of selected static instructions, we find that nearly the entire trace is composed of those same instructions. From this, we can conclude that the code in the start-up region is no different from the rest of the trace in terms of the code that it is executing.

Now we show that the low IPC of the warm-up phase can be attributed to warm-up-related events. The TraceVis screen shot in Figure 12 has TraceVis graphing the L2 cache miss rates on the left and two regional statistics on the right. From both the histogram and the regional statistics, we can quickly observe that the L2 miss rate for the warm-up period is significantly higher than that of the rest of the trace (with the notable exception of the horizontal region in the trace's center). Since L2 misses carry a high penalty in this processor model, the degraded IPC could reasonably be attributed these misses.

C. Mid-trace Plateau

Next we investigate the shallow region at the trace's center. As we saw in Figure 11, the region at the center of the trace executes a set of static code that is different from the rest of

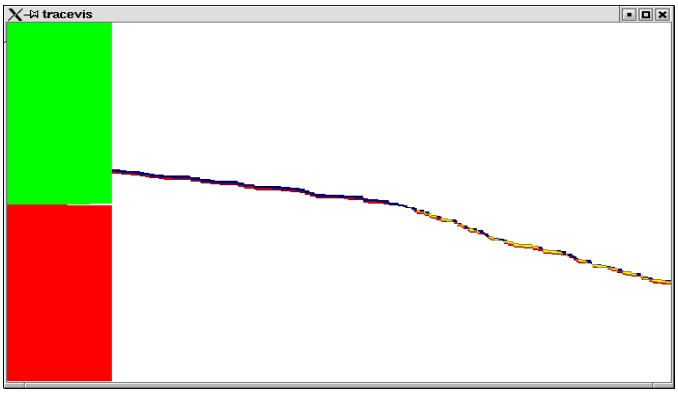

Fig. 13. A detailed view of middle region. *Roughly 6k instructions are represented in this image. The histogram to the left shows the composition of the trace in terms of static instructions. The histogram shows that the middle region is composed of two distinct phases – each one executing a different set of code.*

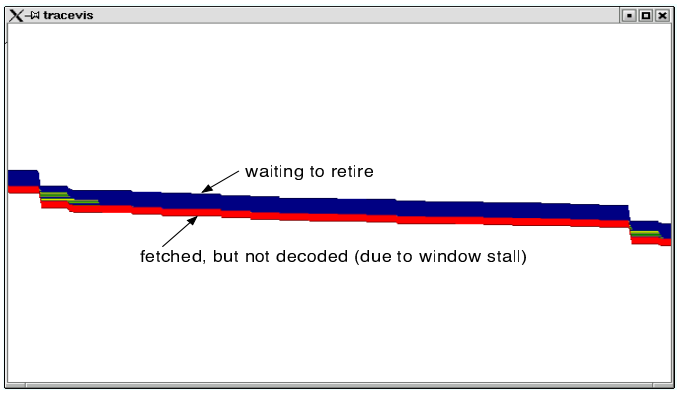

Fig. 14. The top portion of the middle region. *The data in this graph suggests that most of the processor time is spent in fetch stalls and waiting-to-retire stalls. In some cases instructions are in the pipeline for as long as 10,000 cycles. Intuition suggests that the root cause of the stalls is store misses; source code correlation backs up this theory.*

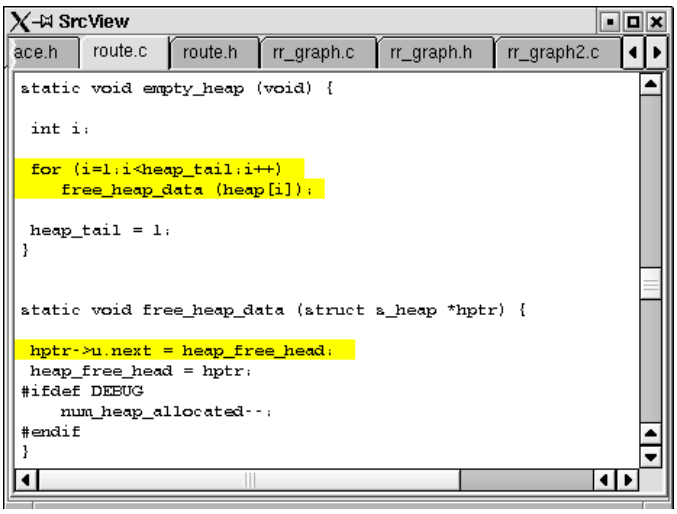

Fig. 15. Source code for the code region of Figure 14. *Re-initializing all of the `hptr` data structure would account for the large number of store misses visible in the trace graph.*

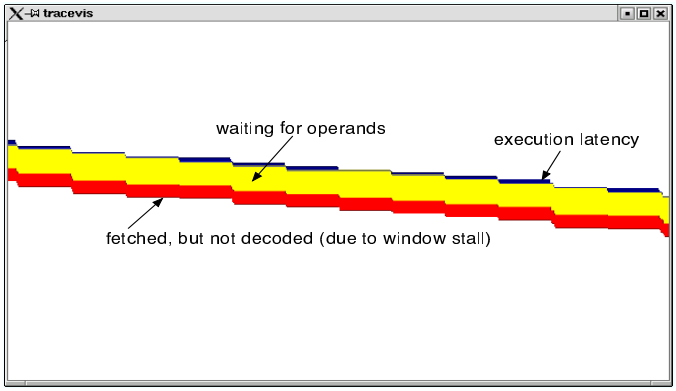

Fig. 16. The bottom portion of the middle region. *The data in this graph suggests that most of the processor time is spent in waiting-for-operand stalls. The stalls are the result of load instructions missing in the L2 cache.*

the trace.

Figure 13 shows a zoomed-in view of the trace's center. The trace visualization shows that the center region itself can be broken down into two distinct regions – each one executing a completely different set of static instructions.

Top Phase: We investigate the top phase of the center region in Figure 14. Here we see that the phase is dominated by two pipeline stages: *fetch* and *waiting-to-retire*. Since our sequentially consistent machine retires instructions in order, we can reason that the long fetch latencies are caused by instruction window back-pressure created by the long *waiting-to-retire* latencies. Using the instruction details information, TraceVis reveals that the long *waiting-to-retire* latencies are themselves caused by L2 D-cache misses. Using the static code correlation mechanism of TraceVis we can determine that the stall inducing instructions arise from the following line of assembly code: `stq a1, 8(a0)`.

This discovery intuitively makes sense: Each store executes in a single cycle – which accounts for the fact that there is very little execution latency observed in this region – but, since the store misses in both the L1 and L2 caches, it can take an additional 400 cycles to retire. Furthermore, from the trace visualization, it appears that the store misses are being somewhat serialized in this region. After careful analysis of the trace, we were able to trace this behavior back to a bug in our simulator which previously went undetected.

Using the same static-to-dynamic code mapping mechanism we can also find the corresponding high-level source code. Figure 15 shows the relevant lines of source code. From the source view we can see that the entire trace region consists of only three lines of source code. We can also see the basic structure of the behavior and the root cause for the performance penalties: the application is freeing a data structure by re-initializing all of that structure's pointers by pointing them to a common target. Assuming that this structure is not within the working set, it is clear why this high-level behavior would cause store misses.

Bottom Phase: Now we investigate the remaining portion

of the trace's middle region. Figure 16 shows this phase to be dominated by *fetch* stalls (again the window has filled) and *waiting-for-operands* stalls. Using the same mechanism as above, we find that the long latencies are caused by load instructions that miss in the L2 cache. This result too makes intuitive sense; the dominance of *waiting-for-operand* behavior in this region can be attributed to long latency load instructions that create chains of stalled dependent instructions.

From the correlated high-level source code, we see that in this phase the application is again re-initializing a data structure. In this case however, re-initialization is performed by walking a linked-list structure. This pointer chasing behavior can reasonably explain the prevalence of L2-missing load instructions, especially if the structure is large and/or not in the working set.

D. Summary

The findings in this section are significant not only in terms of the insight they provide to the workings of the application and the simulator; they are also significant in terms of the manner in which they were obtained. All the conclusions reached were done so using TraceVis's interactive visual environment. Using a high-level summary, we were able to quickly identify regions of distinct behavior. Then, using increasingly low-level information, we were able to identify specific causes of behavior at the level of the microarchitecture, the machine code and finally the programmer-level algorithm.

IV. Implementation

In this section we discuss some of the implementation details of TraceVis. For each case, we state the motivation and and the rationality for our decisions and then present drawbacks and alternatives to these decisions.

A. Language and Windowing Toolkit

TraceVis was written in C++ using the Qt [7] windowing toolkit. The motivation for this choice in language/windowing system was based on the fact that TraceVis needed to be both high performance and cross-platform compatible. In this regard, C++ afforded a good deal of performance, and, since it is available on Windows, Mac and X11-based systems, Qt allowed TraceVis to support multiple platforms easily.

TraceVis does not make use of OpenGL [8] or any other graphics hardware rendering libraries. The decision to use exclusively CPU-based rendering was the result of preliminary tests which showed that the graphics demands of TraceVis could be met by a moderately equipped system (Intel Pentium 4, 2.0Ghz, 512MB RAM) without having to resort to hardware acceleration. If the need arises, we believe that extra performance could be gained by leveraging graphics hardware found on most commodity PCs.

B. Trace File Format

Currently our trace format stores each instruction in 40 bytes: a 64-bit PC field, a 64-bit start cycle, a 16-bit duration for each of seven pipeline stages, three 16-bit fields for holding the distances (in instructions) to producers of its register operands, and a 32-bit vector of event flags. This relatively tight encoding allows a 10 million instruction trace to be stored in 400 MB (or less than 50 MB when compressed with bzip2), which can fit in the physical memory of most workstations. Writing a 10 million instruction trace file only adds 8 seconds to the execution time of our timing simulator.

TraceVis has been designed to be flexible to support retargetting to other simulators and expansion of its capabilities. Specifically, its trace format is encapsulated from the display code, which uses accessor functions to read the trace file, enabling the trace format to be modified with a minimum of effort. Also, we provide a patch for the widely used SimpleScalar simulator [9] to generate our trace format. The statistics counting and display code has been likewise implemented in a generic fashion to allow new metrics to be plotted with a minimum of effort.

C. Multi-View Synchronization

Because TraceVis's internal format–like many other tools–is indexed by instruction, it is not trivial to do the synchronization of multiple views as described in section II-G. The synchronization requires determining the instruction index for a given cycle, when the data structure is set up to do the relation in the other direction. To address this difficulty, we propose an efficient interactive algorithm for view synchronization [10] that avoids the space overhead of building a reverse mapping data structure.

D. Limitations

TraceVis has two main limitations, as it stands. First, no support is available for visualizing wrong path instructions. This could easily be rectified by including the wrong path instructions in the trace, flagged so they could be rendered differently; the drawback of doing so is that then the slope of the rendered trace is no longer a function of average IPC. We believe it would be preferable to display the wrong path instructions next to the correct path instructions, but then there is no longer a linear relationship between the place in the trace and a screen location, something our renderer currently relies on for performance. We are currently exploring a two trace scheme where wrong path instructions are stored in a second trace and rendered as an overlay when zoomed in on the trace.

Second, there is no support to visualize the memory behavior of the program. A few desirable features include: 1) being able to observe memory dependences between instruction in the same way that register dependences are now visualizable, and 2) the ability to relate primary and secondary misses to the same cache block to facilitate distinguishing the number of independent data cache misses for a given region of the program. These features are a focus of future work.

V. Related Work

In this section, we briefly discuss related work. In short, most published tools are either not available or are closed-source products that must be licensed, and and TraceVis compares favorably—in terms of features and performance—with the open source tools.

VAMPIR [2] and PARAVER [3] are representative of the collection of visualization tools designed for performance debugging of parallel applications. These tools focus on the communication and synchronization behavior of parallel programs (typically instrumenting programs by profiling calls to a parallel library (*e.g.*, MPI)), but potentially could be adapted to study questions at the granularity considered by most architecture research. Both of these tools are closed source and require commercial licensing.

Reilly, et al. describe an in-house tool used by Alpha engineers for visualizing Alpha 21264 executions [4] and we suspect that other vendors have comparable tools. While there are similarities with TraceVis, screen shots—the article has few details—show a view where processor state (*e.g.*, which instructions are being decoded) is plotted over time.

Stolte, et al. [11] demonstrate an application of the Rivet Visualization Environment for execution visualization. The Rivet tool offers trace visualization featuring a multi-tiered statistics graph, a view of the high-level source code and a fully-animated reenactment of the instructions working their way through the various processor structures. This tool excels in its ability to simultaneously represent multiple levels of detail (*e.g.*, statistics at different granularities), but lacks a summary view of the program's execution. Neither Rivet nor the described tool has been made available in any form.

GPV [6] is a pipeline visualization tool that, like TraceVis, presents a graphical view of instruction flow through the processor. GPV also provides some degree of statistics graphing (specifically, IPC) as well as assembly-level static information. Additionally, GPV contains functionality to superimpose multiple traces on top of one another, enabling contrasting program executions on two different microarchitectures. GPV, however, is not as feature-rich as TraceVis. GPV does not include methods for searching or annotating traces, nor does it allow for correlation back to the assembly file or high-level code.

TraceVis is also considerably faster and more scalable than GPV. There are two main reasons for this distinction. First, whereas TraceVis is written in C++, GPV is written in Perl/Tk. While Perl/Tk arguably makes GPV more cross-platform compatible, it also comes with a performance penalty versus a compiled language like C++. Second, whereas TraceVis reads its trace information from binary-encoded trace files, GPV uses ASCII-based files as its input. While ASCII is stand-alone human-readable, using it as input to a visualization tool requires an extra parsing stage. Furthermore, ASCII files are necessarily larger than a binary-encoded equivalent.

VI. Conclusion

In creating TraceVis, we strove to meet the standards set forth in the paper's introduction. By this measure, TraceVis largely succeeds in its goals. We have created an interactive program for exploring execution traces and the application/microarchitecture interaction at large.

TraceVis achieves easy access to high-level information through whole-trace visualization which allows users to quickly identify regions of interest. Features such as interactive zooming, regional statistics, code correlation and individual instruction information provide on-demand access to increasingly low-levels of detail—all the way down to the raw trace itself.

TraceVis is able to maintain interactiveness, even when viewing large numbers of instructions, as long as the trace fits in main memory. The key technique employed is sampling. Though sampling is imprecise, users are typically interested in qualitative trends when zoomed out, and these trends can effectively be approximated.

Pattern detection is available in TraceVis via the trace graph and also through statistics graphing. The trace graph offers an opportunity to identify regions of similar IPC, or at a lower level, regions with similar processor activity. Statistics graphing offers the opportunity to investigate otherwise non-visible behavior such as event frequency or code composition.

TraceVis makes persistent annotation available via bookmarking and code coloring. Both of these features enable users to add their own information to the trace and to do so in a way that builds a relationship with a trace that spans multiple sessions.

Finally, the construction of TraceVis was kept general enough that the tool can be applied to configurations other than those presented in this paper. Re-configurable parameters such as the number of pipeline stages and observed events as well as the support for visualizing multiple threads of execution make TraceVis a useful tool for analyzing a wide variety of computing environments.

VII. Acknowledgments

This work was supported by NSF CAREER award 434 CCF 03-47260, grants CCR 03-11340 and EIA-0224453, and equipment donations from AMD and Intel.

References

[1] J. Roberts, "TraceVis: An Execution Trace Visualization Tool," `http://www-faculty.cs.uiuc.edu/ zilles/tracevis`.

[2] W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach, "VAMPIR: Visualization and analysis of MPI resources," *Supercomputer*, vol. 12, no. 1, pp. 69–80, 1996.

[3] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "Paraver: A tool to visualize and analyze parallel code," European Center for Parallelism of Barcelona, Tech. Rep. UPC-CEPBA-1995-03, 1995.

[4] M. Reilly and J. Edmondson, "Performance simulation of an alpha microprocessor," *Computer*, vol. 31, no. 5, pp. 50–58, 1998.

[5] D. A. Patterson and J. L. Hennessy, *Computer Organization & Design: The Hardware/Software Interface*, 2nd ed. San Mateo: Morgan Kaufmann, 1998.

[6] C. Weaver, K. Barr, E. Marsman, D. Ernst, and T. Austin, "Performance analysis using pipeline visualization," in *ISPASS*, 2001.

[7] Trolltech, "The Qt Application Development Framework," `http://www.trolltech.com/products/qt/`.

[8] J. Neider, T. Davis, and M. Woo, *OpenGL Programming Guide – The Official Guide to Learning OpenGL, Release 1*, 1993.

[9] T. Austin, E. Larson, and D. Ernst, "Simplescalar: An infrastructure for computer system modeling," vol. 35, no. 2, pp. 59–67, Feb. 2002.

[10] J. Roberts, "Tracevis: An execution trace visualization tool," Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, July 2004.

[11] C. Stolte, R. Bosch, P. Hanrahan, , and M. Rosenblum, "Visualizing application behavior on superscalar processors," in *InfoVis*, 1999.