Visualizing Application Behavior on Superscalar Processors

Chris Stolte, Robert Bosch, Pat Hanrahan, and Mendel Rosenblum
Computer Science Department
Stanford University

Abstract

The advent of superscalar processors with out-of-order execution makes it increasingly difficult to determine how well an application is utilizing the processor and how to adapt the application to improve its performance. In this paper, we describe a visualization system for the analysis of application behavior on superscalar processors. Our system provides an overview-plus-detail display of the application's execution. A timeline view of pipeline performance data shows the overall utilization of the pipeline, indicating regions of poor instruction throughput. This information is displayed using multiple time scales, enabling the user to drill down from a high-level application overview to a focus region of hundreds of cycles. This region of interest is displayed in detail using an animated cycle-by-cycle view of the execution. This view shows how instructions are reordered and executed and how functional units are being utilized. Additional context views correlate instructions in this detailed view with the relevant source code for the application. This allows the user to discover the root cause of the poor pipeline utilization and make changes to the application to improve its performance.

This visualization system can be easily configured to display a variety of processor models and configurations. We demonstrate it for both the MXS and MMIX processor models.

Keywords: Computer systems visualization, visualization systems, superscalar processors.

1 INTRODUCTION

The processing power of microprocessors has undergone unprecedented growth in the last decade [5]. Desktop computers produced today outperform supercomputers developed ten years ago. To achieve these performance enhancements, mainstream microprocessors such as the Intel Pentium Pro [11] and the MIPS R10000 [15] employ complex pipelines with out-of-order execution, speculation and rename registers.

The implementation techniques used by these processors are intended to be invisible to the programmer. This is true from the standpoint of correctness: application writers need not know the details of the processor implementation to write code that executes correctly. In order to write code that runs *well*, however, programmers need an understanding of their applications' interactions with the processor pipeline. While optimizing compilers aid in producing compiled code that can take advantage of these powerful processors, they are unable to leverage semantic knowledge about the application in performing their optimizations. Changes made to the code structure of an application by the programmer can increase the instruction-level parallelism that a processor can exploit, resulting in increased performance.

However, because of the complexity of these processors, few software developers understand the interactions between their applications and the processor pipeline. The analysis of application behavior on superscalar processors is complicated by several factors:

- **Having to look at the details.** Many different events can cause poor utilization of the processor pipeline: contention for functional units, data dependencies between instructions, and branching are examples. High-level statistics can indicate that these hazards exist within an application, but they cannot indicate when, where, or why these events are occurring. Understanding specific hazards requires a detailed examination of pipeline behavior at the granularity of individual instructions.
- **Having to know where to look.** Modern processors can execute hundreds of millions of instructions in a single second. Therefore, it is not feasible to browse through either a trace file or detailed visualization of an application's entire execution searching for areas of poor performance. High-level performance overviews of the execution are required to identify regions of interest before detailed visualizations can be used for analysis.
- **Having to have context.** Most programmers think in terms of source code, not in terms of individual instructions. In order to modify their applications to enhance performance, programmers need to be able to correlate instructions in the pipeline with the application's source code.

We have developed a visualization system that addresses all of these issues. Our system consists of three displays: a timeline view of pipeline performance statistics, an animated cycle-by-cycle view of instructions in the pipeline, and a source code view that maps instructions back to lines of code. These views combine to provide an overview-plus-detail [13] representation of the pipeline, enabling the effective analysis of applications. A programmer can utilize the timeline view to observe the time-varying behavior of the pipeline and identify regions of execution where events of interest (such as poor pipeline utilization) occur. The detailed pipeline view can then be used to display and animate the flow of instructions through the pipeline, providing an understanding of why the pipeline is stalling. Finally, the source code view can be used to correlate the problematic instruction sequences with source code, where changes may be made to improve application performance. In addition to program analysis, this visualization system is also useful for several other tasks, including compiler analysis, hardware design, and processor simulator development.

The flexibility of our system enables us to visualize several different processor models, as well as a variety of configurations of a particular processor model. In this paper, we include visualizations of the MXS [1] simulator and two configurations of the MMIX [8] processor model. While our current focus is on the study of processor pipelines, this system could be extended to display other types of pipelines, such as manufacturing assembly lines and graphics pipelines.

2 RELATED WORK

Although there are many systems available for high-level analysis of application performance, there are few systems available for this detailed visualization of application execution on superscalar processors. Existing systems include DLXview [3], VMW [2], BRAT [12], and the Intel Pentium Pro tutorial [11].

DLXview [3], an interactive pipeline simulator for the DLX instruction set architecture [5], provides a visual, interactive environment that explains the detailed workings of a pipelined processor. Performance evaluation is a secondary goal of their system: their focus is on the pedagogical nature of visualization. For performance analysis purposes, the pipeline displays of DLXview provide too much detail without enough overall context.

The Visualization-based Microarchitecture Workbench (VMW) [2] is a more complete system for the visualization of superscalar processors. This system was developed with the dual goals of aiding processor designers and providing support to software developers trying to quantify application performance. However, there are several disadvantages to the visualizations and animation techniques used by the system. VMW provides very limited high-level information on application performance, and it is difficult to correlate this information with the detailed views. Animation is used to depict cycle-by-cycle execution, but the animation is not continuous – it consists of sequential snapshots of processor state. While we initially used this approach, we found this animation technique was both difficult to follow and detrimental to understanding the instruction flow.

During the development of the PowerPC, IBM used a simulation tool called the Basic RISC Architecture Timer (BRAT) [12] to study design trade-offs. BRAT provides a graphical interface that allows the user to step through trace files, displaying the processor state at each cycle. BRAT provides only the single, detailed view of the processor state and does not utilize animation in the visualization. Like VMW, this visualization is tightly integrated with the simulator and thus not a general-purpose tool.

Intel distributes an animated tutorial [11] that illustrates the techniques the Pentium Pro processor utilizes to improve performance. Similar to DLXview, the pedagogical intent of this tutorial has resulted in a different design than our system. The tutorial provides a limited cycle-by-cycle view of the instructions in the pipeline with explanatory annotations. No contextual performance data or source code displays are provided.

3 BACKGROUND

To provide a context for our visualization, we begin by describing the salient characteristics of superscalar processors that impact application performance. We first introduce the major techniques that superscalar processors use to improve performance, and then explain the types of events that can cause a processor pipeline to be underutilized.

Given a fixed instruction set architecture, a reasonable measure of a processor's performance is the throughput – that is, the number of instructions that complete execution and exit the pipeline in a given period of time. Modern microprocessors utilize several techniques to improve their throughput:

- **Pipelining.** Pipelining overlaps the execution of multiple instructions within a functional unit, much like an assembly line overlaps the steps in the construction of a product. For example, a single-stage floating point unit might require 60 cycles to complete execution of a single divide instruction. If this functional unit were pipelined into six stages of 10 cycles apiece, the unit would be able to process multiple divide instructions at once (with each stage working on a particular piece of the computation). While it would still require 60 cycles to compute a single divide, the pipelined functional unit would produce a result every 10 cycles when performing a series of divide instructions.
- **Multiple Functional Units.** Superscalar processors include multiple functional units, such as arithmetic logic units and floating-point units. This enables the processor to exploit instruction-level parallelism (ILP), executing several independent instructions concurrently. However, some instructions cannot be executed in parallel because one of the instructions produces a result that is used by the other. These instructions are termed *dependent*.
- **Out-of-Order Execution.** In order to improve functional unit utilization, many superscalar processors execute instructions out of order. This allows a larger set of instructions to be considered for execution, and thus exposes more ILP. Out-of-order execution can improve throughput if the next instruction to be sequentially executed cannot utilize any of the currently available functional units or is dependent on another instruction. Although instructions may be executed out of order, they must *graduate* (exit the pipeline) in their original program order to preserve sequential execution semantics. The reordering of instructions is accomplished in the *reorder buffer*, where completed instructions must wait for all preceding instructions to graduate before they may exit the pipeline.
- **Speculation.** Rather than halting execution when a branch instruction is encountered until the branch condition is computed, most processors will continue to fetch and execute instructions by predicting the result of the branch. If the processor speculates correctly, throughput is maintained and execution continues normally. Otherwise, the speculated instructions are *squashed* and their results are discarded.

Despite the use of these techniques, superscalar processors are often unable to achieve maximum throughput. There are many possible causes for underutilization of the pipeline.

When there are not enough functional units to exploit the ILP available in a code sequence, instructions must wait for a unit to become available before they can execute. These *structural hazards* often occur in code that is biased towards a particular type of instruction, such as floating-point instructions. The functional unit for those instructions will be consistently full, and the other units will often remain empty for lack of instructions. Consequently, the throughput of the processor is limited to the throughput of the critical functional unit alone.

Dependencies prevent instructions from executing in parallel. Out-of-order execution enables the pipeline to continue execution in the face of individual dependencies; however, if a code sequence includes enough dependencies, the lack of ILP will limit pipeline throughput.

Speculative execution can impact throughput in two ways. First, most processors cannot speculate through more than four or five branches at once. Once this *deep speculation* is reached, the processor cannot speculate through subsequent branch instructions. This forces the pipeline to stop fetching instructions until one of the pending branches is resolved. Second, processors do not always predict the result of a branch correctly. When *branch misprediction* occurs, throughput suffers since the incorrectly speculated instructions must be squashed from the pipeline.

Because main memory accesses often require hundreds of cycles to complete, *memory stall* can have a major impact on pipeline performance. When an instruction cache miss takes place, the processor cannot fetch instructions into the pipeline until the next sequence of instructions is retrieved from memory. The resulting lack of instructions in the pipeline reduces the processor throughput. Misses to the data cache increase the effective execution time of load and store instructions, since they must wait for the memory access to complete before they can graduate. This delays the execution of any dependent instructions, and eventually stalls the pipeline by preventing subsequent instructions from graduating.

Finally, some instructions, such as traps and memory barrier instructions, require *sequential execution*, forcing the pipeline to be emptied of all other instructions before they can execute. This has an obvious detrimental effect on throughput.

Although high-level visualizations can indicate that these events are occurring during the execution of an application, only detailed visualizations can reveal the instruction flow and dependencies that are responsible for these performance bottlenecks. This detailed knowledge is critical for adapting the application to improve the performance.

4 VISUALIZATION ENVIRONMENT

The pipeline visualization system discussed in this paper was developed using Rivet. Rivet is a visualization environment we are developing to support the rapid prototyping of visualizations for the exploration and understanding of real world problems, with an emphasis on the analysis and visualization of computer systems. Several attributes of Rivet were particularly important for the development of this visualization system.

Flexibility. Rivet provides flexibility through a compositional architecture. Components such as data objects and visual primitives, written in C++ and OpenGL, are designed to be *composed* to form objects with greater functionality. Primitives and objects also export an interface to a scripting language such as Tcl, which allows them to be composed further to create sophisticated, interactive visualizations.

One of our design goals for the pipeline visualization system was to make it easily adaptable to many processor models. To support this, our processor pipeline display is composed from two major classes of visual primitives, *containers* and *pipes*, both written in C++. We use the Tcl scripting language to combine these primitives into higher-level building blocks: the functional units, stages, and data paths of the processor pipeline. We then combine these objects according to the configuration of the particular processor model under study to represent the entire pipeline.

The decomposition of the pipeline into its constituent elements enables us to easily adapt the layout to represent a variety of processor models with different pipeline organizations. It also enables us to easily configure the visualization according to the parameters of a particular processor model, such as the number of functional units or the size of the reorder buffer.

Aggregation. The complexity of computer systems demands that a visualization environment be able to efficiently manage and display large data sets. To simplify this task, Rivet provides built-in data management objects that support data aggregation. Data is collected at a fine granularity over a long period; it is then built into an aggregation structure that includes both the raw data and smaller, less detailed aggregates. When displaying the data, Rivet chooses the appropriate data resolution based on the time window to be displayed and the available screen space.

The study of superscalar processors requires large volumes of data be collected and visualized. Our experimental runs often generate data for hundreds of thousands of execution cycles. For each cycle, information about any instruction that changes state must be collected. The use of Rivet aggregation structures enables us to explore this data from a high-level overview down to individual data elements.

Animation. Animation is a core service provided by Rivet. This support includes the ability to request timed callbacks to visual primitives and path interpolation for a variety of animation paths. The Rivet redraw mechanism supports incremental redraw of visual primitives, important for efficient animation of objects.

Animation is crucial for understanding the cycle-by-cycle behavior of the pipeline. Our original pipeline implementation simply displayed the state of the pipeline at a particular cycle with no visual transitions between cycles. Without the visual cues provided by animation, we found it very difficult to track instructions as they advanced through the pipeline.

5 PIPELINE VISUALIZATION SYSTEM

Our pipeline visualization combines three major components to provide an overview-plus-detail display of application execution. We first describe each of the components of the system, and then present an example showing how the system is used to understand application behavior.

5.1 Timeline View: Finding Problems

The first task in understanding the behavior of an application on a processor is to examine an overview of the application's execution to locate regions of interest. The *timeline view*, shown in Figure 1, utilizes a multi-tiered strip chart to display overall pipeline performance information at multiple levels of detail. The bottom tier shows data collected over the entire execution of the application. The user interactively selects regions of interest in each tier, which are expanded and displayed in the next tier. This visualization exploits the aggregation mechanism described in Section 4: the data in each tier is displayed at the highest resolution possible, determined by the number of horizontal pixels available for display.

The multi-tiered strip chart is used to indicate the reasons that the pipeline was unable to achieve full throughput on a particular cycle (or range of cycles in the aggregated displays). In a superscalar processor, throughput is lost whenever the pipeline fails to graduate a full complement of instructions from the pipeline in a given cycle. Because instructions must graduate in order, we attribute this pipeline graduation stall to the instruction at the head of the graduation queue (i.e. the oldest instruction in the pipeline). The reasons for failure to achieve full pipeline throughput can be classified into the following categories:

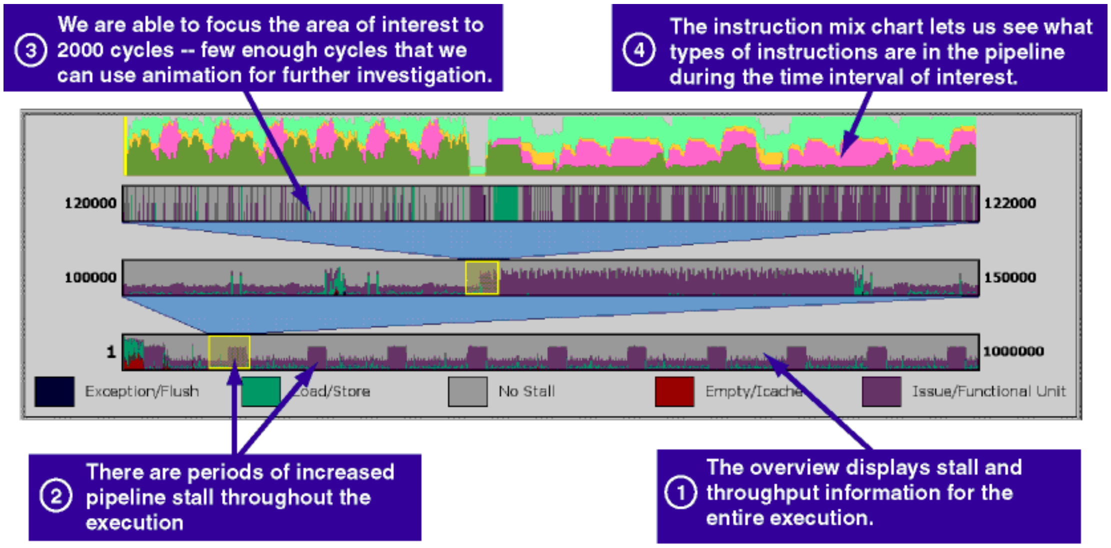

Figure 1: ***Investigation of an application's processor pipeline behavior typically begins by examining high-level performance characteristics. The timeline view provides a multi-tiered strip chart for the exploration of this data. Pipeline throughput statistics for the entire execution are shown on the bottom tier of the strip chart, with pipeline stall time classified by cause (as shown in the legend at the bottom of the window). The yellow panes are used to select time intervals of interest in each tier, which are displayed in more detail in the next tier. Directly above the multi-tiered chart is a simple strip chart that shows the instruction mix in the pipeline during the time region of interest: load/store (green), floating-point (pink), branch (yellow) and integer (cyan). This strip chart serves to relate this high-level view to the detailed pipeline view.***

- **Empty/Icache.** An instruction cache miss is preventing any instructions from being fetched from memory, so the pipeline is completely empty (and there is no head of the instruction queue to blame for the stall).
- **Exception/Flush.** Either an exception occurred or an instruction in the pipeline requires sequential execution. In either case, the pipeline must be flushed before continuing execution, again leaving the pipeline empty until instruction fetch resumes.
- **Load/Store.** The head of the graduation queue is a memory load or store operation that is waiting for data to be retrieved from the memory system.
- **Issue/Functional Unit.** The head of the graduation queue is either waiting to be issued into a functional unit due to a structural hazard or is still being processed by a functional unit.

In addition to the pipeline stall information, the timeline view includes a second chart that displays the mix of instructions in the pipeline. This chart classifies instructions by functional unit and shows the instruction mix during the same time window as the top tier of the multi-tiered strip chart. By relating the reasons for pipeline stall to the instructions in the pipeline, this display serves as a 'bridge' between the timeline view and the pipeline view.

5.2 Pipeline View: Identifying Problems

The pipeline view is illustrated in Figure 2. This visualization shows the state of all instructions in the pipeline at a particular cycle and animates instructions as they progress through the pipeline. The animation techniques used in this view are similar to those used for program debugging in [10] and algorithm animation in [14].

Pipeline stages and functional units appear as large rectangular regions with numerous instruction slots. Each stage is represented as a single container, with the number of slots indicating the capacity of the container. Functional units are composed of one or more containers, since these units may themselves be composed of multiple pipeline stages. The functional units are color-coded using the same color scheme as the instruction mix strip chart. The layout of the pipeline is interactively configurable. At any time, the user can reorder the layout of the functional units or resize the stages and functional units to focus on a portion of the processor pipeline.

Instructions in the pipeline are depicted as rectangular glyphs. The glyphs encode several pieces of information about the instructions in their visual representation. The fill color of the rectangle matches the color of functional unit responsible for execution of the instruction. The glyph contains text identifying the instruction; depending on the space available, either the opcode mnemonic or the full instruction disassembly (including both the mnemonic and the arguments) is displayed. The border color of the instruction conveys additional information. If the instruction has been issued speculatively and the branch condition is still unresolved, the border of the instruction is orange. If the instruction was issued as a result of incorrect speculation and will subsequently be squashed, it is drawn with a red border. The head of the graduation queue always has a yellow border, and a red triangle appears next to the text of this instruction.

Dependencies between instructions in the pipeline are displayed as yellow lines appearing between the two instructions. Since a large number of dependencies may be present in the pipeline, the user can selectively filter or disable this feature. To filter this display, the user selects with the mouse the instruction for which dependencies should be drawn.

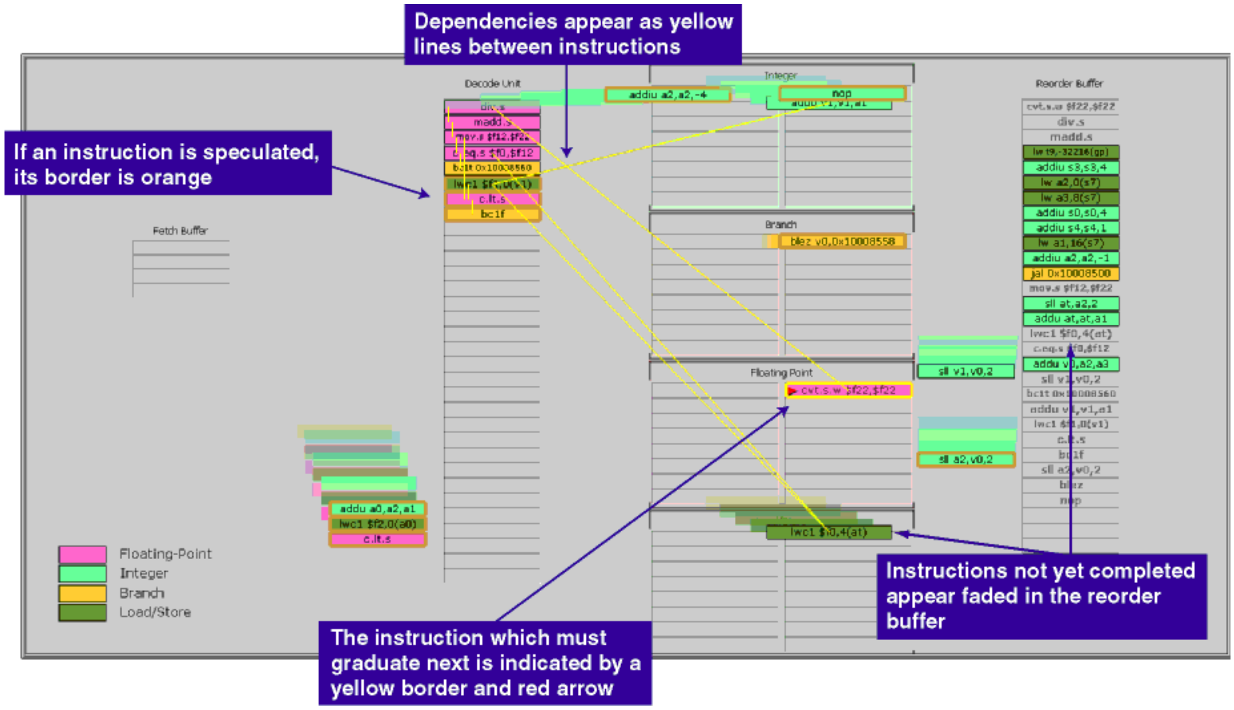

Figure 2: ***The pipeline view shows all instructions in the pipeline at a particular point in time. Pipeline stages and functional units appear as large rectangular regions with numerous instruction slots. This processor has a four-stage pipeline – fetch, decode, execute and reorder – arranged from left to right in the figure. Instructions are portrayed as rectangular glyphs, color-coded to indicate their functional unit and labeled to identify their opcode. Additional information about the state of the instruction is encoded in the border color of the glyph. User-controlled animation is used to show the behavior of instructions as they advance through the pipeline. This figure illustrates the animated transition between two cycles of execution of a graphics application executing on the MXS processor model. The pipeline can fetch and graduate up to four instructions per cycle. However, in this case the processor was unable to graduate any instructions because the head of the graduation queue is still being executed in the floating-point functional unit.***

With the exception of the reorder buffer, the pipeline stages order instructions by age: instructions enter at the bottom of the stage and move upward to replace instructions that have exited the stage. In the reorder buffer, instructions are shown in graduation order, with the head of the graduation queue at the top of the buffer. The reorder buffer leaves empty slots for instructions that are executing in the pipeline but have not yet completed. These slots contain a grayed-out text label of the instruction, enabling the slots to be correlated with the instructions in the pipeline.

The user controls the pipeline animation using controls similar to those used to control a VCR. The controls enable the user to single-step, animate, or jump through the animation. The animation may be run either forward or backward, and the speed is variable and under user control. The user can also use a vernier on the instruction mix strip chart to jump directly to a particular cycle.

This visualization can be used to understand the precise nature of the observed pipeline hazards. The user can animate through cycles of interest and visually identify for each cycle the hazards that are occurring. Figure C-1 (color plate) illustrates the visual characteristics of several of the major types of hazards. By interacting with the pipeline view, the user can observe the instruction sequences that are responsible for underutilizing the pipeline and understand the reasons for their poor performance. In order for this information to be useful, however, it must be related back to the source code of the application.

5.3 Source Code View: Providing Context

Once the major regions of poor performance and their causes have been discovered, the user must determine if the application can be altered to improve pipeline utilization. The final component of our system, the *source code view,* allows the user to correlate the high-level performance data and detailed animation views with the application's source code. This view is shown in Figure 3.

This visualization, modeled on the *SeeSoft* system [4], displays "bird's-eye" overview representations of the source files in the application. Each line of source code is represented by a single-pixel horizontal bar; the length and indentation of each bar is proportional to the actual indentation and length of the line in the source. The user can select a region of a source file in the overview to be displayed as full source code text in a separate window, as shown on the right side of the figure.

Both windows in the source code view highlight relevant lines of code. Lines that are executed at some point during the time window of interest in the timeline view are drawn in black, and lines that are being executed in the pipeline view are highlighted in red. As in the pipeline view, a red arrow is displayed next to the line of source code that contains the instruction at the head of the graduation queue.

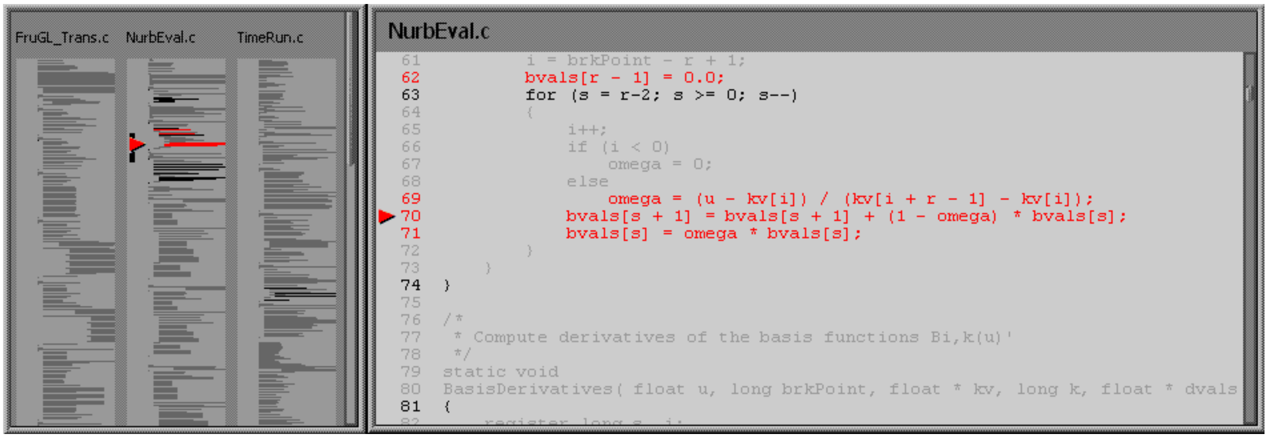

Figure 3: ***The source code view, which relates the instructions in the timeline's region of interest to the source code corresponding to these instructions. The left panel provides a bird's eye view of the source, and the right panel shows a portion of one of the source files (indicated by the vertical bar in the left panel). Both views are color-coded to highlight instructions in the region of interest. Black indicates that the line of code is executed somewhere in the timeline region and red indicates that the line is being executed in the detailed pipeline view. A red arrow indicates the instruction at the head of the graduation queue.***

5.4 Visualizing the MXS Pipeline

We now provide an example of how the three components of the system can be used together to understand the behavior of an application. For data collection, we use the MXS [1] superscalar processor model. This model implements the instruction set architecture used in the MIPS R10000 [15] processor. MXS has been incorporated into the SimOS complete machine simulator [6], enabling us to study the pipeline utilization of realistic workloads. In this example, we describe the analysis and visualization of a graphics application executing for one million cycles. Figure 4 and Figure C-2 (color plate) show a snapshot of the visualization of this data.

We begin the analysis by looking at the timeline view of the execution. The bottom tier of the multi-tiered strip chart shows a periodic execution pattern. Of interest are the phases where we see a significant increase in processor stall time. The chart shows that throughput is limited in these phases because the head of the graduation queue is still executing in a functional unit. There are several reasons why this might occur, such as an unbalanced mix of instructions or a large number of dependencies.

We use the multi-tiered strip chart to zoom in on the transition from high throughput to low throughput. As we zoom to a view of 50,000 cycles, the high-level pattern becomes less apparent but we can still see the two distinct phases of execution. We zoom further to a window of 2000 cycles centered on the phase transition. By comparing the throughput chart with the instruction mix chart, we discover that the bulk of the processor stall time corresponds to periods of heavy floating-point activity in the pipeline.

We investigate this further using the pipeline view. We animate this region of the execution and observe the instructions as they travel through the pipeline. The pipeline view shows a representative stage of the animation. By observing the animation, we are quickly able to see why the pipeline is suffering from poor throughput: there is a cascading dependency chain between nearly all of the instructions in the decode unit. This complete lack of instruction-level parallelism forces the pipeline to process instructions in a sequential fashion. Even worse, the instruction window is dominated by floating-point instructions, including operations with long execution latencies like the divide (the instruction in the floating-point unit in the figure). As a result, there are few (if any) instructions available to graduate per cycle.

We can use the source code view to correlate this pipeline behavior with the application's source code. We see in the source code that the application is executing a tight loop of floating point arithmetic. With this information, the programmer can now attempt to restructure the code to reduce the number of dependencies or interleave other code into the loop to better utilize the processor.

6 DISCUSSION

We have presented the pipeline visualization system in the context of one specific use – the understanding of application behavior on a superscalar processor. However, there are several other important uses for this visualization system.

Compiler Design. One of the major research areas in compiler design is code optimization. Research is being done to study the effectiveness of the code optimization techniques that have been developed and to discover and implement additional optimizations. In particular, with the popularity of superscalar processors, compiler writers are striving to maximize the amount of instruction-level parallelism in compiled code in order to make full use of processor resources. By exposing the detailed behavior of the processor pipeline, our visualization system can be used to study the effectiveness of compiler optimizations and suggest code sequences that would benefit from further optimization.

Hardware Design. When designing new processors, hardware architects need to understand the demands that applications used by their target markets will place on the processor. By using visualization to study the behavior of important commercial applications on existing processors, they can identify where architectural changes such as additional functional units or pipelining would be beneficial.

As a simple example of this use of our system, Figure 5 shows two visualizations of a prime number generator running on MMIX [8], an architecture being developed by Donald Knuth for his series of books, *The Art of Computer Programming* [9]. The

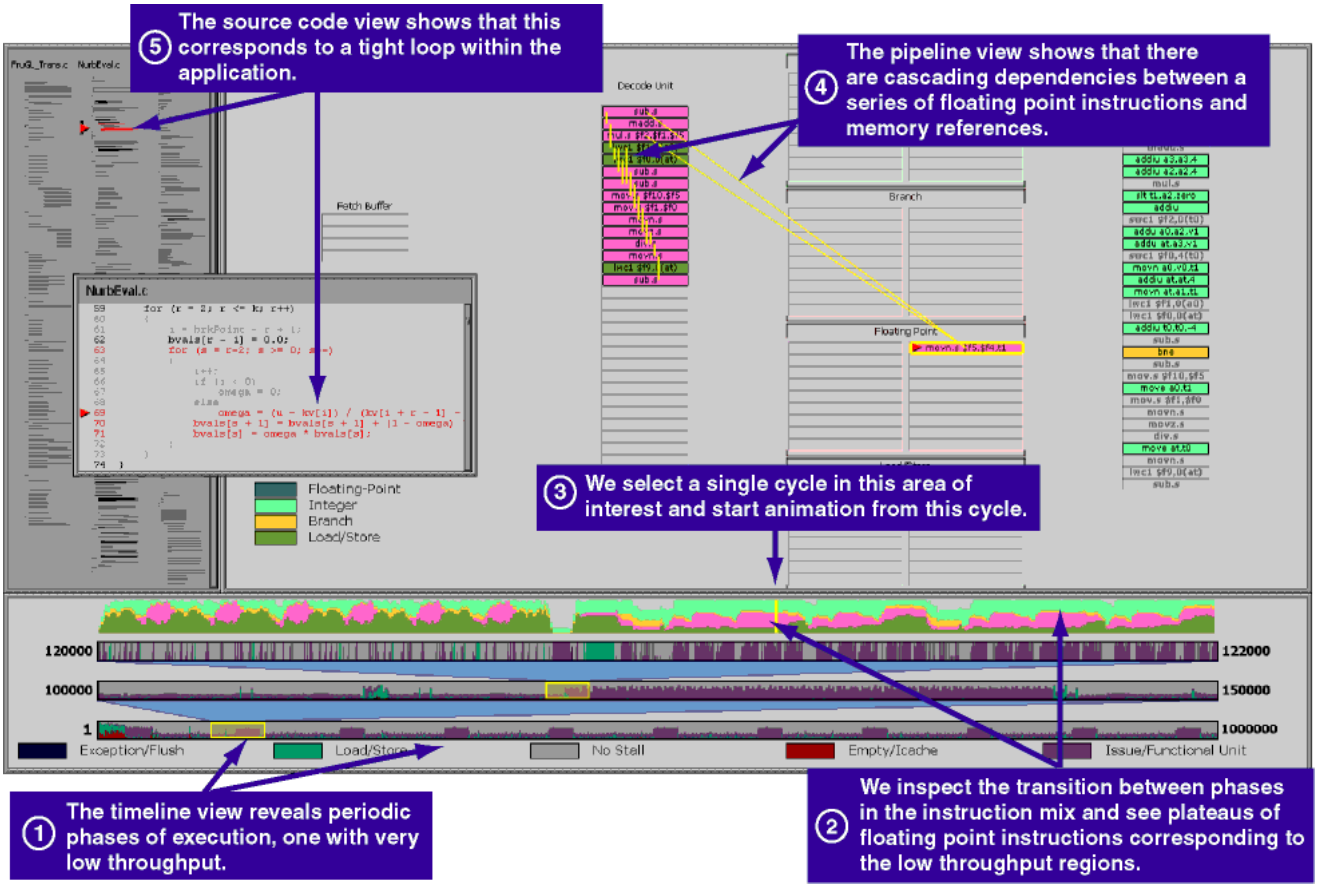

Figure 4: ***The complete processor pipeline visualization system displaying one million cycles of execution. The timeline view shows a periodic behavior, with alternating sections of high and low processor stall. The chart is zoomed in on the region of low utilization. The pipeline view shows that the instruction sequences in this window are highly dependent on one another, with very little instruction-level parallelism available to be exploited. The source code view shows the code segment corresponding to this phase of execution – a tight floating-point loop with dependencies both within the loop and across iterations.***

example shows how pipelining the divide functional unit can improve the performance of this application.

Simulator Development. Simulation is a powerful technique for the understanding of computer systems such as microprocessors. During the design of new processors, simulators are developed to explore the processor design space and validate architectural decisions. Simulators are also used for performance analysis of existing applications and processors. However, because of their complexity, the development of processor simulators is a challenging and error-prone task.

While developing the pipeline visualization system, we uncovered several errors in the MXS and MMIX processor models, many of them timing related. For example, in the original implementation of MXS, it was possible for instructions to advance through all the stages of the processor pipeline in a single cycle. Timing bugs such as these did not affect correctness – simulated programs would still execute correctly – but resulted in timing behavior that was not faithful to the processor model. While the aggregate pipeline statistics obscured these problems, which had existed for some time in the simulators, examination of the timeline view and observation of the pipeline animation made them prominent and enabled us to correct them.

7 CONCLUSION

We have presented a system for the visualization of a particularly complex system – superscalar processors. The main goal of our system is the analysis and optimization of application performance on this class of processors. By providing overview-plus-detail displays, the visualization system allows the user to see both high-level performance characteristics and the intricate details of out-of-order execution, speculation and pipelining. We have also described three other uses for our system: hardware design, compiler design and simulator development.

Our future work will build on this system in several ways. First, there are several other attributes of the processor state that we will incorporate into our visualizations, such as register files and write buffers, in order to provide a more complete picture of the pipeline's behavior.

Second, while superscalar architectures currently dominate the market, processors with alternate designs are also being developed in an attempt to maximize performance. For example, the Intel IA-64 architecture [7] uses a "very long instruction word" (VLIW) style of architecture. We intend to extend our visualization system to model these alternate architectures, enabling the exploration of trade-offs between different processor styles.

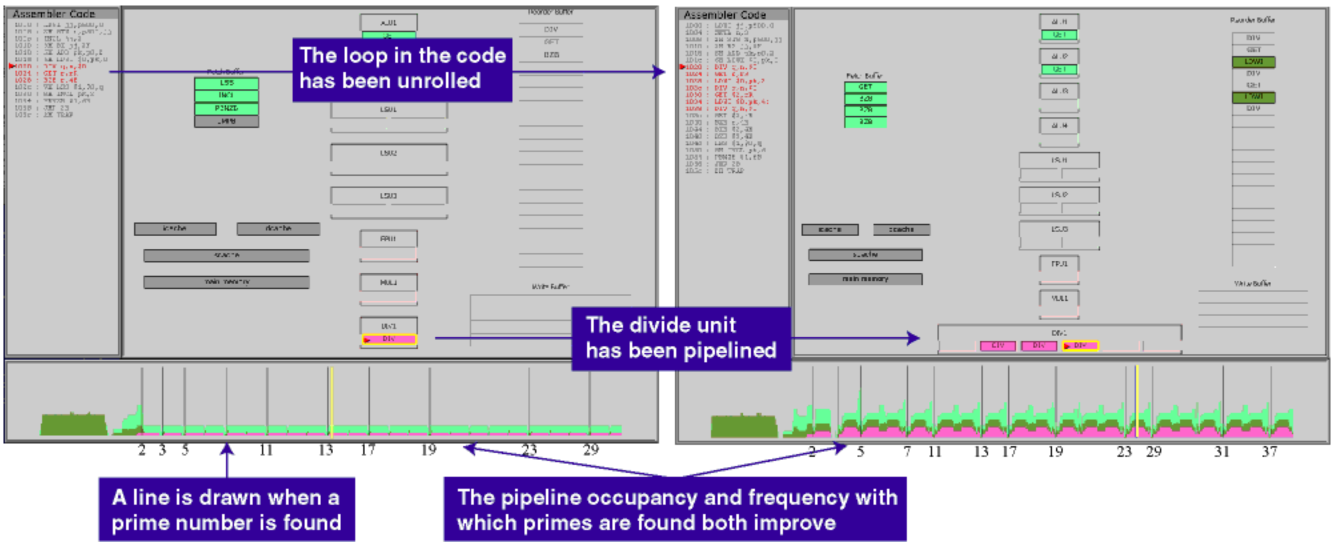

Figure 5: ***The pipeline visualization system can also be used to study the impact of changes to processor implementations. This figure compares the first 2000 cycles of execution of an MMIX program that calculates the first 500 prime numbers. On the left, an initial implementation of the program is executing on a pipeline with a simple 60-cycle functional unit for divide instructions. On the right, a modified version is executing on a configuration with the divide unit pipelined into six 10-cycle stages. In the modified version of the program, the main loop has been manually unrolled three times to better utilize the pipelined divide unit. To aid the comparison, the system draws a thin gray line in the instruction mix chart when the program finds a prime number. Examining the instruction mix chart for each processor, we can see that the second implementation consistently has more instructions in the pipeline and is progressing more rapidly. The increased amount of pink (floating-point instructions) in the instruction mix chart reflects the fact that the pipelined divide unit is enabling the processor to work on several divide instructions at once.***

Finally, although our pipeline visualization system has focused on the study of microprocessors, we would like to explore the application of this system to generalized pipeline systems such as assembly lines and organizational work flow. We expect the overall approach of overview-plus-detail will apply equally well in these areas, and our compositional architecture will enable us to adapt our visualizations to apply to these problem domains.

Acknowledgments

The authors thank John Gerth for his signficant contributions to the design and development of the Rivet system, and Donald Knuth for working with us to develop the visualization of the MMIX processor. We also thank Tamara Munzner, Diane Tang and David Ofelt for both their work reviewing this manuscript and for useful discussions.

References

[1] James Bennett and Mike Flynn. "Performance Factors for Superscalar Processors." Technical Report CSL-TR-95-661, Computer Systems Laboratory, Stanford University, February 1995.

[2] Trung A. Diep and John Paul Shen. "VMW: A Visualization-Based Microarchitecture Workbench." *IEEE Computer* 28(12), December 1995.

[3] *DLXView*. [online] Available: `<http://yara.ecn.purdue.edu/~teamaaa/dlxview/>`, cited March 1999.

[4] Stephen G. Eick, Joseph L. Steffen and Eric E. Sumner, Jr. "SeeSoft – A Tool for Visualizing Line-Oriented Software Statistics." *IEEE Transactions on Software Engineering*, 18(11):957-968, November 1992.

[5] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* San Francisco: Morgan Kaufmann Publishers, 1996.

[6] Stephen A. Herrod. "Using Complete Machine Simulation to Understand Computer System Behavior." Ph.D. Thesis, Stanford University, February 1998.

[7] Intel Corporation. *Merced Processor and IA-64 Architecture*. [online] Available: `<http://developer.intel.com/design/IA64/>`, cited July 1999.

[8] Donald Knuth. *MMIX 2009: A RISC Computer for the Third Millennium.* [online] Available: `<http://Sunburn.Stanford.EDU/~knuth/mmix.html>`, cited March 1999.

[9] Donald Knuth. *The Art of Computer Programming.* 3 vols. Reading, MA: Addison-Wesley, 1997-1998.

[10] Sougata Mukherjea and John T. Stasko. "Applying Algorithm Animation Techniques for Program Tracing, Debugging, and Understanding." *Proceedings of the 15th International Conference on Software Engineering*, pp. 456-465, May 1993.

[11] *Pentium® Pro Processor Microarchitecture Overview Tutorial*. [online] Available: `<http://developer.intel.com/vtune/cbts/pproarch/>`, cited March 1999.

[12] Ali Poursepanj. "The PowerPC Performance Modeling Methodology." *Communications of the ACM*, 37(6):47-55, June 1994.

[13] Ben Shneiderman. "The Eyes Have It: A Task by Data Type Taxonomy for Information Visualization." *Proceedings of IEEE Workshop on Visual Languages*, pp. 336-343, 1996.

[14] John T. Stasko. "Animating Algorithms with XTANGO." *SIGACT News*, 23(2):67-71, Spring 1992.

[15] Kenneth Yeager. "The MIPS R10000 Superscalar Microprocessor." *IEEE Micro*, 16(2):28-40, April 1996.

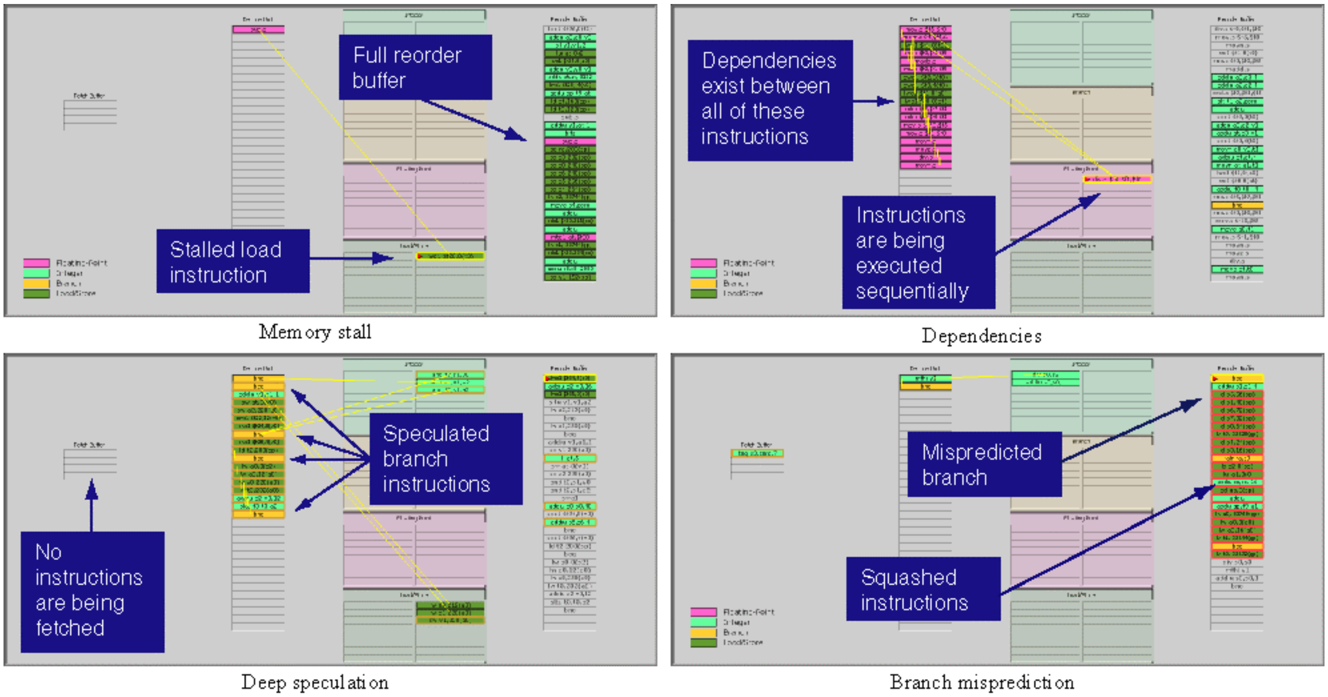

Figure C-1: *Snapshots of a program's execution in the pipeline view, demonstrating a variety of reasons for poor pipeline utilization. After using the visualization system for a short time, users can quickly identify these hazards as they occur in the animation.*

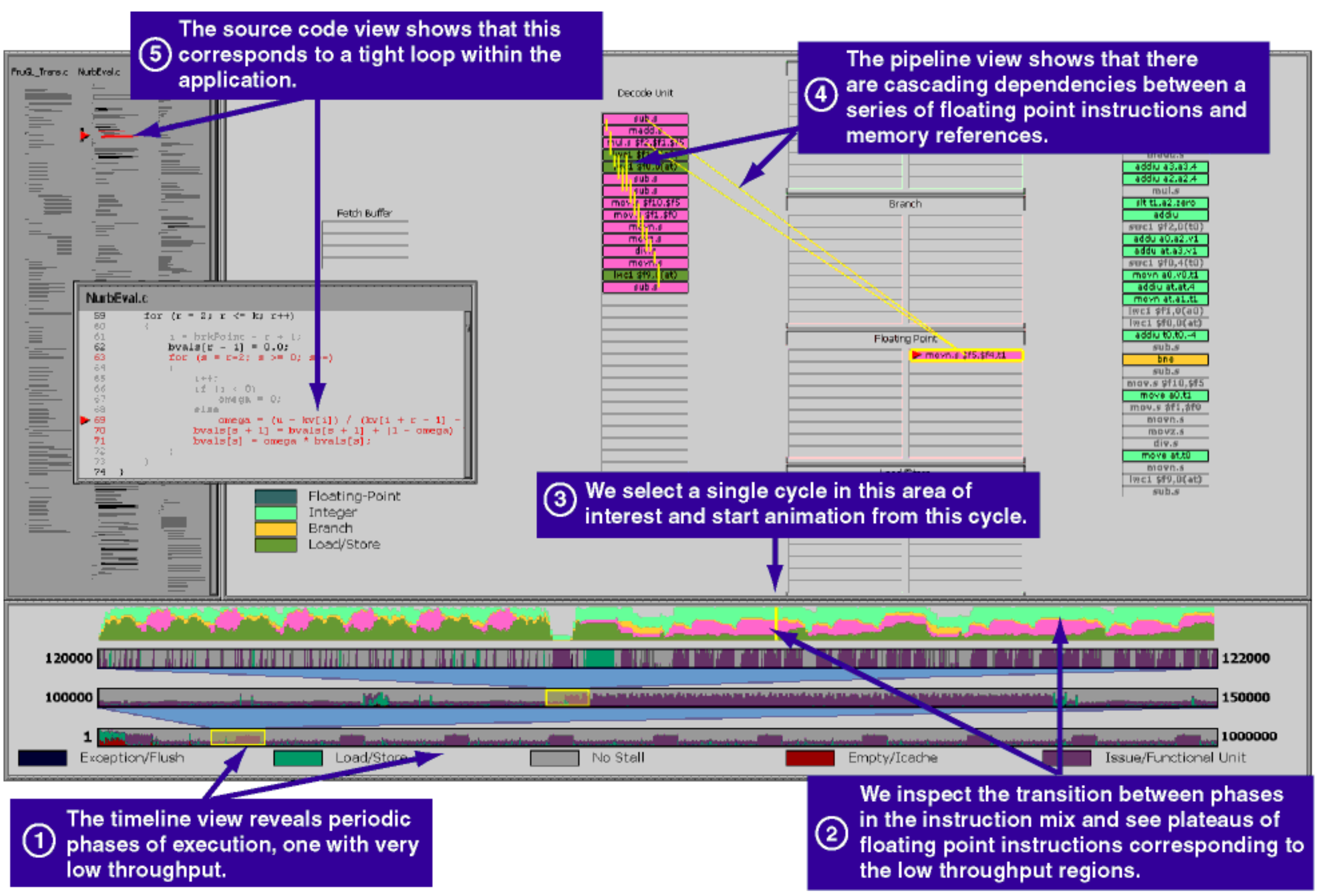

Figure C-2: *The complete processor pipeline visualization system displaying one million cycles of execution.*