dptv: A New PipeTrace Viewer for Microarchitectural Analysis

Adam Grunwald
Department of Computer Science
University of Wisconsin-La Crosse
grunwald5629@uwlax.edu

Phuong Nguyen[†]
FPT Software
Hanoi, Vietnam

Elliott Forbes
Department of Computer Science
University of Wisconsin-La Crosse
eforbes@uwlax.edu

Abstract

In computer architecture research, it is common to test the effectiveness of new microarchitectural features by modeling hardware structures and logic in a software simulator. The benefit of simulation is that those features can be implemented with configurable parameters (sizes, widths, latencies, etc.) that an architect can vary to assess performance across a suite of benchmarks. Typically the simulator gives overall summary statistics for how each benchmark ran, which can be compared to the output of other configurations. But summary statistics are usually averages across entire benchmarks, which can hide key fine-grained details about a configuration's run-time behavior. Furthermore, summary statistics may make it difficult to find exactly where performance diverges when comparing the same code region of a benchmark across two processor configurations.

Pipeline tracing tools, for example gem5's O3 Pipeline Viewer [2], help to visualize instruction-level behavior. These tools require that a simulation not only provides summary statistics, but also a record of all processor events for all dynamic instructions for each cycle. Those events are written to a trace file, which can then be opened by a visualization tool that plots the trace, with each dynamic instruction of a benchmark along the negative y-axis (program order), and pipeline stage events for each cycle on the positive x-axis. Pipeline tracing tools greatly increase the visibility into the run-time behavior of a benchmark for a given processor configuration, but still lack the ability to compare multiple configurations.

This paper presents a new pipeline trace viewer, called Dual PipeTrace Viewer, or dptv for short. This program can visualize a single trace file, like previous trace viewers. But dptv can also take two trace files as input. dptv uses the SDL2 graphics library to provide a more flexible graphical interface, allowing for for easy panning and zooming on the diagram, with additional functionality to search the trace and to peek at specific stage information. When opening two trace files, dptv verifies both trace files represent the same dynamic instruction stream, and then overlays the pipeline stage events for both traces. In this way, dptv makes it more obvious where two processor configurations have performance that diverges.

This paper will outline the specifics of how dptv functions and provide examples of where analysis is made easier using this tool.

[†]Author contributed to this paper while at University of Wisconsin-La Crosse.

1. Introduction

The Dual PipeTrace Viewer (stylized as *dptv*) is a graphical tool for visualizing traces generated by a processor simulator. Traces are a record of the dynamic instruction stream of a benchmark program being executed by a processor simulator. Tools of this nature visualize the trace as a plot with the dynamic instructions along the y-axis and each cycle along the x-axis, making each row in the plot contain processor events for one dynamic instruction. This style of visualization greatly aids a computer architect – trends and patterns can more obviously attract our eye. The visual format of a pipetrace viewer comes naturally, as most undergraduate computer architecture courses use pipeline timing diagrams [11] to visualize the instruction-level parallelism achieved by a pipelined processor execution model.

As microarchitectural complexity has increased, and workloads have grown in scope, architects have relied more heavily on ever-more sophisticated simulators when looking for performance bottlenecks, and opportunities for improvements to systems. However, the effort required to analyze performance bottlenecks have become highly nuanced. Interactions between instructions and microarchitectural structures cause subtle performance differences that can potentially lure a computer architect to iteratively add debugging code to a simulator, re-execute a benchmark, use the insight gained to add more debugging code, re-execute a benchmark, and so on. This problem only increases when considering alternatives for microarchitectural features (sizes of hardware tables, policies decisions, widths, history depths, etc.). Simulators typically output summary statistics for how a given benchmark ran overall. However, summary statistics can potentially hide the subtle interactions and events that can affect overall performance, since these statistics are often averages across a large number of instructions.

The main motivation for a tool such as the one being presented is to ease these burdens. First there is a desire to visualize traces generated from a simulator in a way that is more easily navigable. Pipeline tracing tools like the *dptv* allow for the user to investigate more closely the run-time behavior of a particular execution than statistics alone would provide. Second is the desire to compare two traces of the same benchmark under different microarchitectural configurations more easily. It is common to run a benchmark multiple times to compare which configuration gives the best performance, however it could also be useful to tell where exactly in a benchmark performance diverges between configurations. If designed right, a pipeline tracing tool is a prime candidate for such a task. By superimposing the traces of multiple executions of a benchmark under different microarchitectural models, it becomes easier to both identify at what point performance diverges, but also the specific behavior that may have caused performance to differ.

dptv aims to make the process of visualizing and comparing traces as easy as possible. Figure 1 shows a basic screenshot from *dptv*. Its interface is graphical, using the SDL2 graphics library [3], which allows the user to both zoom and pan around the diagram, familiar to anyone having used photo editing or CAD tools. By hovering the mouse over a particular pipeline stage marker, identified by a character representing the type of stage, more information about the machine state at that point in time can be viewed (shown in the upper-right corner). The trace can be searched for occurrences of text in the pipeline stage information, the program counter, instruction text, and more. Users can zoom in/out on the diagram, enabling both visualization of the long-term performance of the trace, and also investigation of pipeline events over a short period of time.

Several pipetrace viewers have been proposed [2] [13] [14] in the literature. The main feature which differentiates *dptv* from these other tools is the ability to visualize two traces at once. When visualizing two traces, each trace will be interleaved, effectively showing traces atop each other, with each row alternating which trace its information is from. As previously mentioned, this greatly helps with comparing and contrasting the characteristics of the two executions. Since varying microarchitectural features potentially impacts the clock period, *dptv* allows the two traces to operate at different frequencies – scaling the visualization to accurately represent the actual run-time of the traces.

In addition to visualization goals, the development of *dptv* is also making integration into a wide variety of simulation infrastructure as a first class design constraint. As such, *dptv* is instruction set agnostic, simply treating instructions as plain text. Furthermore, pipeline stages can be annotated with arbitrarily many events in a key-value pair format. And *dptv* was written in standard C, with the goal of portability in mind – with only the reliance of the SDL2 library in addition to standard C libraries.

The remainder of this paper will go more in-depth on specific aspects of *dptv*. Section 2 provides more context on the current state of the art for tools of this nature, as well as related prior work. Section 3 will cover the various features provided by *dptv* in depth. Section 4 will give an example in which using *dptv* aids in quickly understanding the difference in performance between two traces. And finally Section 5 will conclude the paper with a brief discussion on the current status of the tool, what is finished and what still needs to be done, and ideas for future work.

2. Background

Architects have grappled with architectural complexities, and scale issues since the earliest simulators and benchmarks. Using graphical visualization tools to better understand computer performance has a well established history.

Gao, et al. [10] provide a survey paper that summarizes the current landscape of a wide variety of visualization tools, 21 tools in total, as of their publication date in 2011. They further derive a characterization of these tools, according to a taxonomy proposed by Card [9], borrowing insight from the Human-Computer Interaction research community. These 21 tools span the gamut from tools that more appropriately graph summary statistics, up to tools that provide visualizations of data synthesized across highly dimensional data sets.

Accelerator and GPU-style many core architectures present a class of bottlenecks that tend not to manifest in general purpose processor architectures. For instance, memory bandwidth is more highly constrained in many-core architectures. The lock-step execution of warps in GPUs, and control divergences present unique challenges. And to meet these challenges, visualization tools more appropriately highlight the needs of these architectures. In [5], Ariel et al. present a tool for displaying DRAM channel utilization, warp control divergences, histograms of static instruction usage, as well as mechanisms to show which instructions are the sources of bottlenecks for GPU and SIMT-style architectures. Similarly, Alsop et al. provide a GPU-specific visualization [4] for highlighting sources of memory stalls.

Not all architectures have performance as their main target metric. Mobile systems have energy efficiency and power as first class design constraints. In [6], the authors instrument a multithreaded processor simulator to track with activity counters that are fed to a tool that is able to display a processor floorplan with a 3D height map such that the height represents the relative power consumption of hardware units. This visualization can be scrubbed forward or backward in time to see how the height map (thus, power consumption) changes as the program behavior stresses different hardware units over time.

The closest relatives of *dptv* exist in four tools, TraceVis [13], PSE [12], GPV [14] and `o3` [2]. None of these tools allow for traces from multiple microarchitectural configurations to be visualized simultaneously.

TraceVis [13] uses the same pipeline timing diagram style of visualization as *dptv*. Furthermore, TraceVis uses the Qt windowing toolkit to allow for zooming in and out, like *dptv* allowing for visualizing both course and fine grained program behavior.

The Processor Simulation Elucidator (PSE) [12] uses the GTK windowing toolkit to similarly display Pipeline Event Diagrams (PEDs). These PEDs are simply pipeline timing diagrams, with pipeline event information associated with each instruction. PSE does have additional modes of visualization not found in *dptv*, used to show how performance metrics (those that will likely become summary statistics) vary over time.

The Graphical Pipeline Viewer (GPV) [14] is an early example among these pipeline timing diagram style viewers, written in Perl TK. GPV is integrated into the once popular SimpleScalar [8] simulation framework, and also incorporates a visualization of power consumption.

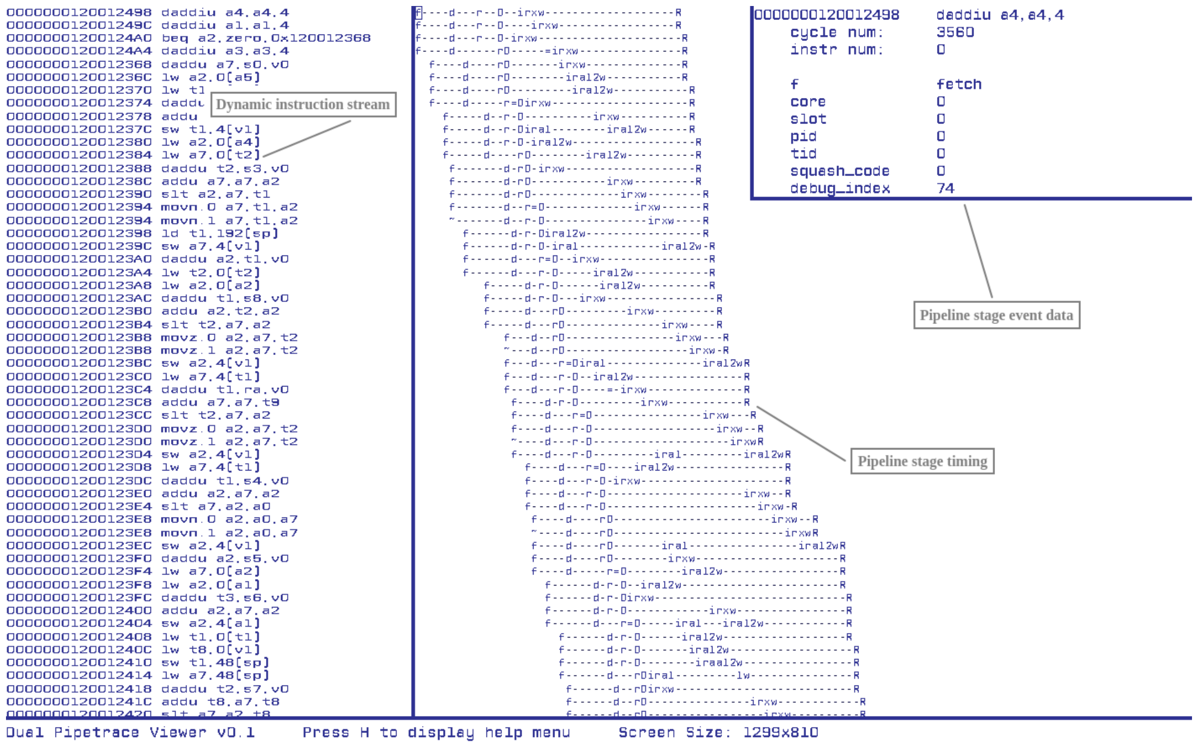
A somewhat rudimentary, but comparable, tool bundled with the popular `gem5` [7] simulator is the `o3 pipeline viewer` [2]. Again, `o3` shows a pipeline timing diagram style visualization. However, `o3` produces a non-interactive, text-only output file, formatted with a fixed column width. Each column represents one cycle of execution time. When an instruction stays in-flight for more cycles than there are columns of text, then the remaining cycles simply wrap to the next line. This limitation makes it extremely difficult to find performance bottlenecks and correlate those bottlenecks with pipeline events.

3. *dptv* Viewer Tool

This section will go in-depth on the usage and features of *dptv*, as well as the visualization provided.

dptv is currently setup to build in a Linux environment, with the only extenal library used being SDL2 [3] for rendering. *dptv* is invoked on the command-line with either one or two trace file names as input. If only one trace file name is provided, then the program will open in single-trace mode. If two traces are provided, the viewer will first verify that the trace data held within the files are the same instruction stream, and if so, the visualization will open in dual-trace mode. This is a key new feature, unique to *dptv*. Traces from two different instruction streams are not meaningful comparisons, and if the tool determines that the *committed* instructions are not the same, it will simply fail to start the visualization. Since only committed instructions are compared, if two traces differ by squashed instructions only, then *dptv* can still visualize the two trace.

It is possible that two processor configurations operate at different clock frequencies, depending on microarchitectural features. *dptv* handles this possibility. If the two traces should be displayed at different frequencies, a command-line argument can be passed when invoking *dptv* which will scale the traces based on the ratio of the frequencies. Additional command-line options exist to change the color palette used, as well as the position and size of the window.



```
000000012001249B daddiu a4.a4.4
000000012001249C daddiu a1.a1.4
00000001200124A0 beq a2.zero.0x120012368
00000001200124A4 daddiu a3.a3.4
0000000120012368 daddu a7.s0.v0
000000012001236C lw a2.0[a5]
0000000120012370 lw t1
0000000120012374 daddu
0000000120012378 addu
000000012001237C sw t1.4[v1]
0000000120012380 lw a2.0[a4]
0000000120012384 lw a7.0[t2]
0000000120012388 daddu t2.s3.v0
000000012001238C addu a7.a7.a2
0000000120012390 slt a2.a7.t1
0000000120012394 movn.0 a7.t1.a2
0000000120012394 movn.1 a7.t1.a2
0000000120012398 ld t1.192[sp]
000000012001239C sw a7.4[v1]
00000001200123A0 daddu a2.t1.v0
00000001200123A4 lw t2.0[t2]
00000001200123A8 lw a2.0[a2]
00000001200123AC daddu t1.s8.v0
00000001200123B0 addu a2.t2.a2
00000001200123B4 slt t2.a7.a2
00000001200123B8 movz.0 a2.a7.t2
00000001200123B8 movz.1 a2.a7.t2
00000001200123BC sw a2.4[v1]
00000001200123C0 lw a7.4[t1]
00000001200123C4 daddu t1.ra.v0
00000001200123C8 addu a7.a7.t9
00000001200123CC slt t2.a7.a2
00000001200123D0 movz.0 a2.a7.t2
00000001200123D0 movz.1 a2.a7.t2
00000001200123D4 sw a2.4[v1]
00000001200123D8 lw a7.4[t1]
00000001200123DC daddu t1.s4.v0
00000001200123E0 addu a2.a7.a2
00000001200123E4 slt a7.a2.a0
00000001200123E8 movn.0 a2.a0.a7
00000001200123E8 movn.1 a2.a0.a7
00000001200123EC sw a2.4[v1]
00000001200123F0 daddu a2.s5.v0
00000001200123F4 lw a7.0[a2]
00000001200123F8 lw a2.0[a1]
00000001200123FC daddu t3.s6.v0
0000000120012400 addu a2.a7.a2
0000000120012404 sw a2.4[a1]
0000000120012408 lw t1.0[t1]
000000012001240C lw t8.0[v1]
0000000120012410 sw t1.48[sp]
0000000120012414 lw a7.48[sp]
0000000120012418 daddu t2.s7.v0
000000012001241C addu t8.a7.t8
0000000120012420 slt a7.a2.t8
```

```
0000000120012498    daddiu a4,a4,4
    cycle num:      3560
    instr num:      0

    f               fetch
    core            0
    slot            0
    pid             0
    tid             0
    squash_code     0
    debug_index     74
```

```
Dual Pipetrace Viewer v0.1      Press H to display help menu      Screen Size: 1299x810
```

Figure 1. *dptv* at default zoom in single-trace mode, annotations indicate different aspects of the visualization

3.1. Visualization

Figure 1 shows an instance of *dptv* which has just been opened visualizing an example trace in single-trace mode. Note that for the figures in this paper the background color has been changed to white to make them more gray scale printer-friendly. The pane on the right shows the trace displayed as previously described, with each pipeline stage plotted against time on the x-axis and dynamic instruction stream order on the y-axis. Each character column of the pipeline stage timing represents a cycle. Therefore, a column shows which pipeline stages were executed on the same cycle, and each row contains instructions from the execution of the execution of the benchmark. The left pane shows the list of dynamic instructions executed (and their instruction addresses) on the same row as their associated stages.

3.2. Stage Information

Pipeline stages are represented on the plot (right pane) by a single character which indicates which stage was executed. For example, ‘f’ represents Instruction Fetch, ‘d’ represents Instruction Decode, etc. The processor model shown in Figure 1 is an out-of-order execution pipeline, so instructions can potentially be buffered for several cycles before being selected for execution. The visualization uses dash ‘–’ characters to indicate that an instruction has not executed any new pipeline stage for that cycle.

By hovering the mouse over one of the stages, additional information can be viewed. This additional information is the event data for the pipeline stage, for the given instruction. During simulation, an arbitrary amount of pipeline stage event data can be saved to the trace file. The event data is in a name/value pair. Figure 1 shows the information from a fetch stage being viewed (notice there is a mouse cursor highlighting the ‘f’ character for the very first instruction). Stage information is shown in the top-right corner and includes the full name of the stage, the instruction it’s from, the cycle and instruction position, as well as stage-specific information such as the values of registers, which lane of the superscalar processor it is being executed on, etc. Stage information is included in the trace so what exact information is included is dependent on the program generating the trace.

3.3. Movement

The user can pan the diagram either by using the left mouse button and dragging the view, or by using key bindings. If the trace is off the window, a key binding can be pressed to snap the view horizontally back into the window. Another key binding can be pressed to reset the view to the default location as upon startup.

Zooming is done using the mouse scroll wheel. Scrolling down will zoom out, and scrolling up will zoom in. The zoom is centered on the current location of the mouse pointer. These zoom mechanics should be very familiar to any user having experience using CAD tools, photo editors, or (for example) Google Maps.

When zoomed out far enough, the letters representing each stage would become unreadable. Instead, *dptv* will switch the visualization such that the text disappears (both panes), and the pipeline stage timing panes switch to lines. These lines will connect like-pipeline stages from one instruction to the next – by default, a line to connect the first pipeline stage, Instruction Fetch and a second line for final commit/retirement of instructions. These pair of lines thus show the lifetime of all instructions, the space between these lines. Figure 2 shows an example trace, zoomed out far enough to show the line representation of the instruction stream. There are roughly 10 thousand dynamic instructions represented at the zoom level represented in this figure.

The pipeline stage that is connected by lines in the zoomed-out mode can be changed to a single stage, for any of the pipeline stages. For example, it might be helpful to see a line that connects the Execute pipeline stage for each instruction. This is changed with an additional command-line argument when invoking *dptv*.

3.4. Searching

Facilities to search through traces is also implemented in *dptv*. A key binding will initiate a search (the slash, similar to initiating a search in `vi`/`vim`), after which the search term can be typed and enter can be pressed to search. After entering in the search term additional key bindings allow the user to continue to the next match of the same search term. Both the current match and all other matches are highlighted. Unlike `vi`/`vim`, the search terms are not regular expressions.

dptv will search traces for occurrences of the search term; by default it will search in the instruction text, pc text, stage identifier (the single character on the plot), stage name, and the fields in the stage information. There is functionality to search in only one selected field, done by typing ‘/’, then a character, then another ‘/’, then the search string. For example, “/i/lw” will search only the instruction text for the string “lw”. Additionally, the contents of a specific stage event data can be searched using “/v:(name)/(value)”. A list of fields to search in and their associated search characters is included in the help text of the program.

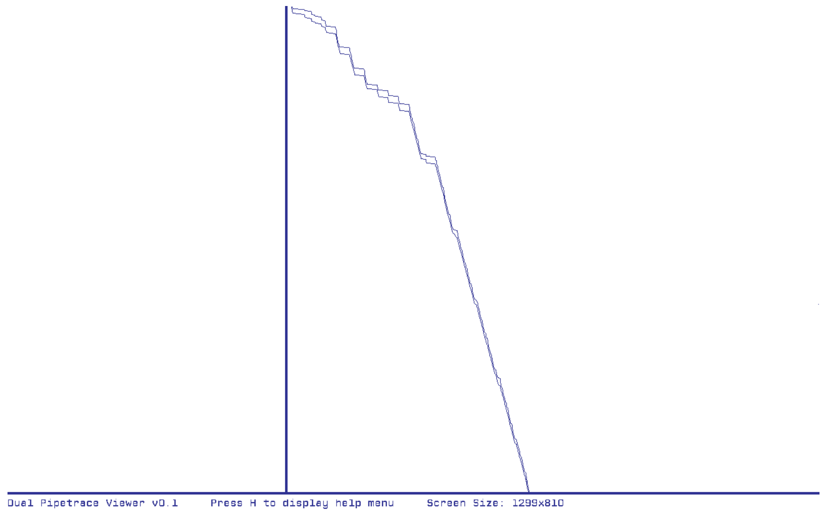


Figure 2. *dptv* at zoomed out in single-trace mode, such that lines connect the fetch stages and retire stages

There is also functionality to jump to a specific point in the trace, initiated by pressing ‘:’, entering the instruction number to jump to, then pressing enter.

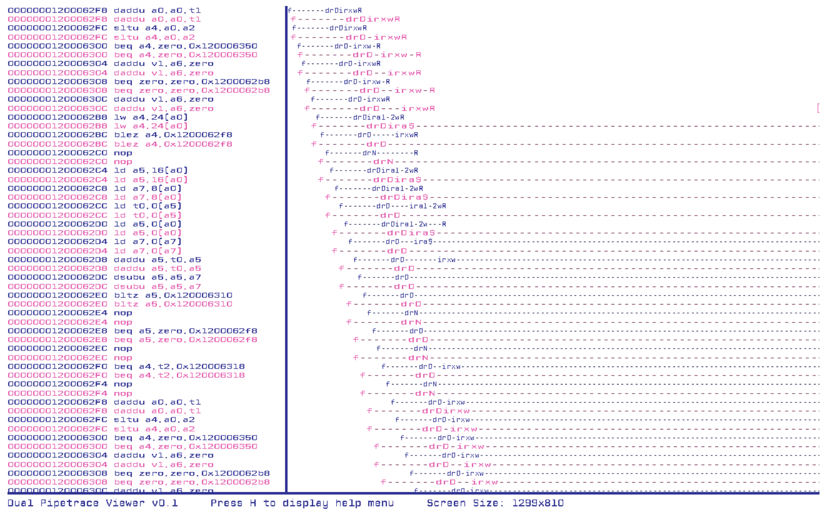
3.5. Dual-Trace Mode

When visualizing two traces, the plot will alternate between each trace for each row, as seen in Figure 3. Notice the right pane shows the same instruction twice, colored differently – these are the same instruction from each simulated processor configuration. Each traces instructions and stages are colored differently, including when zoomed out. These colors can also be changed at the command-line. As discussed previously, both traces are expected to be from the same benchmark, meaning all completed instructions (i.e. all instructions which retire) are the same in both traces. However, squashed instructions may differ, for example if one configuration predicts a taken branch while the other does not – whichever configuration mispredicted will have squashed instructions that will not retire. *dptv* will align all matching retired instructions, inserting dummy instructions to fill any necessary gaps. *dptv* will not attempt to visualize traces which do not have a common stream of committed instructions. If the two traces begin at different points in execution, by default *dptv* will remove the instructions before/after the common section.

Note again that the two traces do not need to be recorded at the same frequency. If they are not (also shown in Figure 3), then the trace with the slower frequency will be stretched horizontally to line the traces up correctly in time. This also means that two stages with the same cycle position but from different traces may not line-up in time.

The panning and zooming system works well with the dual-trace view to make comparing the traces easy. For example, the view can be zoomed far out to look at the trend lines of both traces, and then can be zoomed back in to points where the traces performance diverges, allowing the user analyze the specific details of what happens differently between them.

When comparing two traces, it is possible, in fact very likely, that the two traces have performance that progressively displays one trace off of the window. It is helpful to be able to re-establish a point in the



```
000000012000062F8 daddu a0.a0.t1
000000012000062F8 daddu a0.a0.t1
000000012000062FC sltu a4.a0.a2
000000012000062FC sltu a4.a0.a2
00000001200006300 beq a4.zero.0x120006350
00000001200006300 beq a4.zero.0x120006350
00000001200006304 daddu v1.a6.zero
00000001200006304 daddu v1.a6.zero
00000001200006308 beq zero.zero.0x1200062b8
00000001200006308 beq zero.zero.0x1200062b8
0000000120000630C daddu v1.a6.zero
0000000120000630C daddu v1.a6.zero
000000012000062B8 lw a4.24[a0]
000000012000062B8 lw a4.24[a0]
000000012000062BC blez a4.0x1200062f8
000000012000062BC blez a4.0x1200062f8
000000012000062C0 nop
000000012000062C0 nop
000000012000062C4 ld a5.16[a0]
000000012000062C4 ld a5.16[a0]
000000012000062C8 ld a7.8[a0]
000000012000062C8 ld a7.8[a0]
000000012000062CC ld t0.0[a5]
000000012000062CC ld t0.0[a5]
000000012000062D0 ld a5.0[a0]
000000012000062D0 ld a5.0[a0]
000000012000062D4 ld a7.0[a7]
000000012000062D4 ld a7.0[a7]
000000012000062D8 daddu a5.t0.a5
000000012000062D8 daddu a5.t0.a5
000000012000062DC dsubu a5.a5.a7
000000012000062DC dsubu a5.a5.a7
000000012000062E0 bltz a5.0x120006310
000000012000062E0 bltz a5.0x120006310
000000012000062E4 nop
000000012000062E4 nop
000000012000062E8 beq a5.zero.0x1200062f8
000000012000062E8 beq a5.zero.0x1200062f8
000000012000062EC nop
000000012000062EC nop
000000012000062F0 beq a4.t2.0x120006318
000000012000062F0 beq a4.t2.0x120006318
000000012000062F4 nop
000000012000062F4 nop
000000012000062F8 daddu a0.a0.t1
000000012000062F8 daddu a0.a0.t1
000000012000062FC sltu a4.a0.a2
000000012000062FC sltu a4.a0.a2
00000001200006300 beq a4.zero.0x120006350
00000001200006300 beq a4.zero.0x120006350
00000001200006304 daddu v1.a6.zero
00000001200006304 daddu v1.a6.zero
00000001200006308 beq zero.zero.0x1200062b8
00000001200006308 beq zero.zero.0x1200062b8
0000000120000630C daddu v1.a6.zero
Dual Pipetrace Viewer v0.1     Press H to display help menu     Screen Size: 1299x810
```

Figure 3. *dptv* displaying traces of instructions that were executed on two different processor configurations

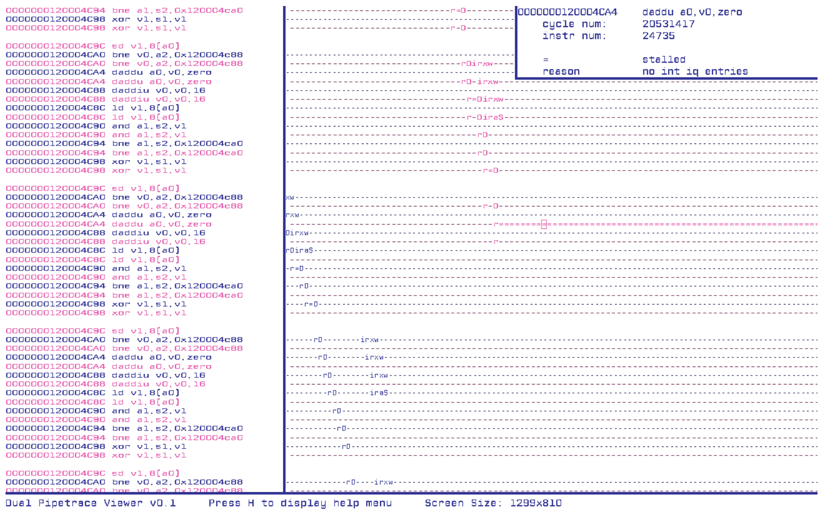
visualization where the traces are realigned in time. For example, suppose the two traces may have the same performance for a phase of the benchmark, but maybe one trace has a cache miss that did not occur in the opposing trace. This causes the lower performing trace to stall for a little bit, pushing the visualization off the right hand side of the window. Maybe later both traces return to the same level of performance.

A similar scenario is shown in Figure 3. At the beginning of time both traces start together, but after a while, both traces will continue to be spaced out (notice fetch 'f' for like instructions diverge) even if their performance reconverges. To remedy this, the right mouse button can be pressed to realign the traces to that point time. The traces will shift horizontally to converge at the point clicked. This realignment can be done at any zoom level.

4. Example Usage

For an example of performance comparison and the utility of *dptv*, we simulated a phase of the `462.libquantum` benchmark from the SPEC CPU2006 Benchmark Suite [1]. This benchmark is known to have performance that scales well with additional microarchitectural resources. We ran the simulation twice: once with a superscalar width of one and again with a superscalar width of two, and all other microarchitectural variables (hardware table sizes, predictor implementations, etc.) remaining constant. The summary statistics showed that the performance of the two executions were very close, however the one-wide simulation ran slightly faster than the two-wide simulation. This is a surprising result, and prompts the question of *why* this would be the case. We load the generated `462.libquantum` traces into *dptv* to elucidate the run-time behavior. Figures 4 and 5 shows a zoomed-in view and a zoomed-out view, respectively. The one-wide trace is colored in cyan and the two-wide trace is colored in magenta in both figures.

It becomes immediately evident that the phase of `462.libquantum` traces consists of a small loop body, seven static instructions. This same loop continues for the entirety of both traces. In the zoomed-out view we can see that both traces seem to delay execution (the horizontal lines on the zoomed-out graph)



```
000000012004C94 bne a1,s2,0x120004ca0
000000012004C98 xor v1,s1,v1
000000012004C98 xor v1,s1,v1

000000012004C9C sd v1,8[a0]
000000012004CA0 bne v0,a2,0x120004c88
000000012004CA0 bne v0,a2,0x120004c88
000000012004CA4 daddu a0,v0,zero
000000012004CA4 daddu a0,v0,zero
000000012004C88 daddiu v0,v0,16
000000012004C88 daddiu v0,v0,16
000000012004C8C ld v1,8[a0]
000000012004C8C ld v1,8[a0]
000000012004C90 and a1,s2,v1
000000012004C90 and a1,s2,v1
000000012004C94 bne a1,s2,0x120004ca0
000000012004C94 bne a1,s2,0x120004ca0
000000012004C98 xor v1,s1,v1
000000012004C98 xor v1,s1,v1

000000012004C9C sd v1,8[a0]
000000012004CA0 bne v0,a2,0x120004c88
000000012004CA0 bne v0,a2,0x120004c88
000000012004CA4 daddu a0,v0,zero
000000012004CA4 daddu a0,v0,zero
000000012004C88 daddiu v0,v0,16
000000012004C88 daddiu v0,v0,16
000000012004C8C ld v1,8[a0]
000000012004C8C ld v1,8[a0]
000000012004C90 and a1,s2,v1
000000012004C90 and a1,s2,v1
000000012004C94 bne a1,s2,0x120004ca0
000000012004C94 bne a1,s2,0x120004ca0
000000012004C98 xor v1,s1,v1
000000012004C98 xor v1,s1,v1

000000012004C9C sd v1,8[a0]
000000012004CA0 bne v0,a2,0x120004c88
000000012004CA0 bne v0,a2,0x120004c88
000000012004CA4 daddu a0,v0,zero
000000012004CA4 daddu a0,v0,zero
000000012004C88 daddiu v0,v0,16
000000012004C88 daddiu v0,v0,16
000000012004C8C ld v1,8[a0]
000000012004C8C ld v1,8[a0]
000000012004C90 and a1,s2,v1
000000012004C90 and a1,s2,v1
000000012004C94 bne a1,s2,0x120004ca0
000000012004C94 bne a1,s2,0x120004ca0
000000012004C98 xor v1,s1,v1
000000012004C98 xor v1,s1,v1

000000012004C9C sd v1,8[a0]
000000012004CA0 bne v0,a2,0x120004c88
```

```
000000012004CA4    daddu a0,v0,zero
    cycle num:     20531417
    instr num:     24735

    =              stalled
    reason         no int iq entries
```

```
Dual Pipetrace Viewer v0.1     Press H to display help menu     Screen Size: 1299x810
```

Figure 4. *dptv* showing fine grained instruction behavior of the `462.libquantum` benchmark for two processor configurations

on a semi-regular basis. The delays seem to always occur on the same instruction when zoomed-in, but not on every loop iteration. In-between the delays the two-wide trace has a steeper slope, meaning it executes more instructions over the same cycle time, as is expected. What makes it fall behind the one-wide seems to be both the number and duration of these delays, so our next goal is to find out what is causing those. The simulator used to generate these traces provides information about each stall in the instruction that caused it. Zooming into a section where the two-wide execution delays and scrolling up (going back in instruction order but staying at the same point in time), we find the result of the delays is an instruction stalling due to a full issue queue. Since the two-wide fills execution resources more quickly than the one-wide, these structures fill, eventually stalling the processor front-end. The simulator used to generate these traces uses typed lanes for the back-end, requiring at least a lane for arithmetic

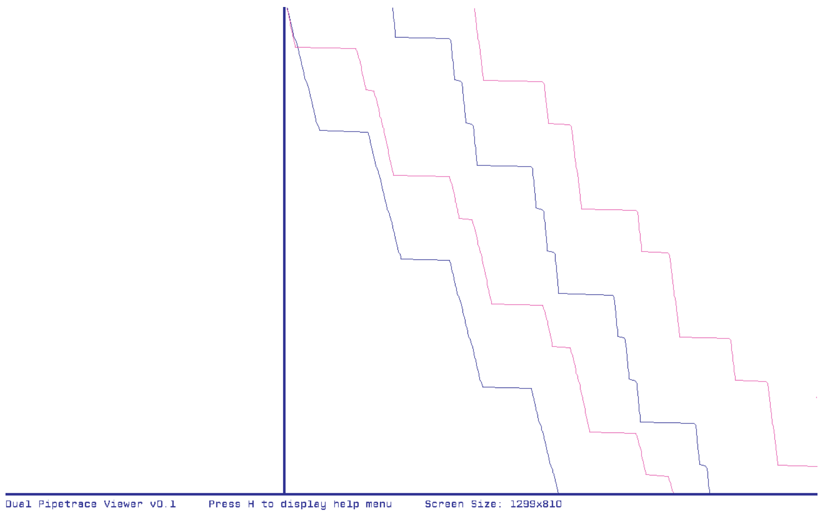


Figure 5. *dptv* showing broad instruction behavior of the `462.libquantum` benchmark for two processor configurations

instructions, a lane for CTIs, and a lane for memory instructions. Thus the back-end width must be at least three-wide, greater than the frond-end width of the simulated one-wide and two-wide. Since the two-wide fills the issue queue at a higher rate than the one-wide, yet they both drain the issue queue at the same rate depending on the instruction mix, this causes the two-wide to exceed the issue queue capacity at times when the one-wide did not.

While the problem most certainly could have been identified without *dptv*, the use of this tool greatly increased the speed at which the issue could be identified and possible fixes considered.

5. Status and Future Work

dptv is still under active development, although as of this publication date, its code base is quite mature and stable. It is able to open large trace files, upwards of hundreds of thousands to millions of dynamic instructions. Although extensive testing has not been done, relatively modest hardware (8th generation Intel Core i5 laptop, 8GB memory, integrated graphics) can fluidly navigate traces. Testing on this viewer tool will continue, but it is not anticipated that any major changes will be necessary.

Development and testing has primarily been carried out on a Linux platform. Future work will ensure compatibility with Apple MacOS and Microsoft Windows environments.

In addition to the stand-alone viewer tool, it is also anticipated to release libraries written in various languages, intended to be easily integrated into existing processor simulations. These libraries will provide a straight-forward API to collect trace information while simulating benchmarks. As a simulation completes, the API can then automatically generate the trace file in the appropriate file format to be read into the viewer tool. Currently, there is only one library, written in C++, which has been integrated into an in-house developed processor simulator that is not publicly available. A possible change to the code base will use the same file format as that used by `o3`, to simplify integration with `gem5`. Currently, *dptv* uses a custom plain-text file format.

Up-to-date project status, and eventual code release will be maintained at https://cs.uwlax.edu/~eforbes/dptv/.

References

[1] “SPEC CPU2006 Benchmark Suite.” The Standard Performance Evaluation Corporation, 2006. [Online]. Available: http://www.spec.org/cpu2006/

[2] “gem5: Visualization,” 2023, Reference. [Online]. Available: https://www.gem5.org/documentation/general_docs/cpu_models/visualization/

[3] “Simple DirectMedia Layer,” 2023, Reference. [Online]. Available: https://www.libsdl.org/

[4] J. Alsop, M. Sinclair, R. Komuravelli, and S. Adve, “GSI: A GPU Stall Inspector to Characterize the Sources of Memory Stalls for Tightly Coupled GPUs,” in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, April 2016, pp. 172–182.

[5] A. Ariel, W. Fung, A. Turner, and T. Aamodt, “Visualizing Complex Dynamics in Many-Core Accelerator Architectures,” in *Proceedings of the 2010 IEEE International Symposium on Performance Analysis of Systems and Software*, March 2010, pp. 164–174.

[6] A. Baranov, P. Panfilov, and D. Ponomarev, “PowerVisor: A Toolset for Visualizing Energy Consumption and Heat Dissipation in Modern Processor Architectures,” in *Proceedings of the 12th International Conference on Parallel Computing Technologies*, September 2013, pp. 149–153.

[7] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, “The gem5 Simulator,” *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, May 2011.

[8] D. Burger, T. M. Austin, and S. Bennett, “Evaluating Future Microprocessors: the SimpleScalar Tool Set,” 1996.

[9] S. Card, “Information Visualization,” in *The Human-Computer Interaction Handbook: Fundamentals*, January 2002, pp. 209–543.

[10] Q. Gao, X. Zhang, P. Rau, A. Maciejewski, and H. Siegel, “Performance Visualization for Large-Scale Computing Systems: A Literature Review,” *Human-Computer Interaction. Design and Development Approaches: 14th International Conference, HCI International*, vol. 10, no. 1, pp. 450–460, July 2011.

[11] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach, 5th ed.* Waltham, MA: Morgan Kaufmann, 2012.

[12] D. Koppelman and C. Michael, “Discovering Barriers to Efficient Execution, Both Obvious and Subtle, Using Instruction-Level Visualization,” in *Proceedings of the First Workshop on Visual Performance Analysis (held in conjunction with Supercomputing SC14)*, November 2014.

[13] J. Roberts and C. Zilles, “TraceVis: An Execution Trace Visualization Tool,” in *Workshop on Modeling, Benchmarking and Simulation*, June 2005.

[14] C. Weaver, K. Barr, E. Marsman, D. Ernst, and T. Austin, “Performance Analysis Using Pipeline Visualization,” in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, November 2001, pp. 18–21.