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 continue to increase 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. We demonstrate through case studies how the tool can be applied to increase performance in a program with code and architectural modifications. [1]

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. There are a few key guidelines for making an architectural pipeline visualization tool:

- *Simple Generic Interface*. The visualization tools 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 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

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.

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

The next section discusses related work in performance visualization. We detail the strengths and weaknesses of each tool, and compare them to GPV. Section 3 details GPV, highlighting its features and internal implementation. Section 4 demonstrates the power of performance visualization through real case studies. Section 5 describes future developments planned for GPV, including GUI enhancements, pipetrace improvements and new visualization constructs. Finally, the paper describes conclusions derived from this work.

2. Related Work

Visualizing a microprocessor is not a new idea. Past works have ranged from pedagogical pursuits to comprehensive performance studies. This section will describe some of the existing visualization infrastructures to provide a context for our tool.

DLXview [2] is a tool that depicts the DLX pipeline that is outlined in Computer Architecture: A Quantitative Approach by John Hennessy and David Patterson [3]. It was created as part of the CASLE (Compiler/ Architecture Simulation for Learning and Experimenting) project at Purdue. While the tool does provide a detailed view of the pipeline at any time instance, it does not contain a facility for examining the performance of large segments of code. Additionally, the visualization tool is customized for the DLX architecture, which limits its application to other simulators. As can be ascertained from the project name (CASLE), DLXview is intended primarily for educational applications. GPV, which is aimed at performance analysis, can display larger segments of code by varying the zoom scale. The ASCII interface to GPV is generic enough to allow it to be interfaced to other simulators.

Another common method for visualizing the performance of a simulator is to abstract away the architecture and provide statistics based on the actual code running. CPROF [4][5] and VTUNE[1] are two examples of programs that display information such as cache misses or branch mispredictions for specific segments of code. This environment provides an ideal framework for directing program optimization efforts, however, it doesn't make all microarchitecture effects apparent. For example, if you have a long chain of high latency dependent instructions (such as multiplies), the slow down due to these instruction is not apparent on a tool that does not display the flow of instructions through the pipeline. We give an example of how GPV can be used to diagnose this type of problem in the case studies section.

RIVET [6-8] is a powerful display environment developed at the Stanford Computer Graphics Laboratory. The tool provides a very detailed time line view to identify problem areas. This view uses multiple levels of selection to gradually decrease the area of code being viewed, while simultaneously increasing the detail. Very small segments of this code can be animated to see the flow of the instructions through the pipeline. Instructions can also be traced back to the source code, in a fashion similar to CPROF and VTUNE. GPV provides most of the powerful facilities found in RIVET, plus additional support to aid in microarchitectural visualization. We also address some of the flexibility issues found in RIVET. First, the code in RIVET is written using C++ and OpenGL, which may lead to platform problems. GPV is built using the Perl TK libraries which have been ported to most operating systems, including Linux, Unix and Windows. Second, RIVET cannot display multiple instruction traces at once or time-varying pipeline traces, which are essential for diagnosing bottlenecks. GPV allows multiple pipeline traces to be displayed at once for easy contrast. Finally, RIVET is optimized for the MXS processor model, which could limit its applicability to other simulator models. GPV is not linked to any particular simulator instruction set making it easy to port it to new simulator models. GPV also employs a pipeline trace visualization construct, which we have found is more efficient then pipeline animation at identifying performance bottlenecks, because the representation captures into a single display the interactions between instructions and resources over time.

3. Graphical Pipeline Viewer

Figure 1 gives an overview of the 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

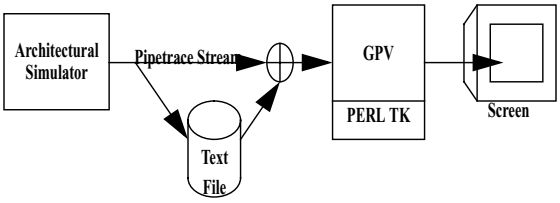

Figure 1 Overview the GPV usage flow

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 that GPV has been designed to communicate with.

3.1 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 space (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 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

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)

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 (which can be configured by the user) 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 statistic that is most valuable for performance analysis and correlate the variable 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.

3.2 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 the line with 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 statistical values 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 istruction sets (ARM and Alpha).

3.3 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

1.Tthe negative sign is because instructions progress in the negative y direction

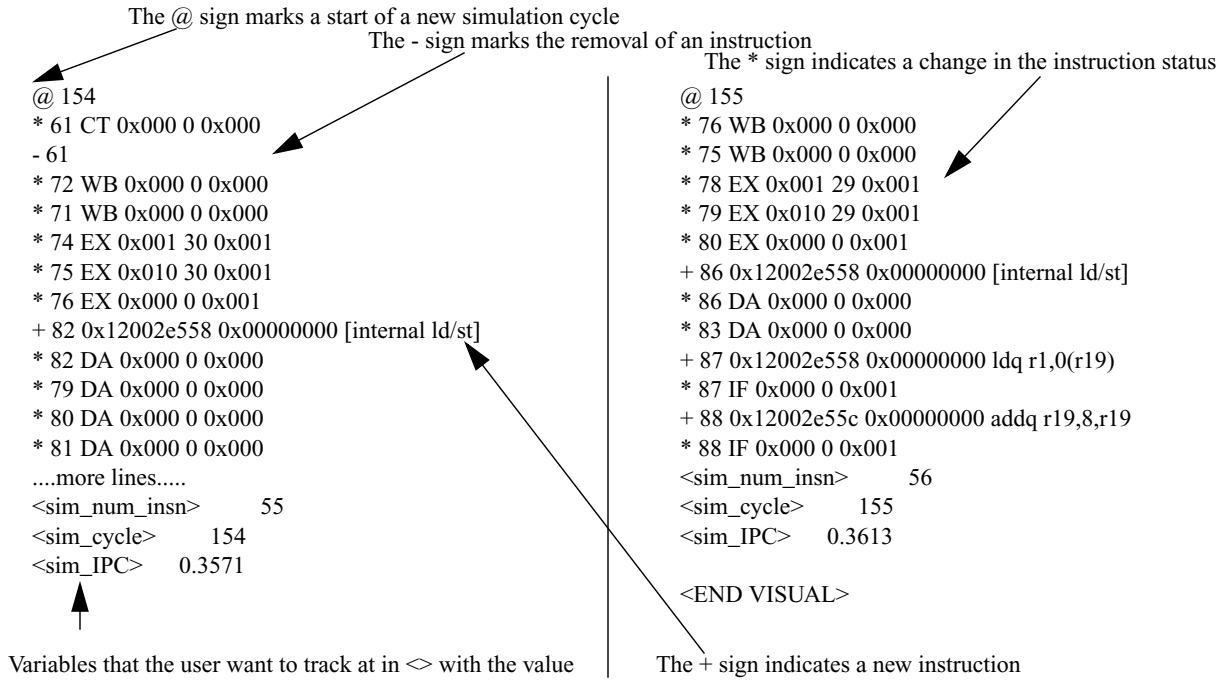

Figure 3 Sample pipetrace stream

for the simulator, execution script, simulator options, benchmark and a few other run parameters. Once filled in, this GUI calls a perl script which can be used independently, to execute 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 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.

4. Case Studies

To demonstrate the value of visualization-directed performance optimization, GPV was applied to the analysis and optimization of the RC6 [13] cryptographic algorithm. This cryptographic kernel was designed by RSA security; it can be found in use in many applications. To do the analysis the SimpleScalar/ARM target was used to generate the GPV pipetraces. These traces were then examined with GPV to locate program bottlenecks. First, we looked at software modifications that were possible to increase performance. Second, we examined what microarchitectural changes could improve the performance of the cipher. Finally, we noted how visualization can be used for simulator validation. This section will provide details on the simulator and configuration used to produce the case studies. An explanation of the optimizations discovered and example graphs proceed the simulation environment description. A demonstration of how GPV can be used for simulator validation is also studied.

4.1 Simulation Environment

The configuration used for the simulation, shown in Table 1, is modeled after Intel's SA-1 StrongARM pipeline [14], found in the SA-11xx series of embedded microprocessors. Intel has released few details of the SA-1 pipeline; our model was constructed using pipeline timing characteristics given in the SA-110 compiler writers' guide [15]. In addition, we used microbench-

Microprocessor Component	Value
Fetch Queue	2
Branch Predictor	Not-taken
Fetch and Decode Width	1
Issue Width	1 (In-order)
Functional Units	1 int ALU, 1 FP multi, 1 FP ALU
Instruction L1 Cache	16K, 32-way
Data L1 Cache	16K, 32-way
L2 Cache	None
Memory (bus width, first block latency)	4-byte, 12cycle

Table 1: Desciption of the Intel SA-1 StrongARM

```
int
rc6core(unsigned int *S, char *fmt, unsigned int ntrials,
        unsigned int Ain, unsigned int Bin, unsigned int Cin, unsigned int Din)
{
  unsigned int trial;
  unsigned int A = Ain, B = Bin, C = Cin, D = Din;
  unsigned int thirtytwo = 32;
  for (trial=0; trial < ntrials; trial++)
    {
      int ii;
      unsigned int t, u, x, w, tmp;

      for (ii=38; ii >= 4; ii-=2)
        {
          x = (D+D+1);
          w = (B+B+1);
          t = x*D;
          u = w*B;
          t = CONST_ROTL(t, 5);
          u = CONST_ROTL(u, 5);
          C -= S[ii];
          A -= S[ii+1];
          C = ROTR(C, u)^t;
          A = ROTR(A, t)^u;
          if (ii==4)
            { tmp = A; A = B; B = C; C = D; D = tmp; }
          else
            { tmp = A; A = D; D = C; C = B; B = tmp; }
        }
    }
  return A^B^C^D;
}
```

A)

```
int
rc6core(unsigned int *S, char *fmt, unsigned int ntrials,
        unsigned int Ain, unsigned int Bin, unsigned int Cin, unsigned int Din)
{
  unsigned int trial;
  unsigned int A = Ain, B = Bin, C = Cin, D = Din;
  unsigned int thirtytwo = 32;
  for (trial=0; trial < ntrials; trial++)
    {
      int ii;
      unsigned int t, u, x, w, tmp;

      for (ii=38; ii >= 6; ii-=2)
        {
          x = (D+D+1);
          w = (B+B+1);
          t = x*D;
          u = w*B;
          t = CONST_ROTL(t, 5);
          u = CONST_ROTL(u, 5);
          C -= S[ii];
          A -= S[ii+1];
          C = ROTR(C, u)^t;
          A = ROTR(A, t)^u;
          { tmp = A; A = B; B = C; C = D; D = tmp; }
        }
        {
          x = (D+D+1);
          w = (B+B+1);
          t = x*D;
          u = w*B;
          t = CONST_ROTL(t, 5);
          u = CONST_ROTL(u, 5);
          C -= S[4];
          A -= S[5];
          C = ROTR(C, u)^t;
          A = ROTR(A, t)^u;
          { tmp = A; A = B; B = C; C = D; D = tmp; }
        }
    }
  return A^B^C^D;
}
```

B)

Figure 4 A) Original RC6 code B) Loop Peel Optimization 1

marks to accurately measure fully exposed pipeline latencies such as branch mispredictions and cache misses. We validated our model against a Rebel NetWinder Developer workstation [16]. The NetWinder contains a 275 MHz StrongARM SA-110 microprocessor, 128 MB of DRAM, and an Ethernet interface. It runs the Linux operating system (version 2.2.13) with a standard GNU tool chain including GCC (version 2.95.1).

The run times of integer microbenchmarks, kernels (e.g., FFT), and large benchmarks (e.g., bzip and GCC) were measured on the NetWinder and compared to their simulated performance on the SA-1 ARM model. The simplicity of the SA-1 pipeline and memory system permitted us to construct an extremely accurate timing model with only a few modifications to the SimpleScalar/ARM performance simulator. The largest measured error in performance (CPI) was only 3.2%

4.2 Software performance analysis

Since we are working with the SA-1 ARM model, every branch must be placed under close scrutiny because this embedded processor employs a simple not-taken branch predictor. We chose to focus our attention on the branches that are in the cipher kernel, shown in Figure 4a. While the program does have a small startup cost, for large iterations the code in Figure 4 constitutes the bulk of the execution time. As shown in the visualization of Figure 5A, there are significant number of branch mispredictions in the cipher kernel. To address the mispredictions, we first sliced off the last iteration of the loop so that the if/else statement inside of the loop could be removed (see Figure 4b). The loop peeling optimization, shown in Figure 5B, resulted in a 15% speedup, due to approximately a 40% decrease in branch mispredictions. To remove the remaining branch mispredictions, we unrolled the entire cipher loop. Figure 6 shows the code for the second optimization. With the removal of virtually every branch misprediction, we obtained a 24% speedup (over the original code). Figure 5C shows that inside the loop all of the branch mispredictions have been removed.

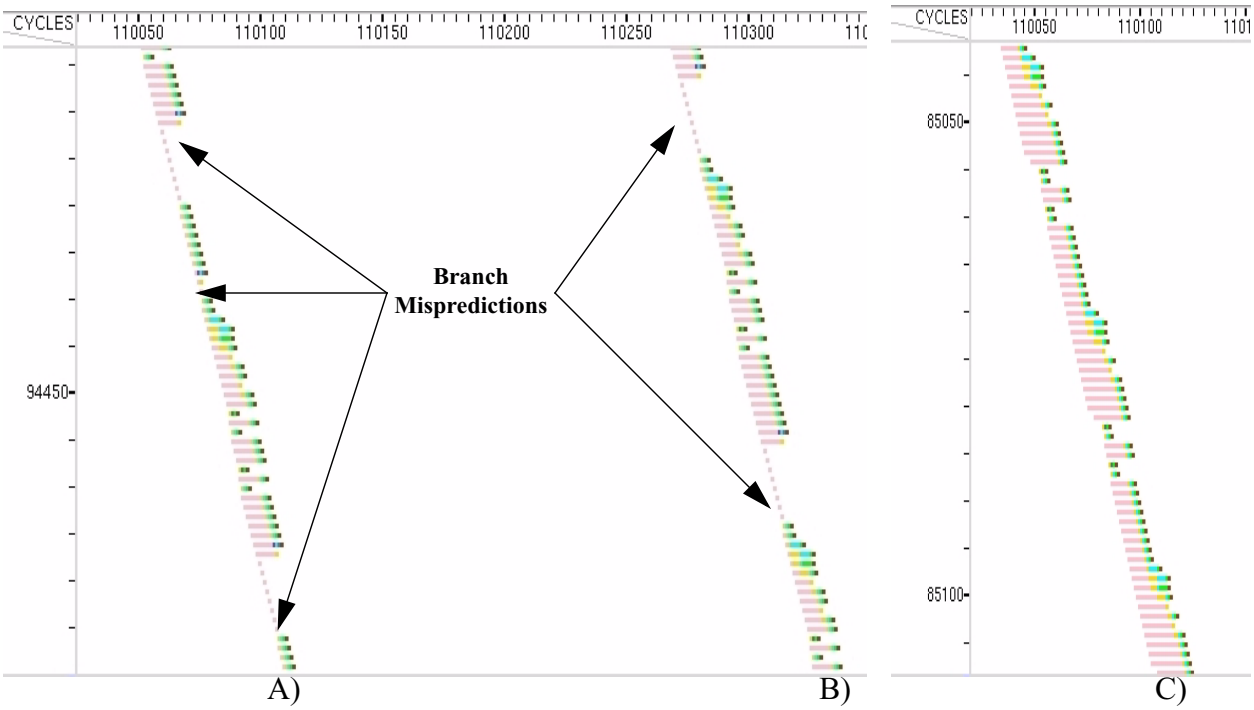

Figure 5 A) Shows the baseline implementation, B) Show the loop peeling optimization, C) shows the performance after the loop unrolling code optimization

4.3 Hardware performance analysis

Further analysis of the RC6 kernel reveals that there are two back-to-back independent multiplies. If these long latency instructions were to executed in parallel a large runtime savings could be obtained. To demonstrate this optimization, we re-configured the simulator to use two integer multipliers (instead of one) and re-simulated the code. As shown in Figure 7a, the two traces are identical until they reach the beginning of the *rc6core()* function call. Once inside the function, the two traces begin to diverge as the model with two multipliers is able to more rapidly process instructions (as denoted by the higher slope pipeline trace). The zoomed view, Figure 7b, illustrates the resource hazards experienced by the pipeline with one multiplier. The second multiplier adds a 9% speedup over the single multiplier model on the second optimization of the code. When both hardware and software optimizations are applied a speed up of 30% is obtained over the original program execution.

As mentioned earlier, architectural optimization may result in several system effects, some good and perhaps some bad. In this case study, adding a second multiplier produced better performance, at the cost of more

```
#define ROUND(II)                                   \
      {                                             \
       x = (D+D+1);                                 \
       w = (B+B+1);                                 \
       t = x*D;                                     \
       u = w*B;                                     \
       t = CONST_ROTL(t, 5);                        \
       u = CONST_ROTL(u, 5);                        \
                                                    \
       C -= S[II];                                  \
       A -= S[II+1];                                \
       C = ROTR(C, u)^t;                            \
       A = ROTR(A, t)^u;                            \
                                                    \
       { tmp = A; A = B; B = C; C = D; D = tmp; }   \
      }

int
rc6core(unsigned int *S, char *fmt,
        unsigned int ntrials,
        unsigned int Ain, unsigned int Bin,
        unsigned int Cin, unsigned int Din)
{
  unsigned int trial;
  unsigned int A = Ain, B = Bin, C = Cin, D = Din;
  unsigned int thirtytwo = 32;
  for (trial=0; trial < ntrials; trial++)
    {
      int ii;
      unsigned int t, u, x, w, tmp;
      ROUND(38);
      ROUND(36);
      ROUND(34);
      ROUND(32);
      ROUND(30);
      ROUND(28);
      ROUND(26);
      ROUND(24);
      ROUND(22);
      ROUND(20);
      ROUND(18);
      ROUND(16);
      ROUND(14);
      ROUND(12);
      ROUND(10);
      ROUND(8);
      ROUND(6);
       {
        x = (D+D+1);
        w = (B+B+1);
        t = x*D;
        u = w*B;
        t = CONST_ROTL(t, 5);
        u = CONST_ROTL(u, 5);
        C -= S[4];
        A -= S[5];
        C = ROTR(C, u)^t;
        A = ROTR(A, t)^u;
        { tmp = A; A = B; B = C; C = D; D = tmp; }
       }
    }
  return A^B^C^D;
}
```

Figure 6 Code optimization two (unroll loop)

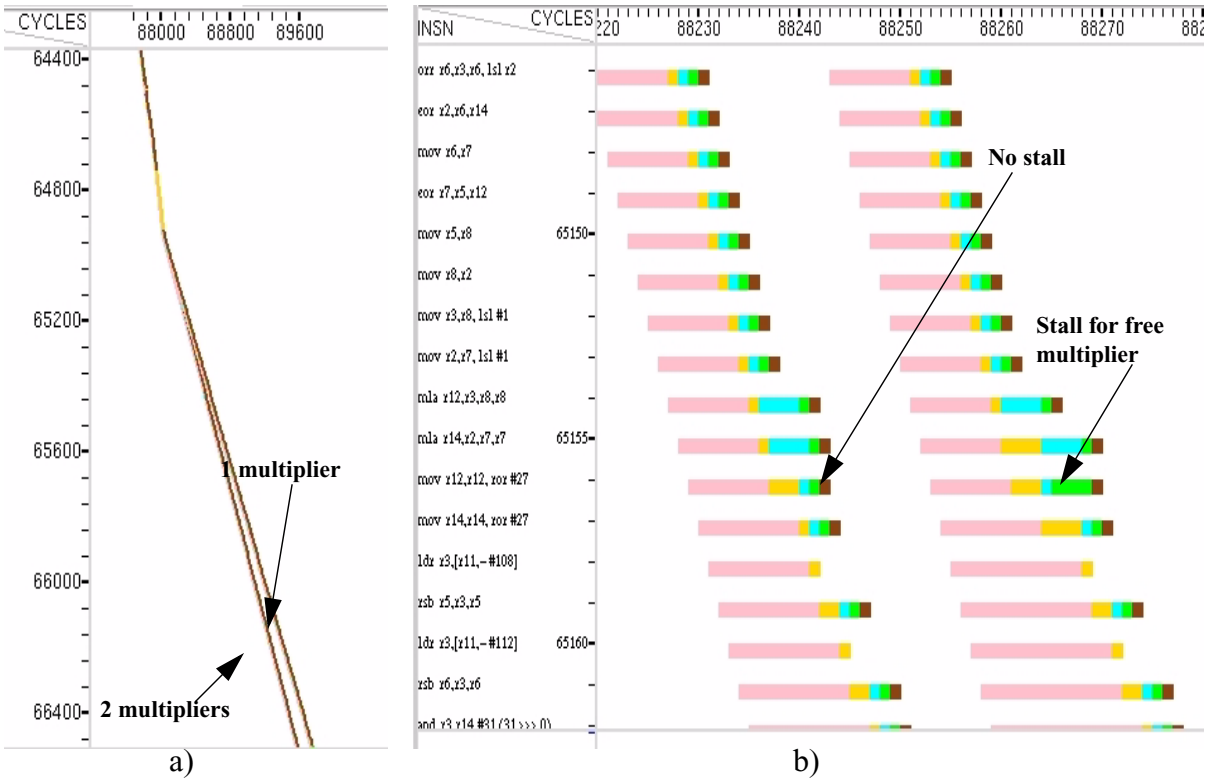

Figure 7 Performance Comparison with one vs. two multipliers

power dissipation. We modified the Wattch [17] power model, based on the SimpleScalar toolset [18], to output the power dissipated both by the whole processor and just the multipliers, labeled as *instant_power_3* and *instant_alu_power* respectively. Wattch includes several different power estimates, which are differentiated by how they estimate the power consumed by non-active elements. We chose to use the aggressive, non-ideal model which includes power estimates from both active and non-active transistors. For this reason, the aggressive non-ideal model shows the greatest difference between the one and two multiplier configurations. Fig-

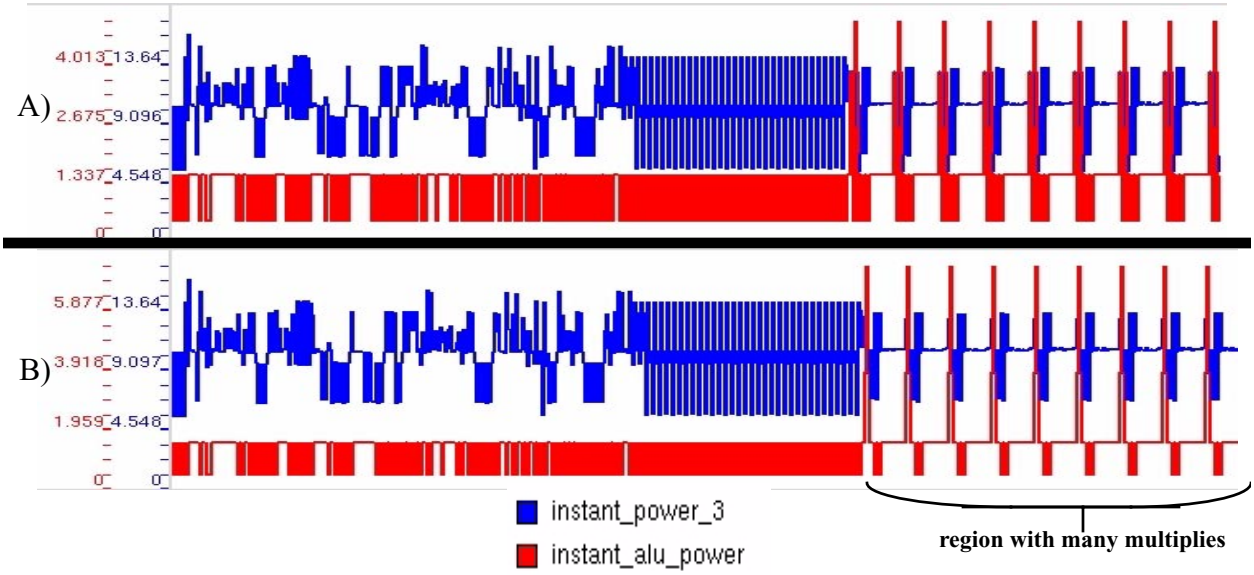

Figure 8 Power Consumption with A) one multiplier B) two multipliers. (*notice that the scales are different)

ure 8 shows the power usage of these models plotted on the resource view for a segment of the code with both optimizations. In the region of code where there are many multiplies, it is easy to see that the two multiplier configuration has higher instantaneous ALU power (notice that the axes have different values). It can also be observed that the single multiplier has wider lines. This is to be expected because the two multiplier configuration burns more power but in a shorter period of time. This also correlates to the expectation that the same amount of energy, minus leakage, will be expelled by the functional units in both configurations.

4.4 Using GPV for Simulator Validation

Visualization of a pipeline makes it possible to easily correlate events. From this view the constraints of the flow can be easily ascertained. This makes a visualization environment a powerful tool to validate the correctness of the simulator. For example, in sim-outorder (the out-of-order superscalar processor simulator of the SimpleScalar toolset) we observed that some instructions were issuing early under the presence of an Icache miss and a branch misprediction.

Original code:

```
ruu_fetch_issue_delay = ruu_branch_penalty;
```

Corrected code:

```
/* also look to see if there is already a issue_delay*/
ruu_fetch_issue_delay = MAX(ruu_branch_penalty,ruu_fetch_issue_delay);
```

What was occurring was that the fetch delay was being assigned the branch penalty latency rather than the max of the two latencies. While this can be considered a modeling problem, since some caches allow accesses to be cancel, it was not the behavior we were expecting. GPV can make this and other sequencing problems very apparent.

5. Future Enhancements

We are still working on GPV, adding more features and improving the robustness to the design. We have three main goals in this development work. We would like to increase the analysis ability of the GUI, increase the speed while decreasing the memory usage, and finally add more visualization constructs.

To extend the analysis ability we are planning to add measurement features inside of the tool. The user will be able to click two points and obtain the cycle count difference, instruction count difference, and number of each type of tracked event in the range. The user will also be able to enter formulas, such as instructions per cycle (IPC), that the tool will evaluate for the clicked range.

To increase the speed of the tool, which operates inefficiently when run remotely (i.e. the visualization and simulation run on different machines), we plan to change the display algorithm used. Presently, the whole trace is rendered and then display boundaries are setup. This causes slow performance for large traces. We plan to sped this process by only rendering what would be visible given the current display window. Similarly, pipetrace files can become large very since they are in ASCII notation. To mitigate this problem, we are investing optimized pipeline stream interfaces. The simplest method of doing this is to just produce a trace file that is compressed. By doing this the ASCII stream can be compressed during creation and then decompressed during use, thereby reducing communication time. We have already added profiling features to SimpleScalar to identify key sections of code where performance bottlenecks lie. We would like to extend these profiling abilities to a graphical overview of the program. Another simulator enhancement for GPV, is to permit a marked instruction or sequence of instructions to activate the pipetrace. We have already built a simulator that initiates pipetraces when the simulated program executes a marker instruction.

While we do have a fairly flexible display system, it does lack a good memory visualization capability. We plan to create a separate window that uses multiple color coded scatter plots to show the number of times a memory location was accessed and when. The number of times accesses to that memory location caused cache misses will also be displayed. This type of plot should make it easy to see where memory congestion is located. Additionally we would like to add a PC-based display similar to those used by CPROF and VTUNE. Since SimpleScalar already has a pc-based statistical gathering function built into it, this feature should be fairly straightforward to implement. We would also like to continue the evolution of the Wattch power model, and add a display that illustrates power dissipation in each unit of the processor. While this can be done in the resource view, as shown in this paper, making a separate display optimized for this view (perhaps on a die-photo) would be more efficient for the visualization of the vast number of power statistics that are recorded.

6. Conclusions

We have shown, through the use of case studies, that microarchitectural visualization can be a powerful tool for locating and correcting software bottlenecks. Visualization also makes detailed comparisons of microarchitectural models expedient, simple and thorough. Finally visualization simplifies the simulator verification process, by making the constraints (or lack of

constraints) of the instruction flow readily apparent to the developer.

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[], spec2000[] 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.intel.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.