Performance Analysis Using Pipeline Visualization

Chris Weaver, Kenneth C. Barr, Eric Marsman, Dan Ernst, and Todd Austin
Advanced Computer Architecture Laboratory
University of Michigan
Ann Arbor MI 48104
chriswea@eecs.umich.edu

Abstract

High-end microprocessors are increasing in complexity to push the limits of speed and performance. As a result, analyzing these complex systems can be an arduous task. Architectural simulators, acting as software processors, are able to run programs and give statistics about the performance of the code on the design. While these statistics are valuable for identifying problems, they often do not provide the fidelity necessary to diagnose the cause of sluggish performance. This paper presents a cross-platform tool that can be used to visualize the flow of instructions through an architectural processor pipeline model. The Graphical Pipeline Viewer, GPV, uses a colorized pipeline trace display to deliver an efficient diagnostic and analysis environment. The resource view of the tool, which can display cycle statistics, aids in distinguishing possible bottlenecks and architectural trade-offs. As such, the tool is able to suggest code and architectural modifications to increase program performance.[1] [2]

1. Introduction

In an effort to optimize system performance, designers must find and fix bottlenecks. Complex systems, such as a microprocessor, can make this a very difficult task because of the numerous interactions between the many components of the design. Software developers are often unaware of the structure of the hardware that executes their software. A mismatch between a programmer's conceptual model of the system and the actual implementation could result in "optimizations" that make the program run slower. Similarly, hardware developers target architectural optimizations to programs representative of their market. In doing so, they must be keenly aware of the effects these changes will have on the performance and utility of other components.

Traditional methods for performance analysis use summary statistics to describe the system's behavior. Architectural simulators are used to determine the large range of effects that can result from a single modification. Profiling can also be used in conjunction with a simulator to locate key areas of interest in the program. Unfortunately, cumulative statistics mask the intricacies of the program execution. Statistics only give an idea of magnitude and not sparsity, which can allow regions of poor performance or adverse interactions between components to go undetected. As a result, a long and tedious series of design analyses is often required to obtain the resolution needed to ascertain the root causes of performance bottlenecks.

The Graphical Pipeline Viewer (GPV), presented in this paper, provides the capabilities and fidelity necessary to quickly locate bottlenecks in complex systems. The visualization interface allows users to zoom in or out to detect the high-level trends in the code or study small regions of code to discover the cause of a slowdown. In addition, its execution comparison capabilities make it perfectly suited for evaluating hardware and software optimizations. While visualization of statistics can help identify potential problems, a tool must be written such that a user can unlock this potential.We have identified several guidelines for making an architectural pipeline visualization tool:

- *Simple Generic Interface*. The visualization tool needs to be designed such that it can easily communicate with an architectural simulator. Our tool does this by using text streams that detail changes in pipeline and resource status. While this is not an optimized interface, it does allow for the easy interfacing to a variety of simulators.
- *Cross Platform Capability.* The more platforms that can run the tool, the more useful it becomes. It is not uncommon for universities to use a variety of computing environments. As a side effect of being cross platform compatible, a single tool can be used for development, testing, as well as demonstrations.
- *Ability to compare simulation runs.* The tool needs the ability to easily compare executions to determine the impact that code or microarchitectural changes have on performance. Our tool supports the visualization of multiple runs in a single window for easy contrast.
- *Ability to get both coarse grain and fine grain detail.* A coarse grain resolution is needed to determine when and where performance is poor, and a detailed view permits close examination of the cause(s) of the delay(s). GPV supports this by allowing the user to change the level of detail in the visualization display.
- *Easy interpretation of graphics and symbols*. The graphics should be designed in such a way that regions of poor performance are easy to isolate. Our approach color-codes high latency events, making them easy to identify on the visualization display. In addition, resource utiliza-

1. While we have tried to use colors that will have high contrast when printed in grey scale, this paper is best viewed in color. The tool that we describe uses color as one of the key methods to differentiate events
2. The reader is referred to the technical report on GPV for case studies that employ this technology. [19]

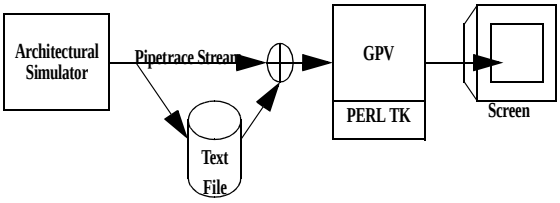

Figure 1 Overview the GPV usage flow

tion is graphical (such as IPC, available functional units, or instruction window utilization), which often reveals problematic regions of the code.

The next section presents our versatile visualization infrastructure that meets these tough demands. We discuss the viewer and the generic input stream that makes it easily adaptable to most simulators. We end with some conclusions derived from this work.

2. Graphical Pipeline Viewer

Figure 1 gives an overview of our pipeline viewer. An architectural simulator is used to produce a pipetrace stream. This stream contains a detailed description of the instruction flow through the machine, documenting the movement of instructions in the pipeline from “birth” to “death”. In addition, the pipetrace stream denotes various other events and stage transitions that occur during an instruction’s lifetime. The pipetrace stream from the architectural simulator can be sent directly into GPV or buffered in a file for later analysis. GPV digests this information and produces a graphical representation of the data. The graph generated by GPV plots instructions in program order, denoting over the lifetime of an instruction what operation it was performing or why it was stalled. In addition, the tool is able to plot any other numeric statistics on a resource graph. Multiple traces can be displayed on the screen at any given time for easy analysis. GPV also supports both coarse and fine grain analysis through the use of a zoom function. Color coded events, which are user definable, makes spotting potential bottlenecks a simple task. The remainder of this section will outline the tool in detail, including the main view, advanced features, trace file format, and other infrastructure with which GPV has been designed to communicate.

2.1 Main Visualization Window

The main GUI window of GPV is illustrated in Figure 2. The GUI has two main graphical display windows, the instruction window and the resource window. The instruction window plots instructions in program order on a time axis (measured in cycles). For example, the third instruction bar in Figure 2, shows the execution of an ADDQ instruction on a 4-wide Alpha simulator. As shown in the figure, this instruction is stalled in Fetch (IF) until the stall in the internal ld/st is resolved, after which it continues to completion. This method for graphing instructions as they flow through a pipeline is a common visual representation, used in many textbooks including Hennessy and Patterson [2]. The instruction axis contains tick mark to indicate the

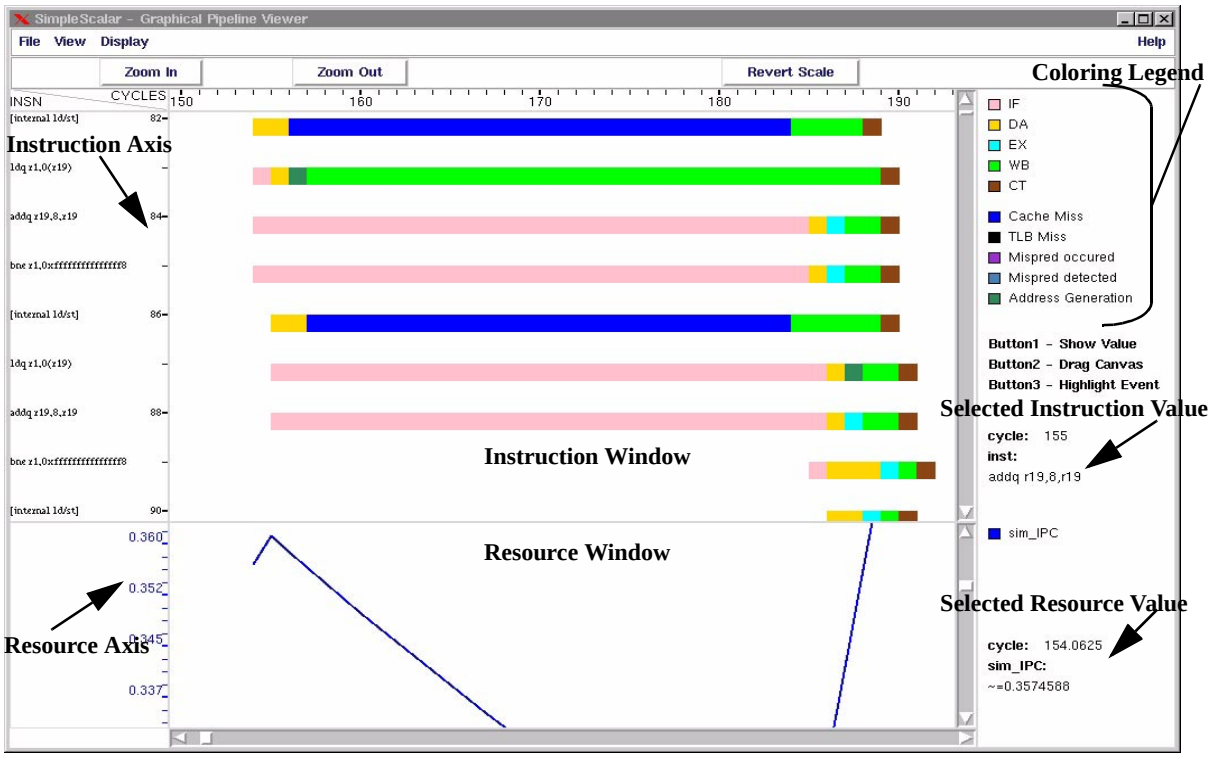

Figure 2 GPV Display Window showing the execution of instructions on a 4-wide Alpha ISA model (note that internal micro-code operations, i.e. internal ld/st, are allowed to finish out of program order)

cycle count. Additionally, the vertical axis will also display the instruction mnemonic when the window is zoomed in enough to fit legible text aside each instruction mark (typically two zooms from when the pipetrace is first loaded). The right panel provides a legend of the coloring that is used to illustrate the instruction's flow through the different stages of the pipeline. Significant events, such as branch mispredictions or cache misses, are displayed in conjunction with the instruction's transitions through the pipeline. The use of color (with a user configurable palette) provides an effective means for spotting potential bottlenecks. A highlight option, which can flash the occurrences of a particular event, can be used as an alternative method of locating bottlenecks.

The bottom window, the resource view, displays graphs of any numeric statistic provided in the pipetrace file. GPV has been designed to plot both integer and real statistics. Up to four data sets (our current development extends this to ten) can be displayed simultaneously with color coded axes that indicate the range of the variable. Since there can be a wide variation in the data range of a statistic, a separate x-axis is provided for each one of the four resources that can be displayed at a time. Both the resource and instruction views are plotted against simulator time on the x-axis. This permits widely varying statistical data sets to be plotted within the same window. To avoid clutter, the GUI allows the selective hiding of individual resource views. The resource view in Figure 2 is shown plotting the IPC of a simulated program. As shown in the figure, the IPC of the program starts to drop during the cache miss. Once the miss has been handled and instructions start to retire, the IPC begins to recover. The flexibility of the resource view allows the user to chose the statistics that are most valuable for performance analysis and correlate these statistics to instructions flowing through the pipeline. This simplifies the task of identifying bottlenecks, as illustrated by the relationship of the cache miss to the IPC drop in Figure 2.

The GUI provides several additional features that assist in diagnosing performance bottlenecks. The display can be zoomed in and out to trade off detail for trend analysis. When the display is zoomed out it is straightforward to determine areas of low performance by locating pipeline trace regions with low slope. The slope of the line is given by [1]:

$$slope = \frac{\Delta y}{\Delta x} = -(IPC)$$

$$slope = -(PipelineWidth \times PipelineEfficency)$$

Thus for a perfect single wide pipeline (no data, control or resource hazards) with no multi-cycle stages the IPC would be 1 (slope of -1). The display will show the areas of low performance by a slope that becomes less steep (a more horizontal line), and areas of high performance with a steep slope.

The GUI also allows users to select instructions for more information. Selecting an individual instruction displays the cycle time of execution and the instruction mnemonic. This makes it possible to get information about single instructions when the pipeline display is too small to label each individual instruction. Similarly, the resource view allows resource graph lines to be selected, which returns the label, cycle number and instantaneous value. Since the resource graphs are displayed as continuous lines from discrete data in the pipetrace file, intermediate points are calculated by linear interpolation.

2.2 Pipetrace File Format

Figure 3 illustrates an example of the pipetrace file structure. Each new cycle is marked with the "@" character. During a cycle, the changes to the pipeline are tracked with the "+","-" and "*" symbols. The plus sign indicates that a new instruction has entered the pipeline. The rest of the line provides the unique instruction number, PC, instruction attributes and assembly mnemonic of the new instruction. A minus sign indicates that an instruction has been removed from the pipeline. It should be mentioned that an instruction can be removed from the pipeline for reasons besides retirement (such as being squashed due to a branch misprediction or micro-op removal), therefore the "-" sign does not imply that the instruction ever entered the commit stage. The asterisk symbol, "*" indicates that the status of the instruction has changed. The rest of the line displays the instruction number, events that are occurring (such as cache misses), the latency of the longest event, and which event to color if multiple events are occurring. At the end of each cycle, tracked statistics are listed with a less then sign "<" and a greater then sign ">" on the left and right of the variable name, respectively. GPV accepts the value of the statistic in both integer and floating point format. Any of the statistics listed in a pipetrace file that begin with an "NT", signifying *no trace*, will be ignored by GPV when it parses the file. This allows the user to easily annotate the pipetrace file. We have found this format to be very flexible. For example, we have successfully interfaced GPV to a variety of simulators, including simulators running different instruction sets (ARM & Alpha)

2.3 Implementation Consideration

Although GPV takes generic text inputs, it was originally designed to work with the SimpleScalar tool set [18]. To this extent, two other Perl/Perl TK tools have been developed to assist in the running of SimpleScalar with GPV. A GUI front was included that contains fields for the simulator, execution script, simulator options, benchmark and a few other run parameters. Once filled in, this GUI calls a Perl script, which independently executes the program. This execution copies the benchmark (presently Spec95[9], Spec2000[10][11], Mediabench[12], and a few other benchmarks), benchmark inputs, and simulator to a

1. The negative sign is because instruction progress in the negative y direction.

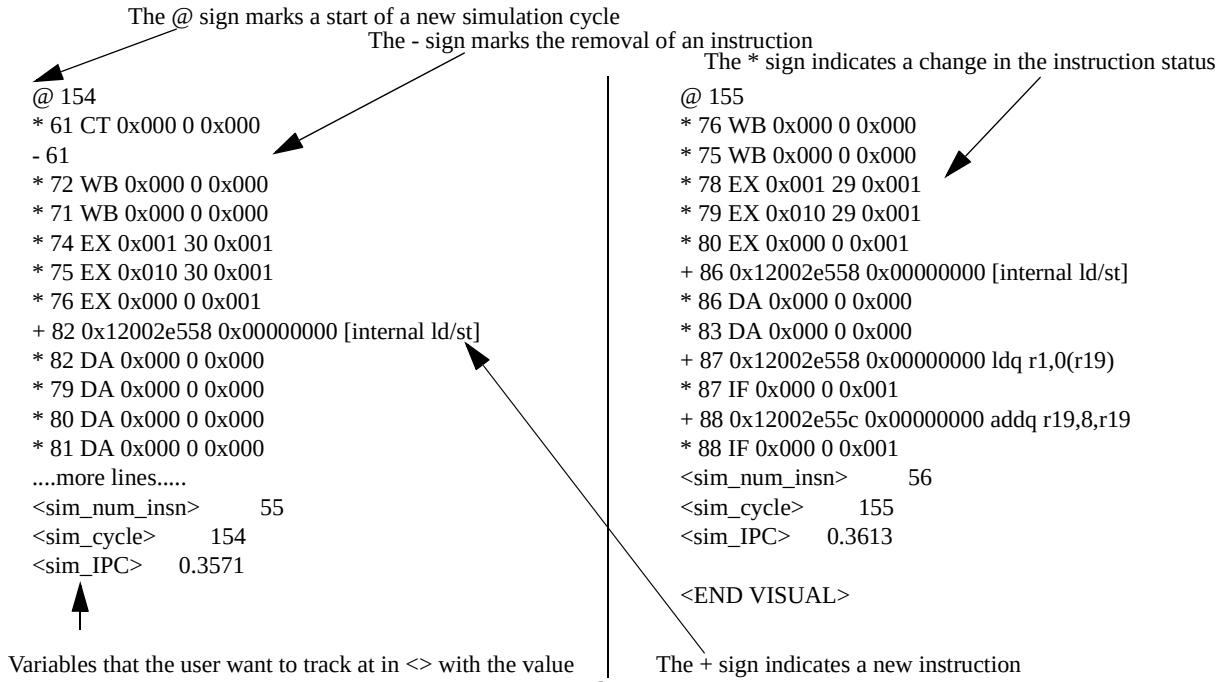

Figure 3 Sample pipetrace stream

directory where the simulation is performed. The execution of the simulation can be automatically piped into GPV. These tools make it possible for a novice user to start simulations using only graphical interfaces. The experienced user, on the other hand, benefits from the flexibility of launching simulations with or without GPV.

3. Conclusions

Visualization makes detailed comparisons of microarchitectural models expedient, simple and thorough. Visualization also simplifies the simulator verification process, by making the constraints (or lack of constraints) of the instruction flow readily apparent to the developer. The Graphical Pipeline Viewer (GPV) realizes the benefits in an easy to use and portable implementation.

4. Acknowledgements

We would like to thank Matt Postiff and Charles Lefurgy for their development of runspec, which is a precocious Perl script for running and simulating spec95[9], spec2000[10] and other various benchmarks. This script was adapted to run any of the supported benchmarks directly on GPV. When used in conjunction with the SimpleScalar frontend GUI a total windows based simulation environment is created.

This work was supported by the NSF CADRE program, grant no. EIA-9975286, and by an equipment grant from Intel.

References

[1] Intel. VTune: Visual Tuning Environment, 1997. http://developer.intel.com/design/perftool/vtune/index.htm.

[2] DLXView.[online] Available: <http://yara.ecn.purdue.edu/~teamaaa/dlxview/>, cited June 2001.

[3] J.L. Hennessy and D.A. Patterson, "Computer Architecture: A Quantitative Approach," Morgan Kaufmann, San Francisco, CA, 1996.

[4] A.R. Lebeck, “Cache Conscious Programming in Undergraduate Cmputer Science,” ACEM SIGCSE Technical Symposium on Computer Science Education, SIGCSE ‘99.

[5] A.R. Lebeck and David A. Wood, “Cache Profiling and the SPEC Benchmarks: A Case Study,” IEEE COMPUTER, 27(10):15-26, October 1994.

[6] Robert Bosch, Chris Stolte, Gordon Stoll, Mendel Rosenblum and Pat Hanrahan,”Performance Analysis and Visualization of Parallel Systems Using SimOS and Rivet: A Case Study,”Proceedings of the Sixth International Symposium on High-Performance Computer Architecture, January 2000.

[7] Robert Bosch, Chris Stolte, Diane Tang, John Gerth, Mendel Rosenblum, and Pat Hanrahan,”Rivet: A Flexible Environment for Computer Systems Visualization,” Computer Graphics 34(1), February 2000.

[8] Chris Stolte, Robert Bosch, Pat Hanrahan, and Mendel Rosenblum,”Visualizing Application Behavior on Superscalar Processors,”In Proceedings of the Fifth IEEE Symposium on Information Visualization, October 1999.

[9] J. Reilly, "SPEC Describes SPEC95 Products and Benchmarks," SPEC Newsletter, September 1995.

[10] "Standard Performance Evaluation Corporation (SPEC2000 CPU benchmark)". Accessible on the Internet at World Wide Web URL http://www.spec.org/osg/cpu2000/.

[11] B. Case. "SPEC2000 Retires SPEC92," The Microprocessor Report, vol. 9, 1995.

[12] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. “MediaBench: A tool for evaluating and synthesizing multimedia and communications systems,” In Proceedings of the International Symposium on Microarchitecture, pages 330--5, December 1997

[13] RSA Security. "RC6," http://csrc.nist.gov/encryption/aes/round2/AESAlgs/RC6, August 1999.

[14] Intel Corporation, "SA-110 Microprocessor Technical Reference Manual," ftp://download.intel.com/designstrong/manuals/27805801.pdf.

[15] Intel Corporation, "Intel StrongARM SA-110 Microprocessors Instruction Timing," ftp://download.in-tel.com/design/strong/applnots/27819401.pdf.

[16] Rebel.com NetWinder Family, http://www.rebel.com/netwinder.

[17] D. Kirovski, J. Kin and W. H. Mangione-Smith. "Procedure Based Program Compression," Proceedings of the 30th Annual International Symposium on Microarchitecture, December 1997.CPROF paper

[18] Doug Burger, Todd M. Austin and Steve Bennett. "Evaluating Future Microprocessors: The SimpleScalar ToolSet". University of Wisconsin-Madison. Computer Sciencies Department. Technical Report CS-TR-1308, July 1996.

[19] C. Weaver, K. Barr, E. Marsman, D. Ernst, and T. Austin. “Performance Analysis Using Pipeline Visualization”. http://www.eecs.umich.edu/~taustin/papers/gpvtech.pdf