TraceVis: An Execution Trace Visualization Tool

James Evan Roberts

July 31, 2004

TABLE OF CONTENTS

1 INTRODUCTION

In this thesis we introduce TraceVis, a tool for exploring application behavior as it relates to its execution on a microprocessor. Through a host of features, TraceVis enables users to better understand the interactions among instructions and the hardware.

With the insight gained from this understanding, users will be able to more effectively evaluate engineering decisions such as compilation strategies, architectural designs and microarchitectural designs.

1.1 Motivation

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.

As a result of these factors, program behavior is often unknown or, at best, is known only in terms of broad generalizations. This type of analysis is typified by summarizing data into numerical statistics (*e.g.*, average fetch time, IPC, hit rates, etc.).

Summary results are problematic for a variety of reasons. First, summary results tend to average multiple behaviors into one. For this reason, summary results are prone to ignoring phases in behavior. Second, summary results do very little to answer the question of *why* a particular measure of behavior was observed. This limitation arises from the fact that statistics are very specific to what they are measuring; that is, they do not include any peripheral information. Lastly, summary results are a very poor means by which to perform a general exploration of code behavior; they offer only a very narrow view of program behavior, and unless an analyst makes an active effort to collect statistics on a particular phenomena, that behavior can easily be overlooked.

The standard approach for dealing with the drawbacks of summary data is to collect *more* summary data. One way of doing this is through an iterative "trial and error" process where the researcher makes a guess as to the cause of a statistical result and then collects statistics to support that theory. The iterative process is problematic in that it requires the analyst to track all of the possible influencing factors for each observed result. Furthermore, with each iteration, the researcher incurs the costs associated with development and simulation time. Alternatively, researchers can simply set up their simulator to collect a tremendous amount of data in the hope of collecting data on every pertinent behavior. This approach, however, can quickly lead to an overload of information which the user is simply unable to comprehend in an efficient manner.

1.2 Goals

Our solution to the obstacles of execution analysis is to present data in a format that is more conducive to inspection than text-based numeric results. Specifically, our solution is to present data in a graphical format. By visualizing application behavior we can effectively translate huge quantities of numeric data into single images. This approach recognizes humans' innate ability to efficiently process tremendous amounts of visual information and to identify patterns and anomalies in that information.

Our stated goals are as follows:

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. **Searchability.** 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.** To as large an extent as possible, users should not have to read text-based or numeric results.

The remainder of this thesis proceeds as follows: Chapter 2 introduces the features of TraceVis. Chapter 3 goes through a detailed use-case scenario in which we use Trace-Vis to analyze an execution trace. Chapter 5 presents related work. Chapter 6 discusses opportunities for future work. Lastly, chapter 7 gives a conclusion.

2 FEATURES

In this chapter 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.

2.1 Basic Trace Graphing

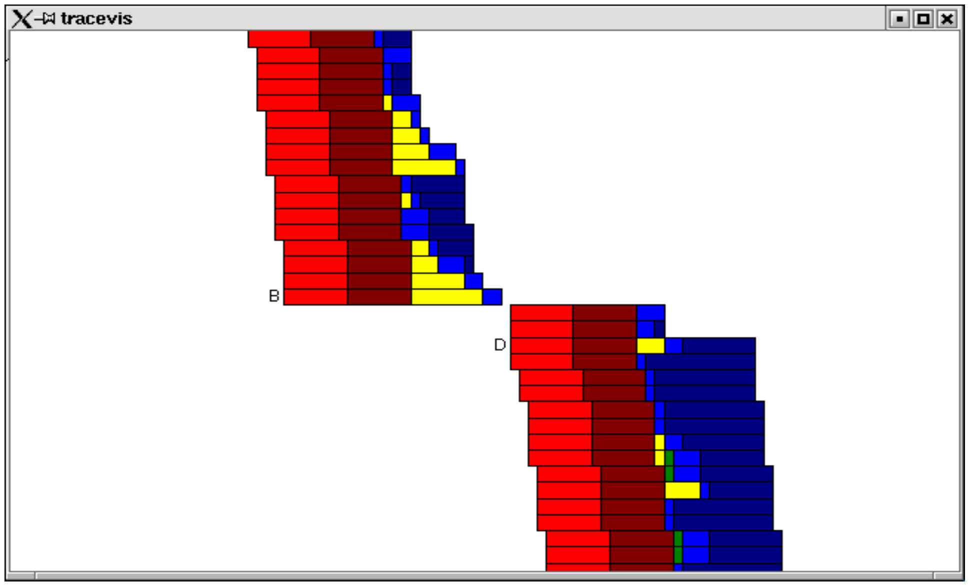


Figure 2.1: The basic trace graphing functionality of TraceVis. Instructions are represented as horizontal bars. Instructions are subdivided into color-coded regions each of which represents a pipeline stage. The 'B' and 'D' denote a branch misprediction and a data cache miss, respectively, for the instruction to its right.

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 2.1 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 graph out bad-path execution arising from branch misprediction or other mispeculated events.

This graphical representation of instructions flowing through the processor pipeline is straightforward and is similar to the notation used in architecture texts [1]. Using 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.

The annotations to the left of the instruction bars denote events that occurred in the course of that instruction's lifetime. Our current model tracks the following events: A branch mispredict is represented by a 'B', an instruction cache miss by 'I', a data cache miss by 'D', and a load/store ordering violation pair by 'L'/'S'.

2D-Navigation about the trace is either via 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.

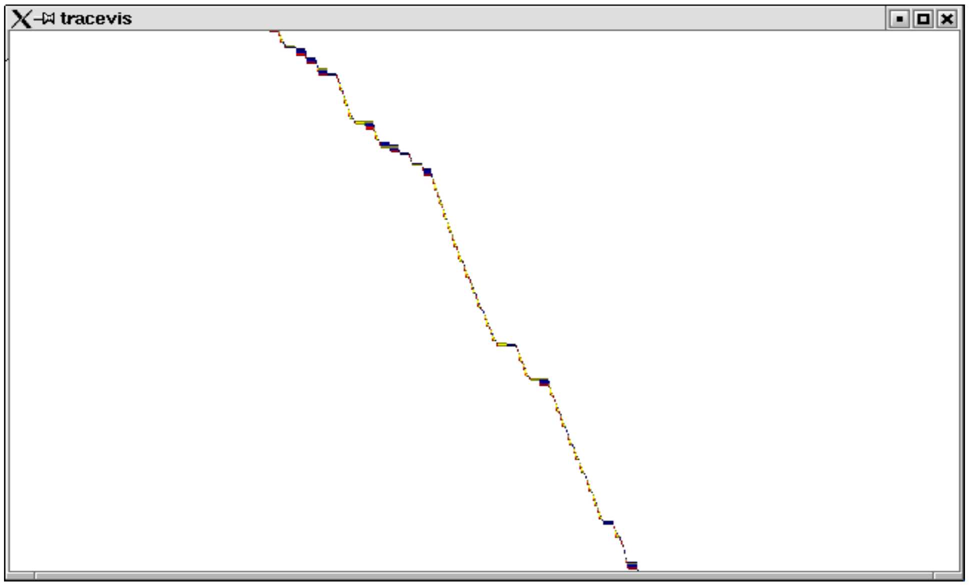


Figure 2.2: TraceVis zoomed-out. TraceVis is capable of rendering at arbitrary levels of zoom. Here around 15k instructions are represented by the rendered image.

TraceVis is also capable of arbitrary levels of zoom. Figure 2.2 shows the same trace from Figure 2.1 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 sections 2.6 and 2.7 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 scanline). 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 subsampling of the trace data is largely imperceptable at these low levels of zoom.

2.2 Bookmarking

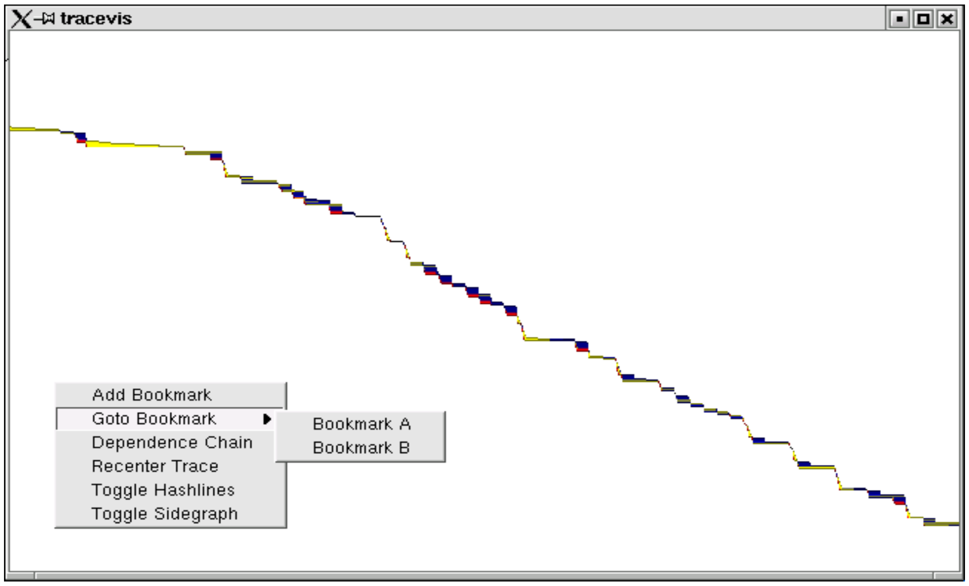
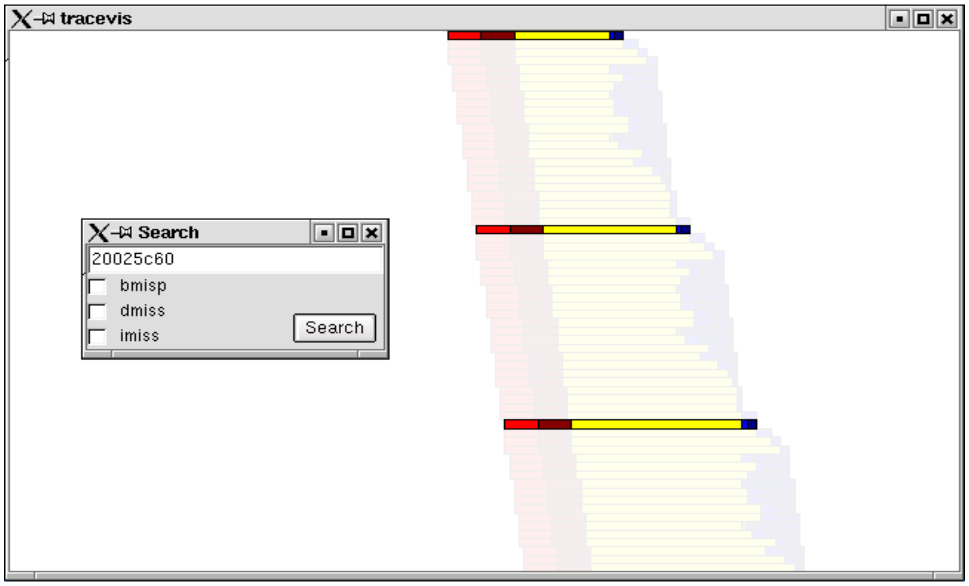


Figure 2.3: Bookmarks in TraceVis. Bookmarks are user-specified tags on specific trace locations. Users can then return to those locations at any time by selecting the bookmark from a list.

TraceVis includes functionality to bookmark specific locations in a trace for later reference. Figure 2.3 shows a sample representation of the bookmarking feature. To create a bookmark, the user specifies a location in the trace and a label for that location. When that label is later selected from the bookmark pull-down menu, the trace relocates to the corresponding location. Bookmarks are saved to disk and are associated with that specific trace so that they remain available whenever that trace is loaded.

2.3 Searching

TraceVis includes a mechanism for searching a trace for a specific instruction address, event (*e.g.*, branch mispredict, cache miss, etc.) or any combination of the two. Figure 2.4 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.



20025c60

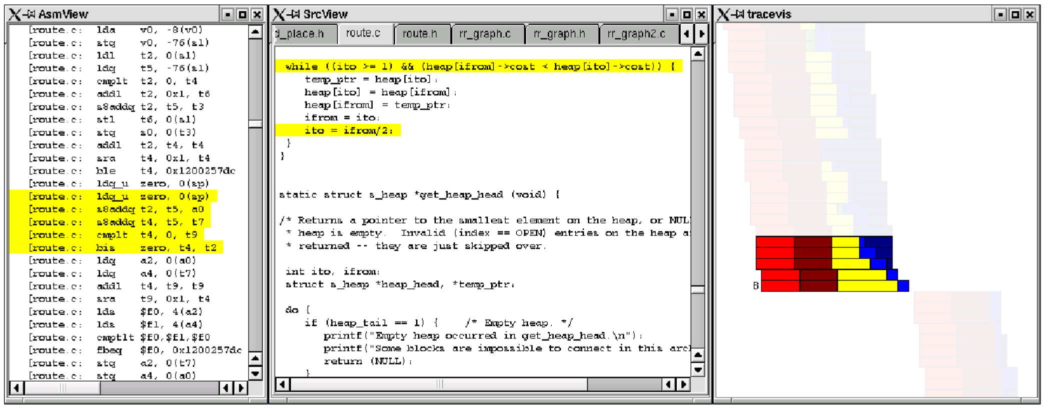
☐ bmisp
☐ dmiss
☐ imiss

Figure 2.4: TraceVis searching for matching instructions. The first matching instruction is moved to the top. Non-matching instructions are grayed-out.

2.4 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 the dynamic trace) is also possible. Figure 2.5 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.



```
[route.c:  lda     v0, -8(v0)
[route.c:  stq     v0, -76(s1)
[route.c:  ldl     t2, 0(s1)
[route.c:  ldq     t5, -76(s1)
[route.c:  cmplt   t2, 0, t4
[route.c:  addl    t2, 0x1, t6
[route.c:  s8addq  t2, t5, t3
[route.c:  stl     t6, 0(s1)
[route.c:  stq     s0, 0(t3)
[route.c:  addl    t2, t4, t4
[route.c:  sra     t4, 0x1, t4
[route.c:  ble     t4, 0x1200257dc
[route.c:  ldq_u   zero, 0(sp)
[route.c:  ldq_u   zero, 0(sp)
[route.c:  s8addq  t2, t5, a0
[route.c:  s8addq  t4, t5, t7
[route.c:  cmplt   t4, 0, t9
[route.c:  bis     zero, t4, t2
[route.c:  ldq     a2, 0(a0)
[route.c:  ldq     a4, 0(t7)
[route.c:  addl    t4, t9, t9
[route.c:  sra     t9, 0x1, t4
[route.c:  lds     $f0, 4(a2)
[route.c:  lds     $f1, 4(a4)
[route.c:  cmptlt  $f0,$f1,$f0
[route.c:  fbeq    $f0, 0x1200257dc
[route.c:  stq     a2, 0(t7)
[route.c:  stq     a4, 0(a0)
```

```
 while ((ito >= 1) && (heap[ifrom]->cost < heap[ito]->cost)) {
    temp_ptr = heap[ito];
    heap[ito] = heap[ifrom];
    heap[ifrom] = temp_ptr;
    ifrom = ito;
    ito = ifrom/2;
 }
}


static struct s_heap *get_heap_head (void) {

/* Returns a pointer to the smallest element on the heap, or NUL
 * heap is empty.  Invalid (index == OPEN) entries on the heap a
 * returned -- they are just skipped over.

 int ito, ifrom;
 struct s_heap *heap_head, *temp_ptr;

 do {
    if (heap_tail == 1) {    /* Empty heap. */
       printf("Empty heap occurred in get_heap_head.\n");
       printf("Some blocks are impossible to connect in this arc
       return (NULL);
    }
```

Figure 2.5: Dynamic-to-static code correlation. Highlighted assembly, C source code, and dynamic instructions all match to the same set of static instruction addresses.

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.

An interesting point about the static-to-dynamic correlation is that there is often no dynamic instructions to match a given static instruction. That is, even with a trace as long as 10M instructions, the working set of instructions is often such that only a small fraction of the static code is executed. This stands as anecdotal evidence to the notion that most of the execution time is spent in a very narrow region of the code.

2.5 Static Code Coloring

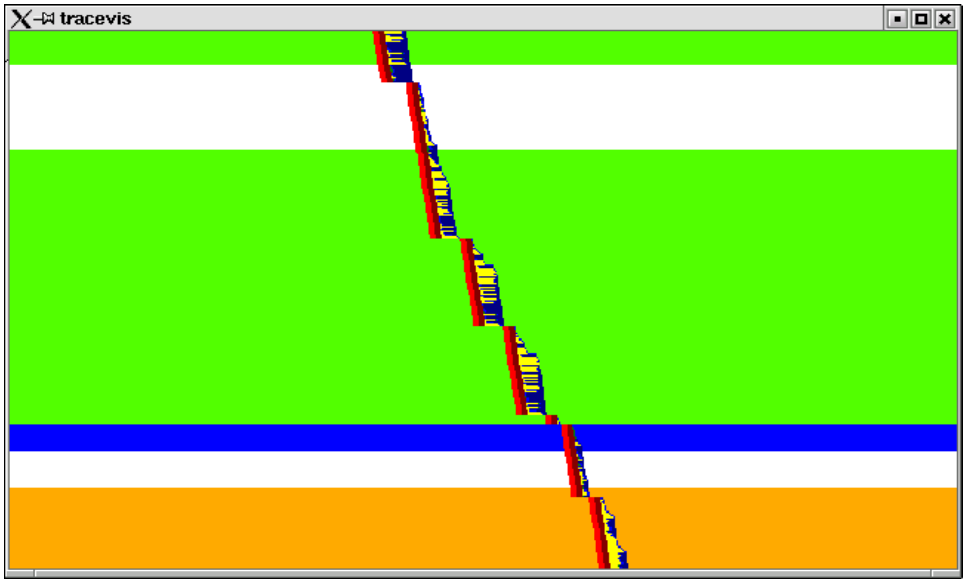


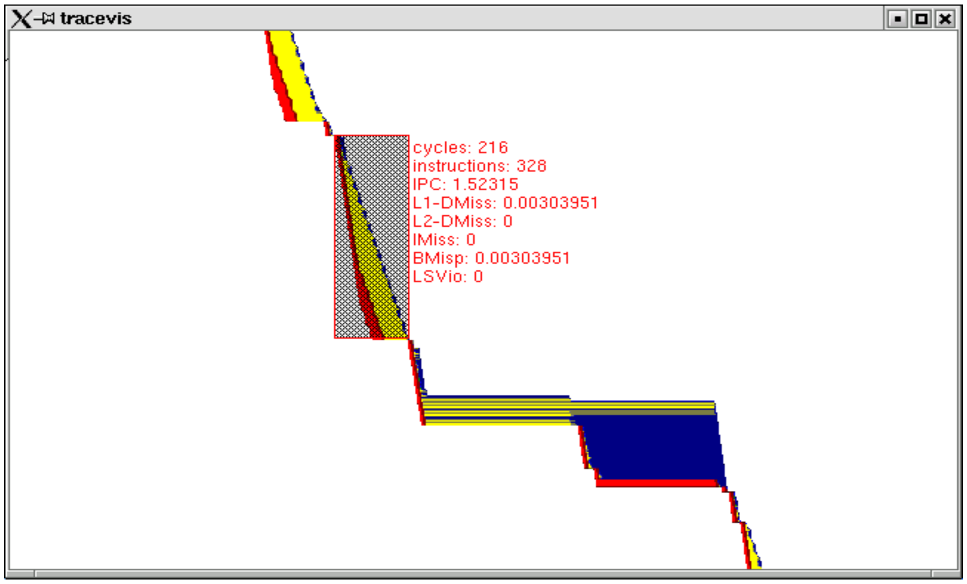
Figure 2.6: Static code coloring. Instructions within a colored region share a common set of instruction address. Uncolored regions contain instructions with addresses that are not in any of the designated sets.

Figure 2.6 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 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 be 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 section 2.7 we will discuss how code coloring can be used in conjunction with statistics graphing for the purpose of tracking program phases.

2.6 Region Statistics



```
cycles: 216
instructions: 328
IPC: 1.52315
L1-DMiss: 0.00303951
L2-DMiss: 0
IMiss: 0
BMisp: 0.00303951
LSVio: 0
```

Figure 2.7: Regional Statistics. The statistics pertain to the selected region. The figures shown are fractions, not percentages.

Figure 2.7 shows the regional statistics gathering feature of TraceVis. The region statistics feature collects a number of metrics for a selected region of the dynamic trace. Statistics collected include IPC, branch misprediction rate, cache miss rate and load/store ordering violation rate. All these metrics (with the exception of IPC) are in terms of *events/instruction*. So, for example, cache miss rate is in terms of *misses/instruction* rather than *misses/memory_accesses*. This metric is an artifact of the current trace file format which contains flags for events, but does not specify the instruction types or opcodes. However, for the qualitative analysis for which this tool was designed, this methodology appears sufficient.

2.7 Statistics Graphing

Figure 2.8 shows the statistics graphing functionality of TraceVis. In this image, TraceVis is graphing the relative frequency of L2 cache misses as a histogram to the left of the trace. Each line of the histogram corresponds to the average frequency of L2 cache misses for all instructions represented on that scanline. 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 2.1) are no longer feasible.

TraceVis is currently capable of graphing statistics on metrics including branch mispredictions, cache misses, load/store ordering violations and trace composition with respect to colored static regions. Figure 2.9 shows TraceVis graphing trace composition statistics. For

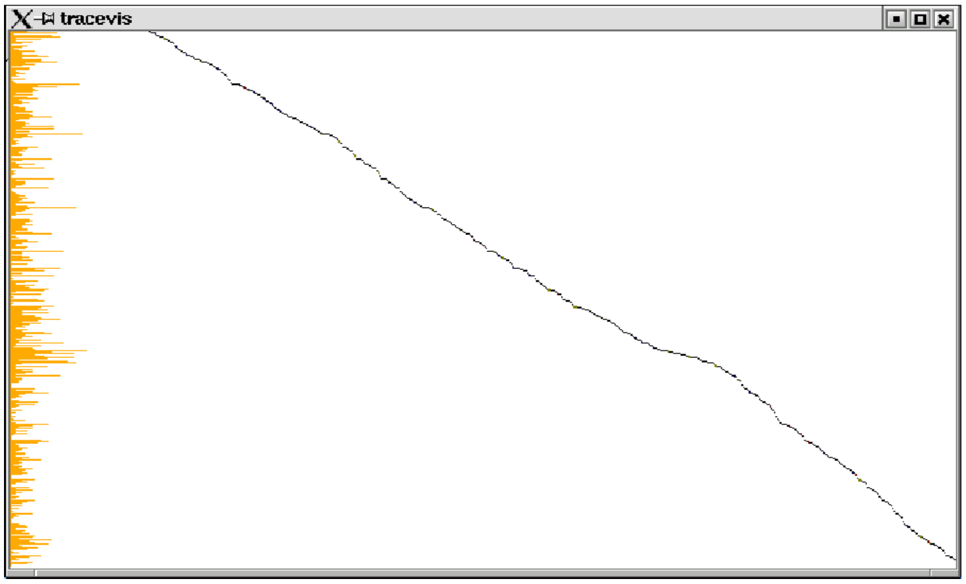


Figure 2.8: Visualization of the L2 cache miss rate. As with regional statistics, the figures are with respect to all instructions, not just memory accesses.

each scanline, the corresponding line on the histogram shows the relative makeup of all the instructions represented by that scanline (in terms of the colored regions of static code). For example, a histogram line that shows equal lengths of orange and blue implies that the instructions represented on that scanline are composed of an equal amount of orange- and blue-colored static instructions.

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 this representation is sufficient for the purposes of this tool. In chapter 6, we discuss methods for improving the fidelity of the statistics graphing feature.

2.8 Dependence Graphing

Figure 2.10 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 is capped by a user-definable setting.

Currently, due to our trace file format, TraceVis graphs only register-based dependences. Memory-based dependences could also be added to the visualization if that information were added to the trace.

2.9 MSSP Mode (and generalized multi-core traces)

TraceVis includes special functionality for visualizing MSSP [2] execution traces. Figure 2.11 shows the MSSP visualization mode of TraceVis. Here the screen is split into two separate

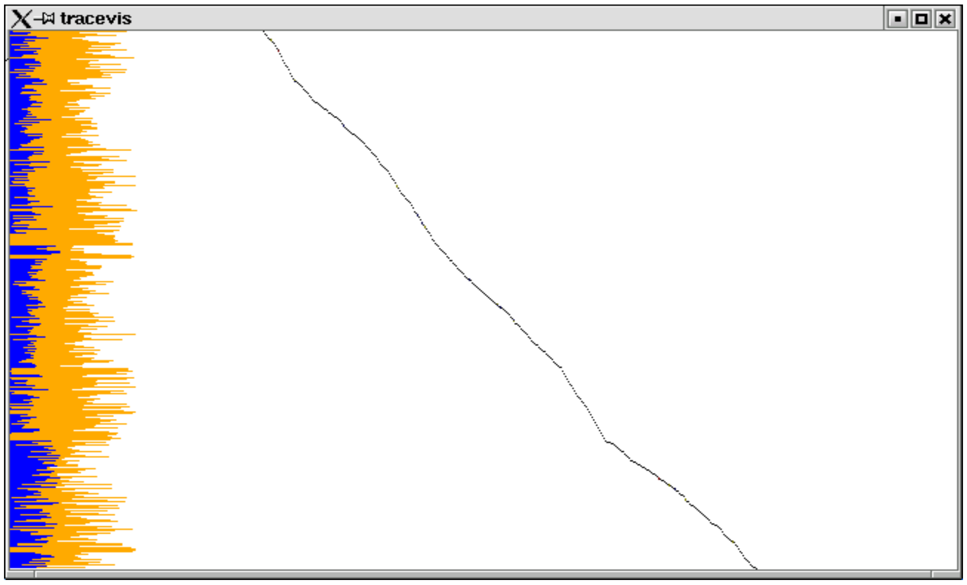


Figure 2.9: A visualization of code composition with respect to colored static instructions. Colored bars in the histogram reflect the frequency of instructions from the correspondingly colored regions of static code.

views: on top is the master thread and beneath is all of the slave threads interleaved together in proper program order.

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. Section 4.2 provides a detailed description of the algorithm that is used for synchronizing the views. In brief, the algorithm insures that the views stay synchronized with respect to time (so that a vertical slice of the trace always reveals the global state of the processor for that given point in time) and automatically adjusts the y-offsets such that the trace body remains within the viewing area.

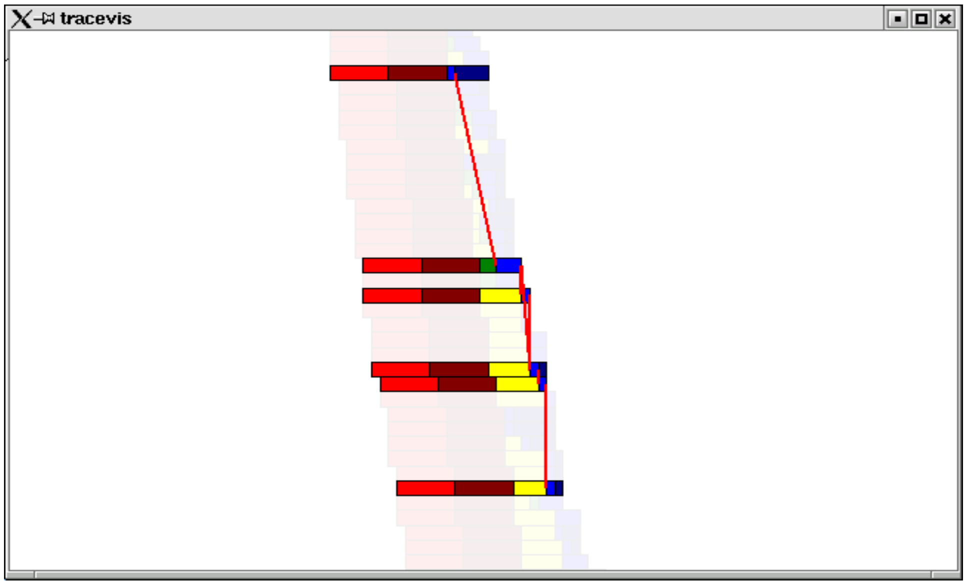


Figure 2.10: Dependence Chain Visualization. The lines, drawn in red to be more visible, represent the data dependence chain originating from the bottom-most instruction.

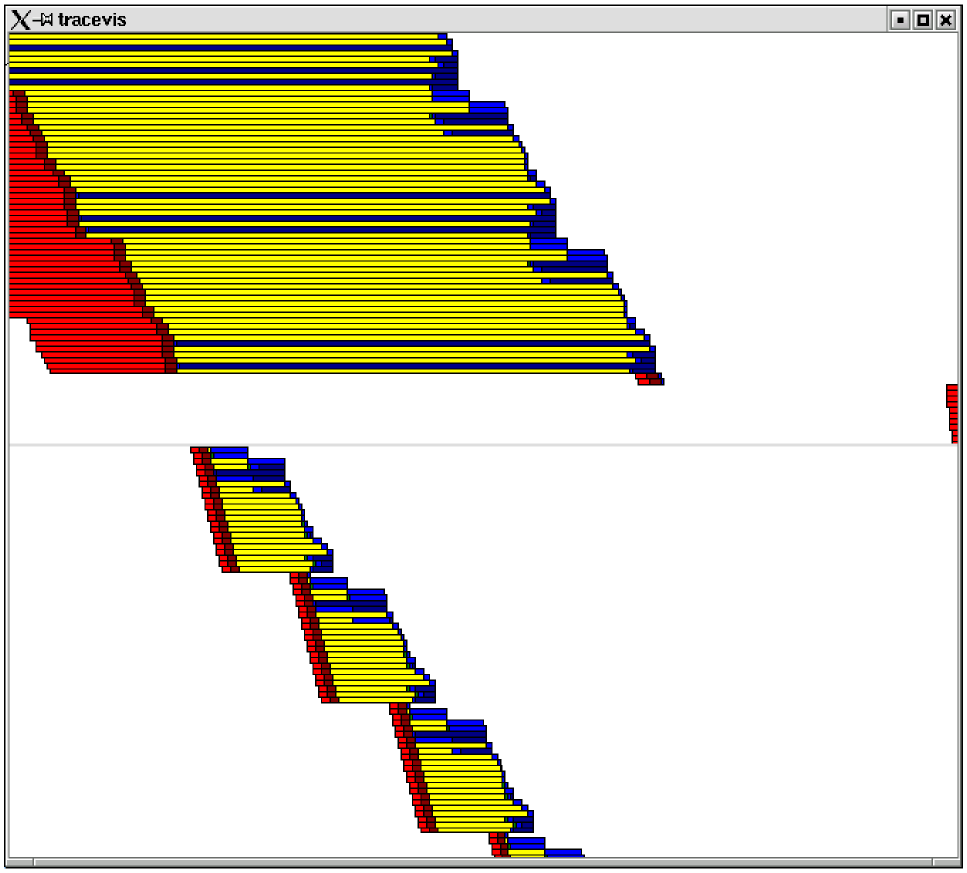


Figure 2.11: Visualization of an MSSP execution trace. The top frame shows the master trace. The bottom frame shows all of the slaves traces interleaved together in program order.

3 USE CASES

In this chapter we present a sample usage of TraceVis. We utilize the various features to analyze a typical execution trace. We start at a high level and then work down to progressively lower levels of detail.

3.1 Full-Trace Visualization

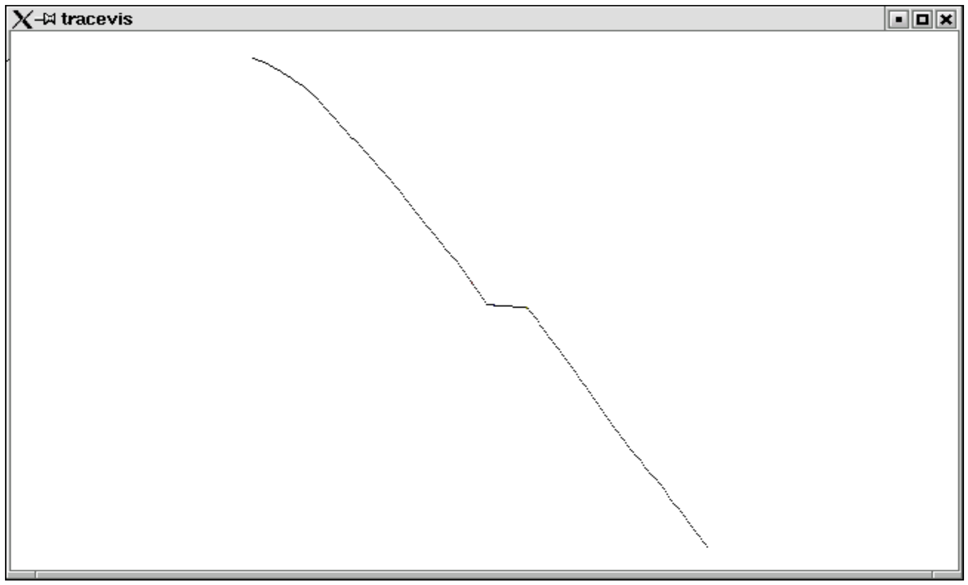


Figure 3.1: Visualization of a 10M instruction-long trace of **`vpr`**. This trace was capured several billion instructions into the benchmark's lifetime.

Figure 3.1 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 lifetime. 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 scanline. With 10M instructions and roughly 500 scanlines, it follows that each scanline 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.

3.2 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 due to warm-up of the simulated machine. That is, the performance penalties paid in this area are not due any to special start-up code, but rather they are due to training of processor structures such as the branch predictor and the cache (via 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 mispredict rate, cache miss rate, etc.) occur at higher rates in the initial phase than in other regions executing the same static code.

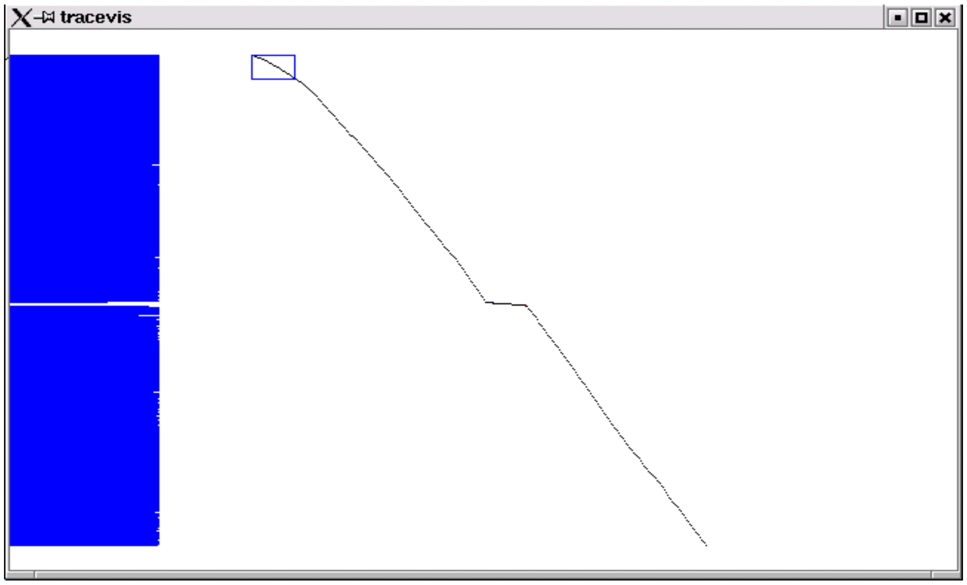
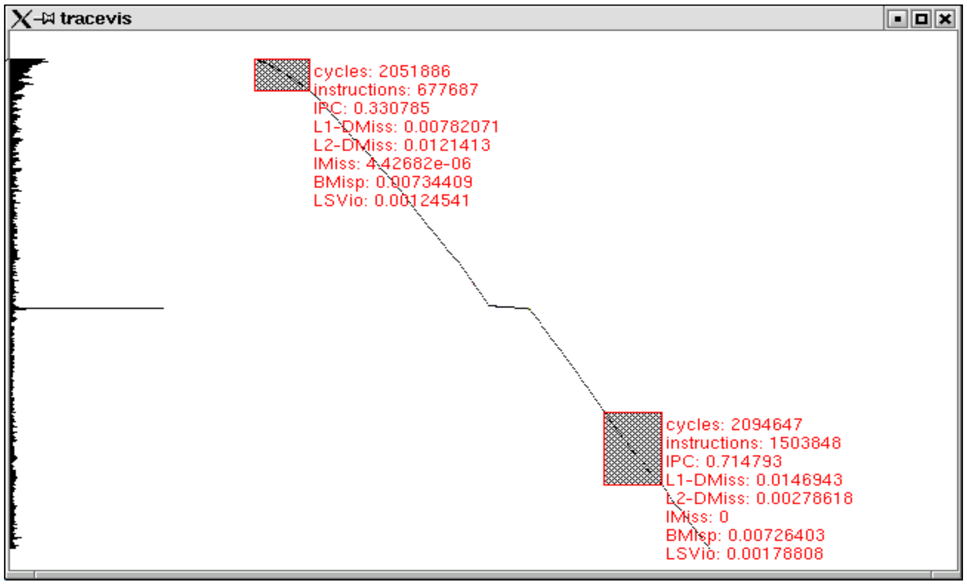


Figure 3.2: Visualization of trace composition with respect to the boxed region in the trace's top left. The nearly solid histogram on the left shows that most of the trace uses the same instructions as the *warm-up* phase of the trace.

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 3.2 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.



```
cycles: 2051886
instructions: 677687
IPC: 0.330785
L1-DMiss: 0.00782071
L2-DMiss: 0.0121413
IMiss: 4.42682e-06
BMisp: 0.00734409
LSVio: 0.00124541
```

```
cycles: 2094647
instructions: 1503848
IPC: 0.714793
L1-DMiss: 0.0146943
L2-DMiss: 0.00278618
IMiss: 0
BMisp: 0.00726403
LSVio: 0.00178808
```

Figure 3.3: 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.

Now we show that the low IPC of the warm-up phase can be attributed to warm-up-related events. The TraceVis screenshot in figure 3.3 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.

3.3 Mid-trace Plateau

Next we investigate the shallow region at the trace's center. As we saw in figure 3.2, the region at the center of the vpr trace executes a set of static code that is different from the rest of the trace.

Figure 3.4 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.

We investigate the top phase of the center region in figure 3.5. Here we see that the phase is dominated by two pipeline stages: fetch and *waiting-to-retire*. Since we model an in-order retire machine, we can reason that the long fetch latencies are caused by instruction window backpressure created by the long *waiting-to-retire* latencies. Using the instruction details information, TraceVis reveals the the long *waiting-to-retire* latencies are themselves caused by L2 D-cache misses.

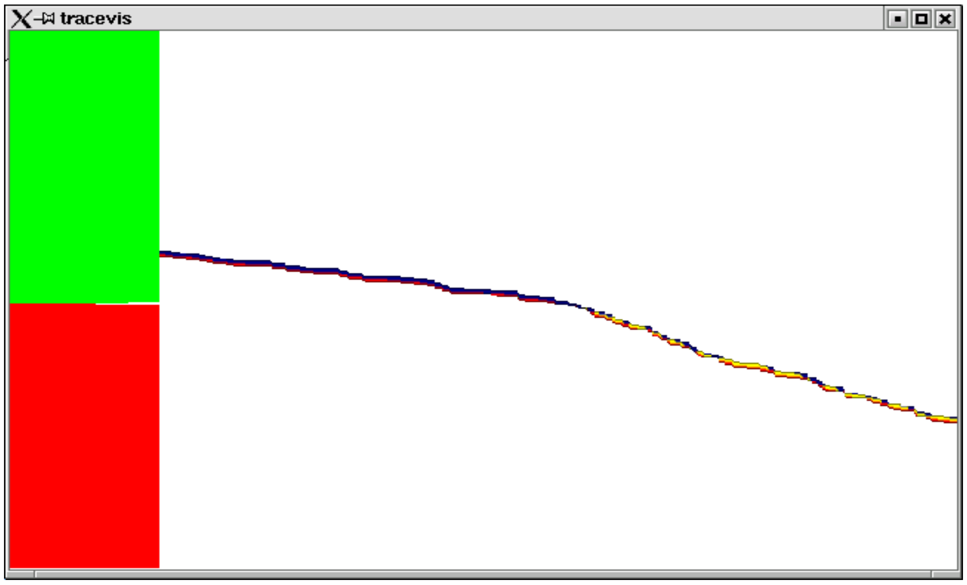


Figure 3.4: A detailed view of middle region. Roughly 6k instructions are represented in this image. The histogram to the left shows the composition of the trace in terms of static instructions. The histogram shows that the middle region is composed of two distinct phases – each one executing a different set of code.

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 was undetected up to this point.

Using the same static-to-dynamic code mapping mechanism we can also find the corresponding high-level source code. Figure 3.6 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.

Now we investigate the remaining portion of the trace's middle region. Figure 3.7 shows a close-up rendering of the phase. Compared to the behavior shown of the previous phase (shown in Figure 3.5), it becomes clear that this area is dominated not only by instruction window fills, but also by *waiting-for-operands* constraints.

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.

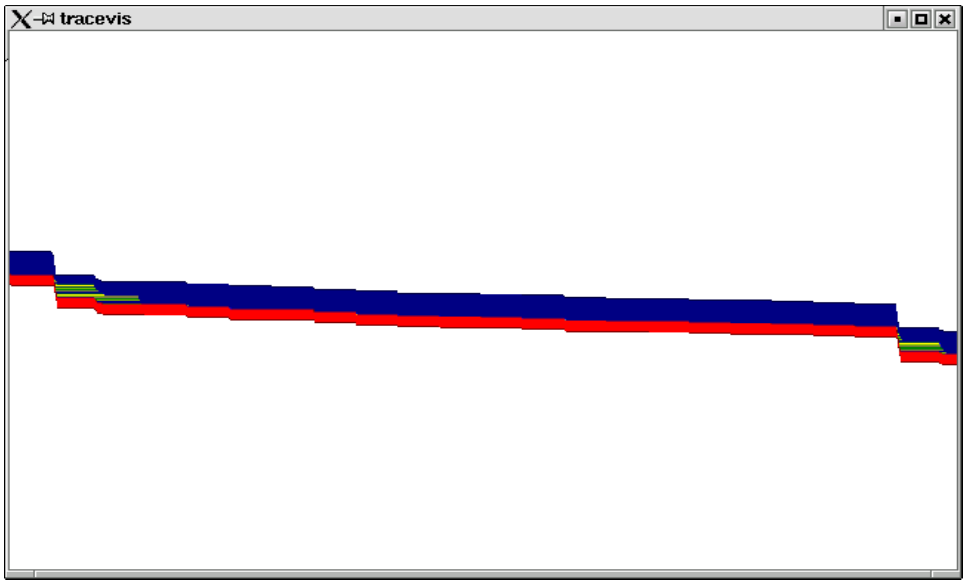
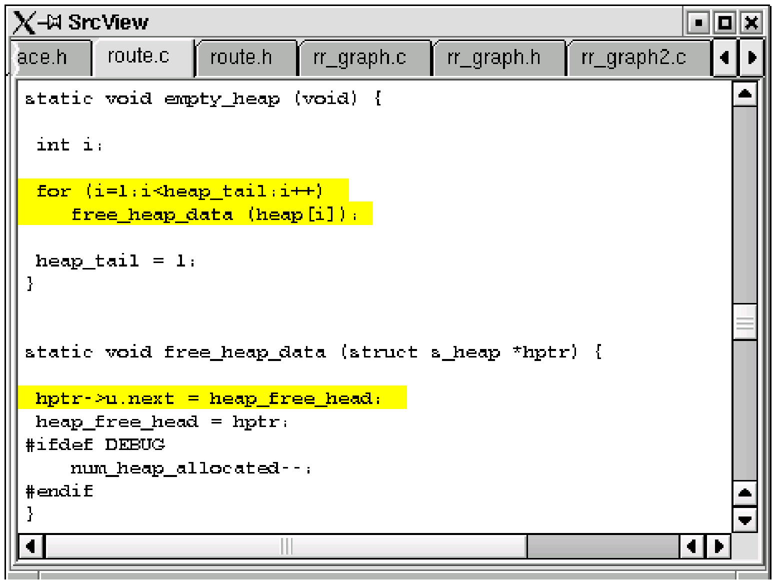


Figure 3.5: The top portion of the middle region. The data in this graph suggests that most of the processor time is spent in fetch stalls and *waiting-to-retire* stalls. In some cases instructions are in the pipeline for as long as 10,000 cycles. Intuition suggests that the root cause of the stalls is store misses; source code correlation backs up this theory.

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.

3.4 Summary

The findings in this chapter 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.



```
static void empty_heap (void) {

 int i;

 for (i=1;i<heap_tail;i++)
    free_heap_data (heap[i]);

 heap_tail = 1;
}


static void free_heap_data (struct s_heap *hptr) {

 hptr->u.next = heap_free_head;
 heap_free_head = hptr;
#ifdef DEBUG
    num_heap_allocated--;
#endif
}
```

Figure 3.6: Source code for the code region of figure 3.5. Re-initializing all of the *hptr* data structure would account for the large number of store misses visible in the trace graph.

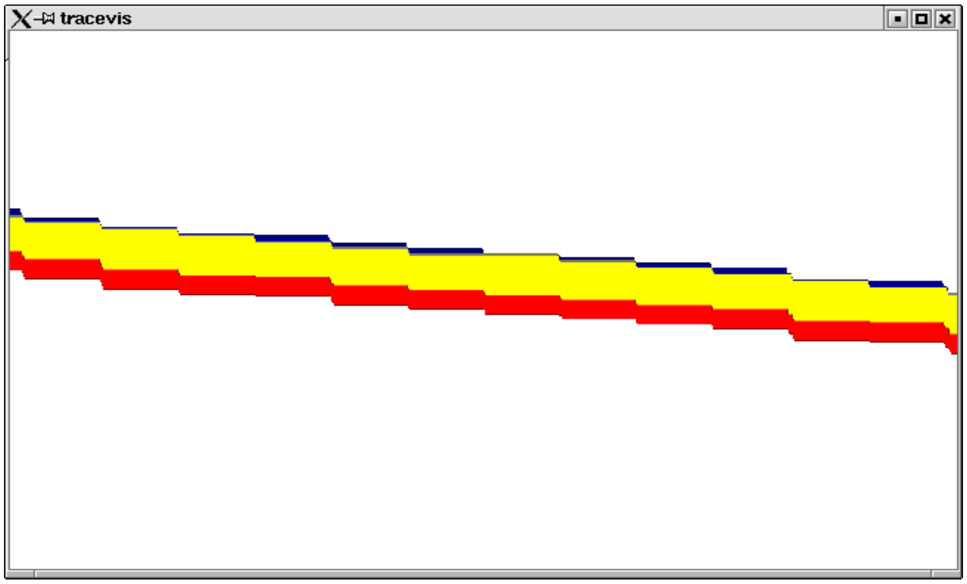


Figure 3.7: The bottom portion of the middle region. The data in this graph suggests that most of the processor time is spent in *waiting-for-operand* stalls. The stalls are the result of load instructions missing in the L2 cache.

4 IMPLEMENTATION

In this chapter 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.

4.1 Language and Windowing Toolkit

TraceVis was written in C++ using the Qt [3] 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 be platform compatible.

TraceVis does not make use of OpenGL [4] 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.

4.2 Multi-View Synchronization

As mentioned in section 2.9, synchronized viewing of multi-threaded traces is a non-trivial task. In this section we discuss TraceVis's mechanism for dealing with this problem.

Ideally, in a multi-thread visualization (such as the one in figure 2.11) we would like to be able to traverse the trace of any one view and have all the other views mirror those adjustments. Specifically, we would like the views to remain synchronized with respect to time and to automatically adjust their y-offsets such that their respective trace bodies remains centered in their viewing areas. Unlike translations in along the x-axis, however, adjustments to y-offsets potentially have no correlation across the views. In fact, due to varying rates of IPC among the threads (and therefore varying slopes of their graphs), y-offsets are potentially unique for each view.

The major hurdle to automatic adjustment of y-offsets is that our data structures are not amenable to this type of functionality. That is, the problem would be easily solved if TraceVis had a table of instruction identifiers indexed by cycle number; then for a given point along the x-axis we could simply lookup the instructions corresponding to that point in time and adjust the y-offset accordingly. However, due to the format of our trace file, we have the inverse situation: a table of cycle times indexed via instruction identifiers. Constructing the cycle-indexed table from out instruction-indexed one is not a feasible option due to the space it would require. So, to find the instructions that correspond to a given point in time, we must search the trace data for a matching timestamp.

We can simplify the process of searching the trace data using the following assumptions: a) The trace proceeds monotonically from the upper-left to the lower-right,[1] and b) the trace body can be modeled as a volume bounded by two non-intersecting lines – that is, the lines delineated by fetch times and retire times. Using these assumptions we can make the following assertions: For any given point in space, (x,y), if the instruction at that y-value retires prior to (*i.e.*, to the left of) the x-value, then the point is *above* the trace body. Similarly, if the instruction at the y-value fetches subsequent to (*i.e.*, to the right of) the x-value, the point is *below* the trace body.

Using these assertions, we are able perform view synchronization as follows: First the view is translated along the x-axis by the same amount that the other views were translated. Then, taking the point at the center of the viewing area, (x,y), we determine both the cycle number for that point and the fetch and retire times for the instruction that corresponds to the y-value of that center point. Using these three time values we then apply our assertions to determine if the center point is *above*, *below* or *centered on* the trace body. If the center point is above the trace, we translate the viewing window down by a small increment and then repeat the process. Similarly, if the center point is below the trace, we translate the view upwards and repeat the process. The algorithm terminates when the trace is centered on the screen or when the view has been translated to the top or bottom extents of the trace.

[1]MSSP-based slave traces potentially violate this assumption. That is, when slave traces are interleaved together in program order the resulting trace may contain out-of-order fetches and retires. However, since the overall trend of the trace is from top-left to bottom-right, the auto-y-adjustment algorithm is still able to function correctly.

5 RELATED WORK

In this chapter, we describe a number of previous pipeline visualization tools. TraceVis compares favorably with these tools in terms of features, performance and/or availability.

GPV [5] 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. This feature allows users to contrast the executions of two different traces (which presumably perform the same task, but execute code that was produced by different compilation strategies or was being executed on different microarchitectures).

GPV, however, is not as feature-rich as TraceVis. GPV does not include methods for searching the trace or adding annotations to the trace (via bookmarks or static code coloring), 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-encode 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.

Stolte, et al. [6] present an execution visualizing tool for aiding in application-level optimizations. The Stolte 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 on the screen. Navigation, for instance, is via manipulation of bounding boxes in a series of increasingly detailed timeline views – each one zooming in on the bounded region of the view above it. The source code view is also able to simultaneously represent multiple levels of detail.

What the Stolte tool lacks is a summary view of instruction flow through the pipeline. The animations which substitute for the summary view, while perhaps valuable from a pedagogical standpoint, are arguably too detailed for routine trace analysis. A summary view, such as TraceVis's or GPV's trace graph, effectively compresses the information of many animations into a single, static frame. A final drawback to the Stolte tool is that it is not publicly available.

Finally, we suspect that most commercial microprocessor development teams possess in-house visualization tools which they use in their analysis and design of microarchitectures.

In [7] Reilly, et al. offer a screenshot of a pipeline visualization tool used by Alpha engineers for visualizing Alpha 21264 executions.

While the article stops short of actually discussing the tool, from the screenshot we can deduce several key features. The Alpha tool presents a graphical view of instruction flow much like TraceVis. The tool also includes a graph of the processor state over time. This processor state graph essentially offers the same information as the Stolte tool's animations, but does so in a static, summarized format. The Alpha tool also includes the ability to graph mispeculated instructions.

Like the Stolte tool, the Alpha tool is not publicly available. For this reason, and because the tool is not heavily discussed in the article, it is difficult to speculate about its usability and/or scalability.

6 FUTURE WORK

We believe that there remain many ways to improve TraceVis with regards to its features, its usability and its performance. In some instances these enhancements were inspired by those found in the related work. Below we highlight a few of these improvements.

Simultaneous multiple levels-of-detail views. Currently, it is quite easy for users to lose track of their overall location while traversing a trace graph; this is especially true when the user is working on the trace while zoomed-in to a high level of detail

We would like to expand TraceVis so that, when zoomed-in, the interface provides visual cues regarding the user's overall location. The Stolte tool provides a very intuitive mechanism for this functionality: it displays multiple levels of detail on-screen simultaneously, and draws bounding boxes in the views to represent the portions of the trace that are visible in the next lowest level of detail view. We would like TraceVis to provide this functionality for both the trace view and the source code views.

Visualize data that is currently text-based. Some text-based interactions could be converted to be graphics-based. Regional statistics, for instance, could be converted from text-based numerics to color-coded bar graphs.

Improvements in coarse granularity visualization. Visual representation of the trace at levels of coarse granularity is a feature that has not been fully explored. Currently, when zoomed out beyond the point of one instruction per scanline, TraceVis does a rudimentary blend of the top-most and bottom-most instructions falling on that line. A more sophisticated algorithm would more closely approach true anti-aliasing while maintaining interactive frame rates.

Statistics graphing could also be improved. Currently, statistics for the side-graph are collected on a per scanline basis via random sampling. The sampling rate is constrained by what is feasible at interactive frame rates. To improve the fidelity of the graphed statistics, TraceVis could employ an iterative sampling method that would give rough estimates at first, but would become increasingly accurate if the screen were left idle. Additionally, statistics for instructions outside of the current frame could be calculated in the background and then be immediately available when the trace is translated up or down. Lastly, the statistics mechanism could utilize some type of mipmap-like functionality [8] to allow statistics to be transfered from one zoom-level to the next rather than simply re-calculating the results at every adjustment.

Expanded dependence graphing capabilities. Currently, TraceVis graphs only register-based dependences. However, many more dependences exist in the system, and we would like TraceVis to visual more of them. For example, section 2.8 alluded to the feasibility of adding memory-based dependences to the visualization. Furthermore, TraceVis could also graph hardware-based and control dependences.

Improved flexibility Work is already being done that will allow TraceVis to read a variety of trace formats. Also, parameters such as the number of pipeline stages and the variety of events are being generalized so that they will become user-modifiable via a configuration file.

7 CONCLUSION

In creating TraceVis, we strove to meet the standards set forth in section 1.2. 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 (on the order of 10M), by using a sampling methodology. 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.

REFERENCES

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

[2] C. Zilles, “Master/slave speculative parallelization and approximate code,” Ph.D. dissertation, Aug. 2002.

[3] Trolltech, “The Qt Application Development Framework,” `http://www.trolltech.com/products/qt/`.

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

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

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

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

[8] L. Williams, “Pyramidal parametrics,” in *SIGGRAPH*, 1983.