THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

GRAPHICAL VISUALIZATION OF ARCHITECTURAL SIMULATORS

By

KELLEY C. JONES

A Thesis submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Master of Science

Degree Awarded:
Spring Semester, 2007

The members of the Committee approve the Thesis of Kelley C. Jones defended on April 4, 2007.

Gary Tyson
Professor Co-Directing Thesis

David Whalley
Professor Co-Directing Thesis

Xin Yuan
Committee Member

The Office of Graduate Studies has verified and approved the above named committee members.

To the three most important people in my life, Mom, Jared, and Lindsay...

ACKNOWLEDGEMENTS

I would like to thank my advisors, Dr. Gary Tyson and Dr. David Whalley, for their guidance and help throughout this process. I never anticipated accomplishing such a large undertaking, and I appreciate your encouragement and support.

I would also like to thank the other members of the Compilers and Architecture group for their helpful input when developing the visualizer. I am grateful for the constant help and patience from the senior members of the group in learning the research infrastructure. In addition, I would like to specifically thank Stephen Hines for graciously helping throughout the writing process and offering insightful advice.

I would like to thank my family and friends for their love and support. You believed in me, even when I did not, and I sincerely thank you for always being there for me.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

This thesis describes an architectural visualization tool developed to illustrate the instruction flow in a modern processor pipeline simulation. It was designed to aid in better understanding the complexities of a modern pipeline design. The visualizer allows the user to control the execution of the simulator by stepping ahead cycles in execution as well as setting breakpoints and watchpoints on microarchitected state data. The visualizer can also display the contents of microarchitected state at any time, enabling careful analysis of the state of the simulator. We believe that these features within the visualizer will allow a developer to more easily analyze the effects of a new compiler optimization or microarchitectural addition in more depth than typical simulator statistical data provides. Furthermore, we feel the visualizer is a valuable teaching aid in computer architecture classes, as it allows students to interactively visualize the control and data flow of a program through a particular pipeline design. Lastly, the visualizer was created in such a way to provide easier interfacing to other simulators.

CHAPTER 1

INTRODUCTION

As architectures advance and microarchitectural components become increasingly complex, the ability to understand the behavior of a program becomes harder. Not only is the program flow through the machine more complicated, but also cache access patterns, resource stalls, conflicts and other pipeline bottlenecks become harder to understand. Advanced microarchictectures also lead to complex simulator implementations. This, in turn, causes simulator validation and error correction to become much harder to handle and rather time consuming. Furthermore, understanding why a new compiler optimization or microarchitectural component does not affect performance as expected is harder to do with only statistical data that is generally gathered from a simulation.

This thesis presents a visualization tool that will display a graphical representation of a simulator. The goal of creating such a tool is to aid in better understanding the complexities of a modern pipeline design. More specifically, this tool hopes to aid in simulator validation and error correction. Complex simulator implementations require extensive testing and validation before they can be safely used in gathering performance data. We hope the use of the visualization tool will facilitate a quicker and more efficient testing and validation phase of simulator implementation.

In addition, the visualization tool may be used to accelerate analyzing why a particular compiler optimization or microarchitectural addition does not produce the benefits expected. In many cases, solely relying on performance statistics from simulation does not clearly explain why such additions do not perform as expected. Whereas, with the aid of a visualization tool, the developer may carefully step through simulation to determine what is happening within the pipeline.

Some of the key features of the visualizer are controlling the execution of the simulator,

displaying the contents of microarchitectural state, data highlighting, and a state registration process. The user is able to control the execution of the simulator within the visualizer via two features. The first feature allows the user to skip ahead through particular units of execution. For example, the user may skip through one cycle of execution, or 10 instructions. Following the specified time frame, the visualizer will then display the current state of the simulator. Another option available is the use of breakpoints and watchpoints, allowing even more powerful control of the simulator. By setting a watchpoint or breakpoint on a piece of data, the user may go directly to a particular event of interest, skipping past less meaningful portions of execution.

When visualizing a complex simulator, there is a large amount of state data being conveyed which can be difficult for the user to digest. To assist in spotlighting important information, the visualizer provides highlighting capabilities. Whenever any state within the visualizer has changed, the value is highlighted to bring the update to the attention of the user. The highlighting feature is also used to alert the user of any errors. This feature can facilitate the debugging of a simulator implementation. Once an error is found, execution is stopped, and the architected state that is incorrect is highlighted within the visualizer. From here, the programmer can pinpoint what caused the incorrect value, and can determine the error in the simulator implementation.

One of the main goals when creating the visualizer was to ease the burden of displaying new simulator additions within the visualizer. In order to provide extensibility, a simplified process for enabling the display of new microarchitectural features was developed. To achieve this, the visualizer needs to generalize state so that any new state can be handled. This capability is accomplished by utilizing a state registration process. When a programmer adds a new piece of state to a simulator, the state must also be registered with the visualizer. In doing so, not only does the programmer categorize the state within some predefined types, but call back functions are also implemented so the visualizer knows how to deal with the state. By encompassing almost all of the information needed to handle a new state within one process, it allows for easier state integration into the visualizer.

Figure 1.1 illustrates the high-level overview of the flow of information when visualizing a simulator. The user makes selections within the visualizer to request state information or control simulator execution. The visualizer then constructs the corresponding requests, and sends the commands to the simulator. The simulator will respond with the requested data,

which the visualizer, in turn, displays to the user. From the figure, we see there are two main parts of the visualization framework. There is the user interface, on the client side, and a simulator on the server side. Modifications to the simulator are made to make calls to server-side visualizer code which then extracts the necessary state information.

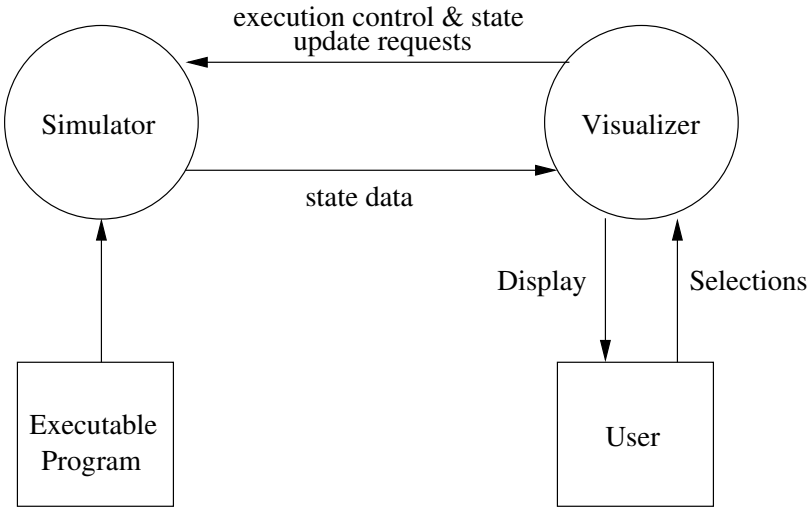


Figure 1.1: Overview of the Visualizer framework.

The remainder of this thesis is broken down as follows. Chapter 2 discusses the functionality of the visualization tool from the user's point of view. This chapter discusses how to use each feature within the visualizer, in detail. Chapter 3 discusses the implementation details of the visualization features including the design choices made when developing the tool. Chapter 4 explains the process of interfacing the visualizer with a simulator, as well as discussing the process and challenges involved when interfacing the visualizer with the SimpleScalar tool set. Related work is reviewed in Chapter 5. Chapter 6 then highlights some of the future enhancements planned for the visualizer. Finally, Chapter 7 contains the conclusion of the thesis.

CHAPTER 2

FUNCTIONAL DESCRIPTION

In this chapter the functionality of the visualizer is discussed from the user's point of view. The functionality discussed here includes how to instantiate the visualizer, viewing the state of the processor at different intervals, stepping through execution, creating and using breakpoints and watchpoints to skip to certain points of execution, and viewing statistics.

2.1 Overview of the GUI

The main window of the visualizer GUI can be seen in Figure 2.1. The bottom of the window contains the execution control for stepping forward. Below that is a message area which informs the user of the current status of the visualizer. The messages here can also prompt the user on what input is required. States that are always displayed can be found on the left side of the main window. The states found here are typically the program counter, cycle count, and general purpose registers. In the center of the main window lies an area where the user can view states that are not designated as always displayed. The user may select a state to view in this portion of the window. On the right side of the main window is a list of the most recent and upcoming instructions, disassembled for easier comprehension. The menu bar located at the top of the window gives the user options to view even more states. The menu bar also offers advanced options like setting and deleting watchpoints and breakpoints.

2.1.1 General Sequence of Events

The visualizer is not meant as a stand alone tool. Instead, it must be interfaced with an architecture simulator and run concurrently with the simulator. Currently, the visualizer has been interfaced to work with the SimpleScalar tool suite. In this chapter we only consider the

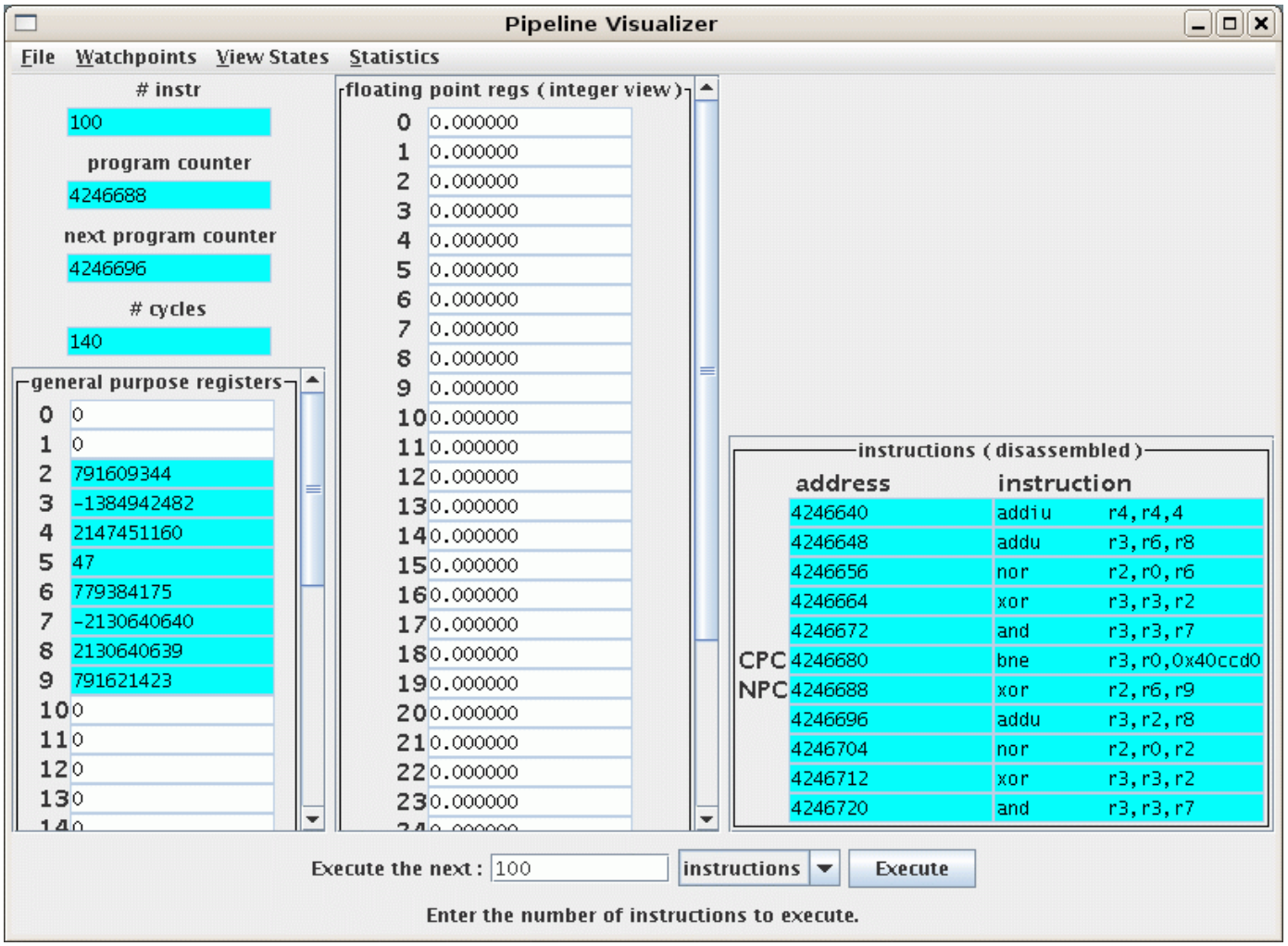


Figure 2.1: The main window of the visualizer.

functionality of the visualizer GUI and save the interfacing process as well as the server-side, simulation behavior for discussion in later chapters.

Before invoking the visualizer, the simulator must be invoked with the program to run, as well as the proper flags to communicate with the visualizer. For example, to invoke the in-order simulator *sim-inorder* in SimpleScalar one would need to type a command similar to the following:

```
sim-inorder -redir:sim ss_results.txt -port 4021 ./a.out.ssexe
```

Here, `./a.out.ssexe` is the program to simulate, and `-port 4021` specifies the port number for communicating with the visualizer. Any other simulator arguments can also be used when

invoking the simulation, such as `-redir:sim ss_results.txt`, which redirects all simulator output.

Once the simulator is invoked with the proper visualization options, the visualizer can then be invoked with the command:

```
java ArchVis <portno> <machine>
```

Here, <*portno*> is the port number that the visualizer will use to connect to the simulator. <*machine*> is the name of the machine on which the simulator is running. Once both the simulator and visualizer are running, an initial transfer of information is necessary to initialize the visualizer. This process is discussed in more detail in Section 2.1.2. To begin this initial transfer of information, the user must click the *begin* button at the bottom of the main window of the visualizer. After the initial information is transmitted, the visualizer awaits direction by the user before simulation begins. The user has two options of how to control execution. The user may step forward a certain amount of units (typically number of cycles or instructions), or set a watchpoint or breakpoint. Both of these options are discussed in greater detail in Section 2.3. Once the user has chosen a method to control simulator execution, the user may click the *execute* button. At this point, the visualizer informs the simulator to run the program until the control requirement is met. The visualizer then receives and displays all of the state changes. Now, the user may again select another control option, and the cycle continues.

After the visualizer has displayed new updates, and before the user selects *execute* again, there are other features that may be utilized. The user may select to view any statistics, parts of memory, or add new watchpoints and breakpoints. Each of these features, as well as other aspects of the GUI are described in more detail in subsequent sections.

2.1.2 Initial State of GUI

When the GUI is first invoked, there is no state displayed. The user must select *begin* so that the visualizer can begin communication with the simulator. First the visualizer receives the state definitions, as the states used in the visualizer can vary from simulator to simulator. Next, the visualizer receives the initial state contents. Once the visualizer has both the definitions and initial state, it can then display the state contents to the user.

2.2 Viewing State

One of the most important features of the visualizer is the ability to view the data within the different states at any time. There are three main types of states: scalar states (counters), one-dimensional states (register file), and two-dimensional states (disassembled instructions). However, within these three types, states can still have different characteristics. Depending on their characteristics, the visualizer treats displaying of the state differently. Each state is identified with a predefined type, or category, during the registration process mentioned in Section 2.1.2. In this section, we discuss the different state categories, how they are viewed, and the characteristics of each type of state.

2.2.1 Internal States

Internal states are those that can be viewed within the main GUI window. To select an internal state for viewing, the user can select a state from the internal state sub-menu in the View menu, as shown in Figure 2.2. Once selected, the internal state will be displayed in the center of the main window. At any given time, only one internal state can be displayed in this location. If the user does not want any internal states to be displayed in the center region of the visualizer window, there is a *none* option in the internal state sub-menu that will clear the center region.

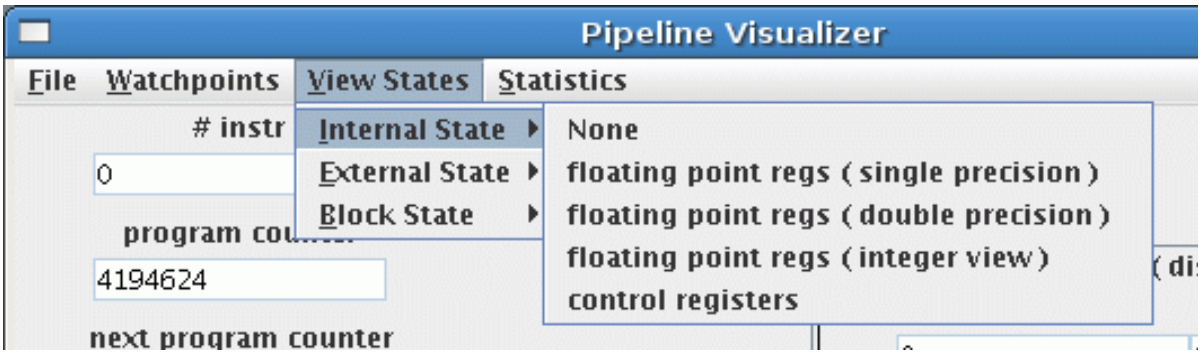


Figure 2.2: The view states menu in the visualizer.

Internal states have two main traits. The first trait is that internal states should be relatively common states; those which the user would be interested in viewing on a regular basis. This means states that are commonly used within instructions are good candidates for

internal states. The other important trait of internal states is that they not be overly large. For example, within a pipelined architecture, the pipeline registers are extremely important to examine when trying to understand the data and program flow. However, having the pipeline registers be internal state would clutter the main window. The pipeline registers contain a fair amount of data, and by placing them in the main window, the user may not be able to view all of the pipeline register data at once. In general, most architected states are categorized as internal states. Architected states, such as the register file, are referenced explicitly in machine code and are therefore accessed in almost every cycle. Not all architected state falls into this category, however, due to the requirement that an internal state be of a relatively small size.

Always Displayed Internal States

Always displayed states are a special case of internal states, located on the left side of the visualizer window. Even within internal states, there are some states that are accessed more often than others. For instance, the general purpose registers are utilized within instructions much more often than floating point registers. Therefore, one would probably want to always be able to look at the contents of the general purpose registers while stepping through an execution. Whereas, having the contents of the floating point registers viewable will not be necessary much of the time. With this in mind, we felt it would be beneficial to allow some state to always be viewable (always displayed state), and allow the other states to be viewable only if selected. This also allows the user to look at the general purpose registers or counters on the left panel while also inspecting other internal state within the center panel.

2.2.2 External States

External states are viewed differently than internal states. As the name suggests, external states are displayed in a window separate from the main visualizer window. An external state can be viewed by selecting a state in the external state sub-menu in the View menu. Once an external state is selected, a new window will pop up containing the specified state. The user is able to resize and move the window to their liking. While only one window of a single particular external state can be created, multiple different external states can be viewed at once. At any time, the user may choose to close the window displaying the external state. At any point, the user may reopen the window as well. Closing the window

does not lose any data, so reopening the window immediately will display the same state contents. Figure 2.3 shows an example of an external state being displayed in a separate window of the main visualization window.

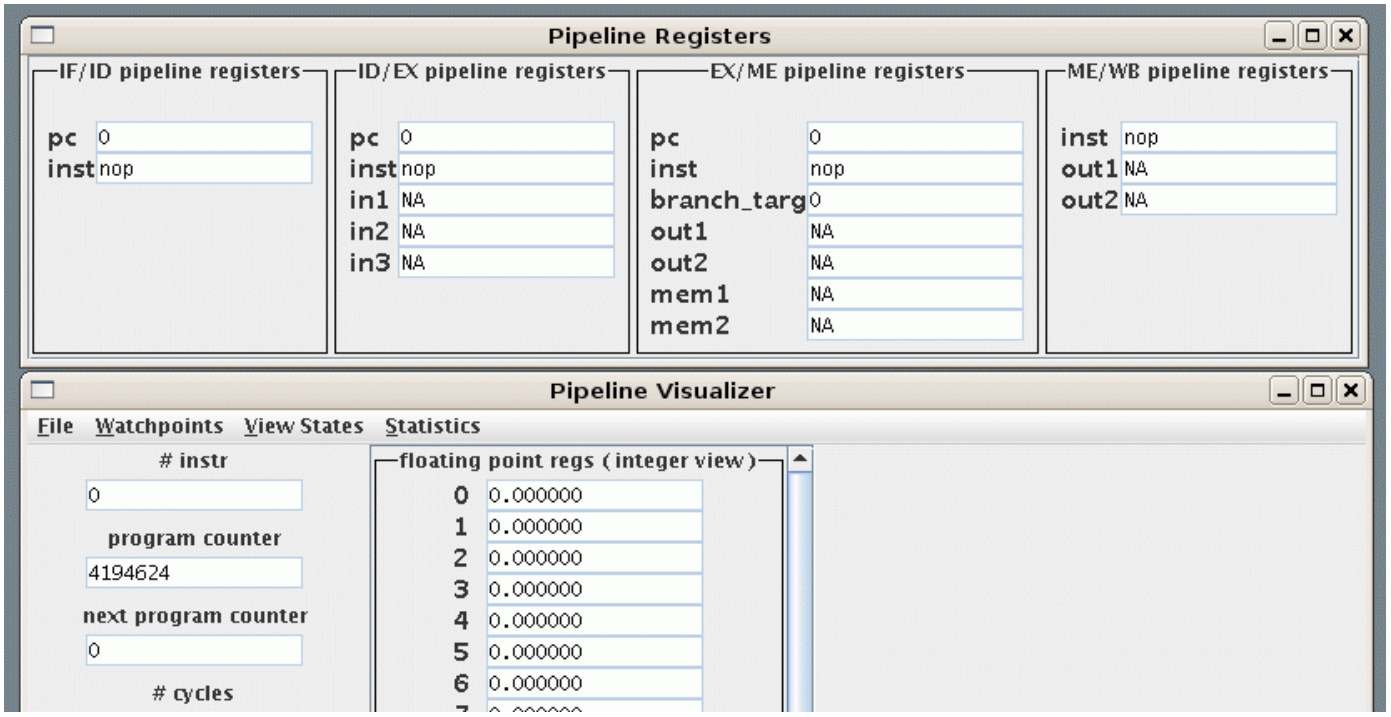


Figure 2.3: An example of an external state window.

Internal states were described as being states that are commonly accessed. External states can sometimes be extremely common, as well, but do not fit well as an internal state. For example, in a pipelined architecture, the pipeline registers are modified each cycle. These registers are imperative when trying to understand the data and control flow of an executing program. Without them, the user would be lost as to what is actually happening within the architecture during execution. However, putting the pipeline registers within the main window would limit the user to what could be viewed. First, since only one internal state can be viewed at a time, the user would have to constantly switch between the pipeline registers and whatever other state he/she is interested in. Furthermore, depending on the length of the pipeline, the pipeline registers could contain large amounts of data, so much that the user could not view it all at once. By making the pipeline registers external state, the user has a little more room to expand the new window created, so they can view the

optimal amount of data in the pipeline registers.

External states give more flexibility as to how much the user wants to view. If the user is running a simple simulator, such as a functional simulator, the user may not want to view much, if any, of the external states. However, if the user is running a more advanced simulator, such as a five-stage pipelined simulator, the user may want to view the pipeline registers. In an even more advanced simulator, like an out-of-order simulator, the user may want to view the rename table, the reorder buffer, and other states important to an out-of-order architecture.

In general, we see that external states are mostly composed of non-architected states, such as pipeline registers. The amount of non-architected state can vary greatly between different architecture designs, whereas the architected states are typically similar between designs. Depending on the complexity of a particular pipeline design, more non-architected state may exist. Furthermore, in the course of exploring new research ideas, non-architected states have a higher likelihood of being modified than architected state, as they only require changes to the simulator implementation, whereas modified architected state also require changes to the instruction set. Clearly, the size and overall importance of a piece of non-architected can vary greatly. For this reason, it is more logical to categorize most non-architected state as external states, or even block states, providing the user the flexibility to view the states only when necessary.

2.2.3 Block States

One final state category that needs distinction from internal and external states are block states. Block states are similar in type to external states in that they are too large to display in the main window. In fact, block states are states which cannot be displayed in full due to their immense size. Memory is a prime example of a block state. Displaying all of the contents of memory at once would require entirely too much space on the computer screen as well as too much processor memory to actually accomplish. In addition, the shear amount of data displayed would only overwhelm and confuse the user. In most cases, the user is only interested in a small subset of the memory, or a block of memory. We provide the capabilities for a user to explicitly specify which part of the data they want to be displayed.

A block state can be viewed by selecting a state from the block state sub-menu in the View menu bar. Once a state is selected, a window will appear prompting the user for some

information, as can be seen in Figure 2.4. The user must specify the beginning index of the block state to view. The user must also indicate how many entries of the block state to display. Once the user enters this information, the validity of the block state request is verified. It must be verified that the values entered by the user not only make sense to the specific block state, but are also valid. If the user enters any invalid values, the visualizer will prompt the user for new values. Otherwise, the visualizer creates a new window that displays the requested block of state, as in Figure 2.5. Upon subsequent updates, the state values will be updated, as long as the block state is still displayed.

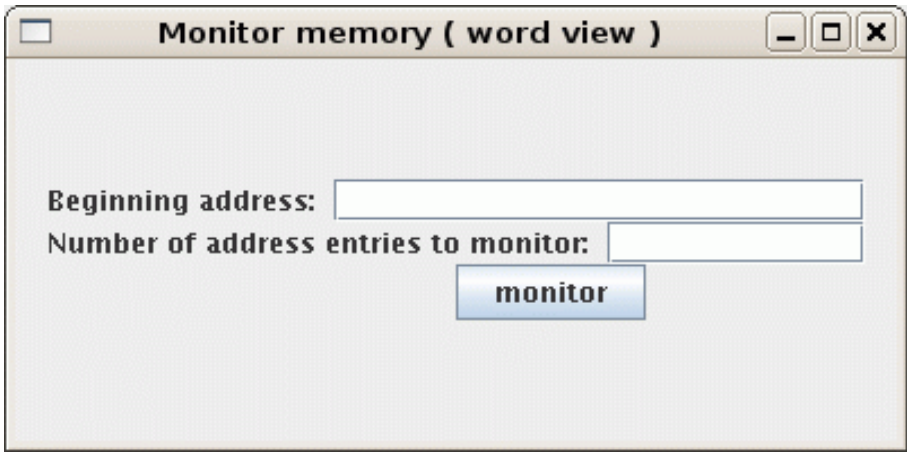


Figure 2.4: A block state prompt window.

A user is able to display multiple different blocks of the same piece of block state. These blocks can overlap or be exact duplicates. At any point, the user may close the block state window, losing all information about that current block of state. If the user wants to view the same block of state, a new window must be created with a new request. The visualizer will then display the current state of the block, again.

2.3 Controlling Simulation

One of the most important features of the visualizer is controlling simulation. It is imperative to the usefulness of the visualizer to be able to control when the user wants to see state contents, and when the user does not care to do so. There are two different ways to control the simulation of the program. The first is execution control. The second is utilizing breakpoints

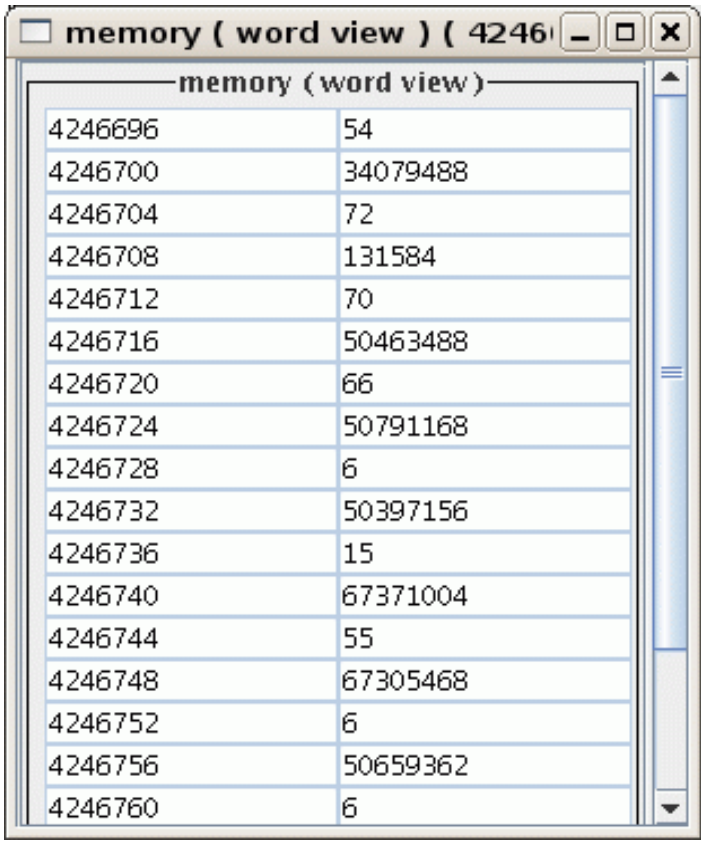


Figure 2.5: A windowing displaying a block of state.

and watchpoints. In this section, both are discussed in detail.

2.3.1 Execution Control Options

When describing the execution of a program within a processor, units of time are typically broken up into cycles, or in simple non-pipelined cases, into the number of instructions executed. In a simple, non-pipelined simulator, the number of instructions and the number of cycles would be exactly the same, since an instruction is fully executed during each cycle. However, in a pipelined architecture, the number of instructions executed during a certain number of cycles is usually not equal to the number of cycles elapsed. Therefore, having both of these variables used as execution control variables provides for different functionality. By stepping ahead n cycles, the visualizer will display the state updates at the end of the nth cycle. By stepping ahead n instructions, the visualizer will display the state updates after

n instructions have fully traversed the pipeline, and have been committed. Both notations can be helpful when studying the behavior of a program in a simulator.

To use an execution control variable to step ahead through execution, the user selects one of the units in the drop down box at the bottom of the main window, as illustrated in Figure 2.6. The user then enters a value and must click the *execute* button. If the value entered by the user is invalid for that control variable, the visualizer will prompt the user for valid input. Otherwise, the visualizer will inform the simulator for the specified time frame. The visualizer then updates the state data for all modified states.

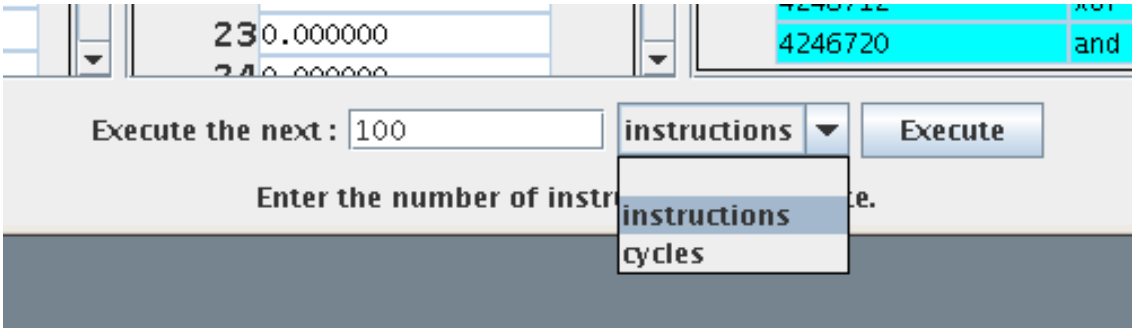


Figure 2.6: Selecting an execution control variable in the main window of the visualizer.

These execution control variables provide the user with the capabilities to skip past uninteresting parts of execution. This is an important feature, as during long-running programs, there is a warm-up period for many micro-architectural states, like branch prediction tables, caches, etc, to reach a steady state. By skipping forward x amount of instructions or cycles, the user can then begin to step through execution with these states in a steady state, and can inspect their behavior.

2.3.2 Breakpoints and Watchpoints

While the execution control options discussed in Section 2.3.1 can be very powerful, there are times when a user could benefit from more extensive and flexible execution control. Any programmer who has ever had to extensively debug a program can attest to how invaluable breakpoints and watchpoints can be. They allow the user to fast-forward a program to a particular action where they can then carefully examine behavior while stepping through the program. By adding the functionality that allows the user to set breakpoints and watchpoints

in the visualizer, the user is given greater power over debugging. In addition, stepping ahead to a particular event allows the user to carefully examine this event, without having the pain of slowly going through execution to find it, one step at a time.

There are two types of watchpoints a user may set: an unconditional watchpoint and a conditional watchpoint. When an unconditional watchpoint is set on a piece of data, the watchpoint will be triggered whenever that data value changes from the current value. With a conditional watchpoint, the user specifies a value for the state. The watchpoint will be triggered when the state is equal to this value.

There are more advanced options for watchpoints available. While it was mentioned that with a conditional watchpoint the user specifies a value at which to stop, it is also possible to select a different comparison. For instance, the user could set a conditional watchpoint on data x stopping only when the value of x is greater than some value y. The different comparisons available are $>$, $<$, $=$, $\geq$, $\leq$, and $\neq$. This gives the user more power in describing what event they are looking for. For example, these comparison capabilities would allow a user to set a watchpoint on general purpose register 2 ($r[2]$), specifying that the watchpoint should be triggered when $r[2] > 4$, or when $r[2]! = 0$.

Another advanced option is an ignore count for the watchpoint. With this option, the simulator only stops when the watchpoint has been reached a certain number of times. Each watchpoint has a default ignore count of zero, meaning it does not ignore any watchpoints, and stops on the first occurrence of the watchpoint. However, the user can set the ignore count to any positive integer. When an ignore count greater than zero is reached, the count is decremented and execution continues. Once the count gets down to zero, execution will then stop. In other words, by setting an ignore count to x, the watchpoint will not stop the next x times the program reaches the watchpoint.

The user may utilize both advanced features to create some powerful watchpoints. For instance, the user may want to only stop execution after a particular PC has been reached five times, indicating a possible loop. To do this, the user just sets a watchpoint on the program counter, and sets the value to be the PC in question, then sets the ignore count to 4, causing the watchpoint to stop on the fifth occurrence of the PC.

The user may set breakpoints and watchpoints at any time when the simulator has paused execution and the visualizer has control. There is a watchpoint menu in the menu bar, which can be seen in Figure 2.7. To add a watchpoint, the user may select the *Add Watchpoint*

option. A prompt will be displayed with a drop down list of states that allow watchpoints. The user selects a particular state. From there, the user may have to enter one or more indices for a certain portion of state. For scalar state, the index fields are not required. For one-dimensional states, the first index must be filled in. For two-dimensional states, both index fields must be filled in. The user can select the watchpoint to be conditional or unconditional. If the watchpoint is unconditional, then there is no need for the user to enter the value or select a comparison. The user may select an ignore count for unconditional watchpoints, however. This feature allows the user to create a watchpoint after a piece of state has been modified n times. If the watchpoint is conditional, the user must enter a value for the piece of data. The default comparison is $=$, meaning the watchpoint is triggered when the piece of data equals the value entered. However, the user may select one of the alternate comparisons. The user may also select an ignore count. The default is zero, meaning the watchpoint is never ignored. An example can be seen in Figure 2.8, which will request a watchpoint to be put on general purpose register 3, with the condition that the register value be ≥ 10000. The watchpoint in question will only pause execution after the fifth time this condition is met, since there is an ignore count of 4. Once the user clicks the *OK* button in the prompt, the validity of the user input is verified. If the input is valid, the watchpoint is created. Once the user selects the *execute* button in the main window of the visualizer, execution will continue. If the watchpoint criteria is met, execution will pause.

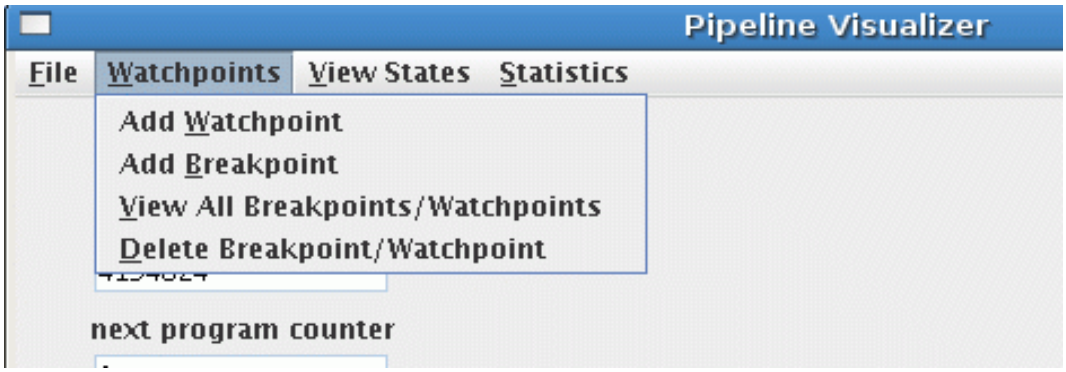


Figure 2.7: The Watchpoints menu in the main window of the visualizer.

Setting breakpoints are similar to setting watchpoints. Under the Watchpoints menu, there is an *Add Breakpoint* option. The user enters a PC value at which they wish to break.

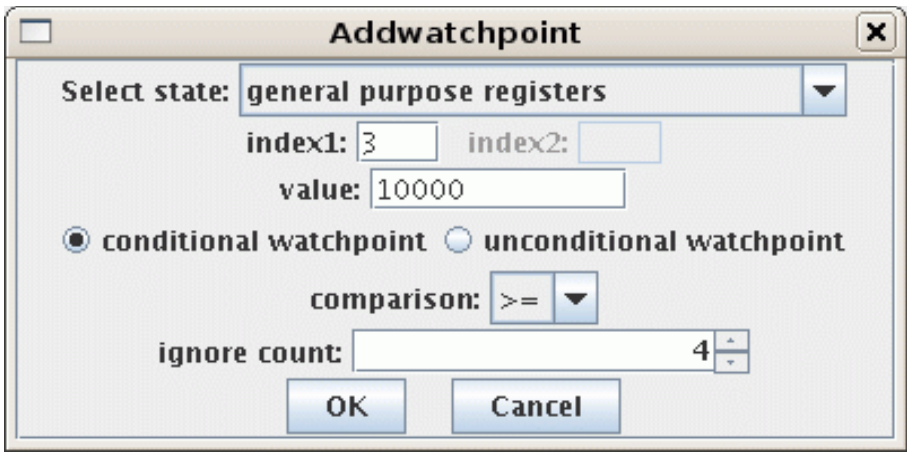


Figure 2.8: A watchpoint prompt.

The user may be prompted to enter a new value, if it is invalid. Once the user has entered a valid breakpoint value and clicks the *execute* button, execution continues until the breakpoint value is reached. A breakpoint is just a special case of a watchpoint, however we include the ability to explicitly add a breakpoint for ease.

The user can view the current watchpoints and breakpoints set by going to the Watchpoints menu in the menu bar, and selecting *view*. A new window will appear with the current watchpoints and breakpoints set, as can be seen in Figure 2.9. The user can delete any of the entries at any time. However, if the simulator is currently executing, the deletion of the watchpoint will not take effect until after execution has paused and given control back to the visualizer.

Not all states are listed as allowing watchpoints because some do not make sense to have a watchpoint. However, if the user feels the need to add watchpoint capability to any state, the server-side code can be modified to add the necessary function pointers for the state. This is discussed further in the section on the implementation of watchpoints.

2.3.3 Execution Control and Watchpoints Together

One important point to note is that execution control variables and watchpoints run at the same time. In other words, if a watchpoint is put on the PC for a certain value, and the execution control is set to step forward 100 cycles, the simulator will stop at the event that occurs first. If 100 cycles have elapsed and the watchpoint has not been met, the watchpoint

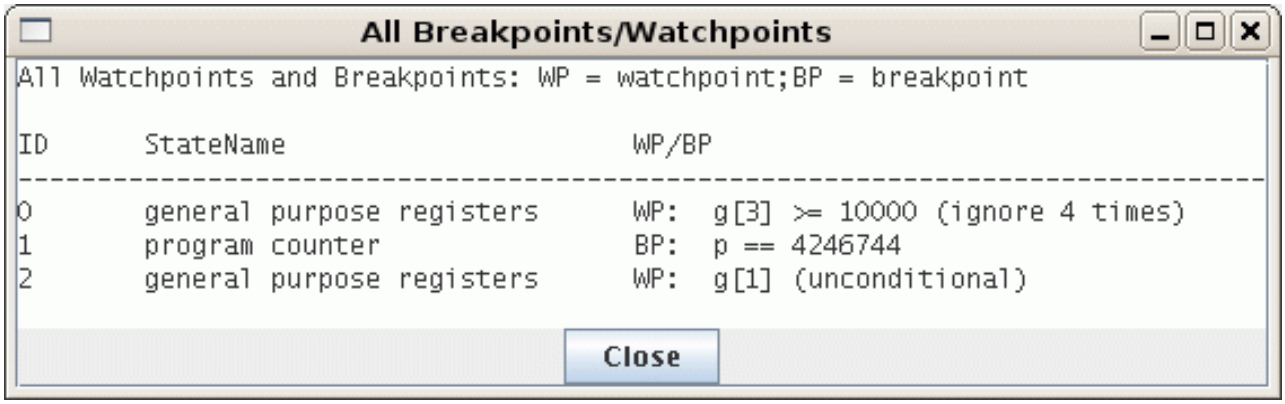


Figure 2.9: A list of all currently active watchpoints.

is still checked upon subsequent step forward commands. However, if the watchpoint is met first, the current execution control command is disregarded. For example, if the watchpoint of the PC is met after only 50 cycles, when the *execute* button is again clicked, the simulator does not remember that it had 50 cycles left to execute. Instead the visualizer sends a fresh execution control command, and the simulator bases its execution off of that. So, if the user had not changed any of the setting of the execution control, the visualizer would again send a step forward command of 100 cycles.

In the future, the option of turning off execution control may be implemented. If so, a check box would be created next to the *execute* button, given the user the option of disregarding the step execution. This is discussed in more detail in Chapter 6.

2.4 Data Highlighting

An advanced simulator can be very intricate and involved. Even if a user is only stepping through execution one cycle at a time, there are still many events that happen within the course of a cycle that a user may not be able to easily pick up on. This could be due to a large amount of states being displayed at once, or even just the complexity of the particular simulator being run. To assist the user in recognizing important events, the visualizer employs the use of data highlighting. There are three types of data highlighting used. The first is update highlighting. Another type of highlighting is for data forwarding. The last type of highlighting is for detected errors. All three types use a different color, to

help distinguish the difference in events. This section looks at the three types of highlighting in detail.

2.4.1 Highlighting Updates

On each update from the simulator, a fair amount of data could change. If the user is currently displaying many different states within the visualizer, it may be hard to figure out what has changed. In order to emphasize the changes since the last update, the visualizer will highlight any data changes. The highlighting can be very exact. For instance, if one of the general purpose register states have changed on the new update, instead of highlighting the whole register file, only the registers that have changed are highlighted. Another example is that if an address in the disassembled instruction entries changes, but the instruction happens to be the same as the last one, only the address data is highlighted. By being as detailed as possible with the highlighting, the user can see exactly what has changed, and exactly what has not. The user can gleam the overall effect of the update from the highlighting without even having to carefully study the actual contents of state. An example of this would be if the later pipeline registers are highlighted as being updated, but the earlier registers are not. The user can easily determine this was due to a stall. It is evident that by providing this update highlighting, the visualizer is making the study of data and control flow easier for the user to digest.

2.4.2 Highlighting Forwarded Data Values

The next type of data highlighting is more specific to pipelined architectures. But, considering the popularity of pipelined architectures, it is a good highlighting feature to include. This form of highlighting is used when a data value will be forwarded from another source to the execute stage, instead of using the contents of one of the registers. The particular register whose value will not be used is highlighted to alert the user that the value will be forwarded. In this way, when the user is viewing the pipeline registers and stepping through cycles, the user will be aware that the value in the register will not be used. This feature also better illustrates the dependencies involved in a pipelined architecture that may not be clearly visible to the user when looking at the pipeline registers. Figure 2.10 illustrates an example of forwarded data highlighting. In the ID/EX pipeline register, the value of in1 is marked as being forwarded. Upon closer inspection, we see the instruction

in that register, "`and r3,r3,r7`", has operands `r3` and `r7`. The instruction in the EX/ME register, "`xor r3,r3,r2`", writes to `r3`. Therefore, the value of `r3` that was read during the decode stage is invalid, and instead the value will be forwarded from the result of the execute stage, `out1`. The highlighting makes the data forwarding more obvious, whereas without the highlighting, the forwarding would probably only be noticed when closely inspecting the instructions in the the pipeline.

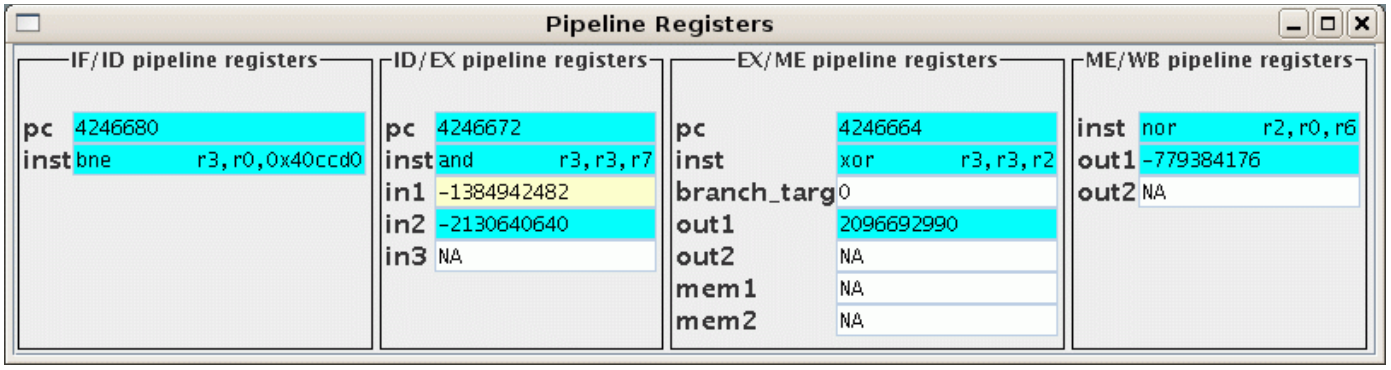


Figure 2.10: A state which has both update highlighting (in blue) and forwarded data highlighting (in yellow).

2.4.3 Highlighting Errors

The last type of highlighting is for spotlighting errors. Within the five-stage, in-order pipelined simulator, there is checking built into the simulator to ensure that an instruction was executed correctly in the pipeline. Once an instruction is retired, if any of the resulting data is incorrect, the incorrect values are highlighted within the visualizer. As soon as an error is detected, execution immediately pauses, even if a watchpoint or execution control value is not reached. This is done so the user is able to see in exactly which cycle the error was detected.

Providing this functionality is beneficial to the user. First, the simulator could be producing incorrect results, and the user may be trying to find out why, by running the visualizer. Stepping through execution one cycle at a time can quickly become tedious. Instead, the user may just run the visualizer until any errors are detected, effectively fast-forwarding to the problem cycle. The user now knows in which cycle the error occurs, and

can carefully step through execution near the cycle in question to discern what could be causing the problem. Alternatively, the user may want to continue execution after an error is found, to possibly discover more errors. Because the incorrect states are highlighted, the user may find that only a certain state is always the cause of the error. This allows the user to pinpoint what portion of the simulator implementation could be incorrect.

2.5 Other Features

While the main features of the visualizer are viewing state, controlling execution, and highlighting important events, there are other features which can provide more assistance to the user. We look at some of the other features available in this section. While minor, these features can prove to be rather powerful, and can convey important information to the user.

2.5.1 Viewing Statistics

The user is able to view all simulator statistics via the Statistics menu in the menu bar. The menu lists all statistics, as illustrated in Figure 2.11. Once a statistic is selected, a new window will appear with the current value of the statistic. The user may select a statistic to view from the menu at any time. However, if the simulator is currently executing, the value of the statistic will not be displayed until the visualizer receives updates. On each subsequent update from the simulator, the statistic value will be updated if it has changed. At any time, the user may close out the statistic window. For convenience, there is also an option to close all statistic windows at once, if the user has multiple statistic windows currently open.

While there is no current support for putting watchpoints on statistics, it is a feature that is being considered for future inclusion. This capability would provide the user with even more power when trying to navigate through a program. This proposed feature is discussed in more detail in Chapter 6.

2.5.2 Disassembled Instructions

In Figure 2.1, a panel containing a number of address/instruction pairs can be seen on the right side of the main visualizer window. This panel contains disassembled instructions. Each entry in the disassembled instructions panel has an address and the string version of

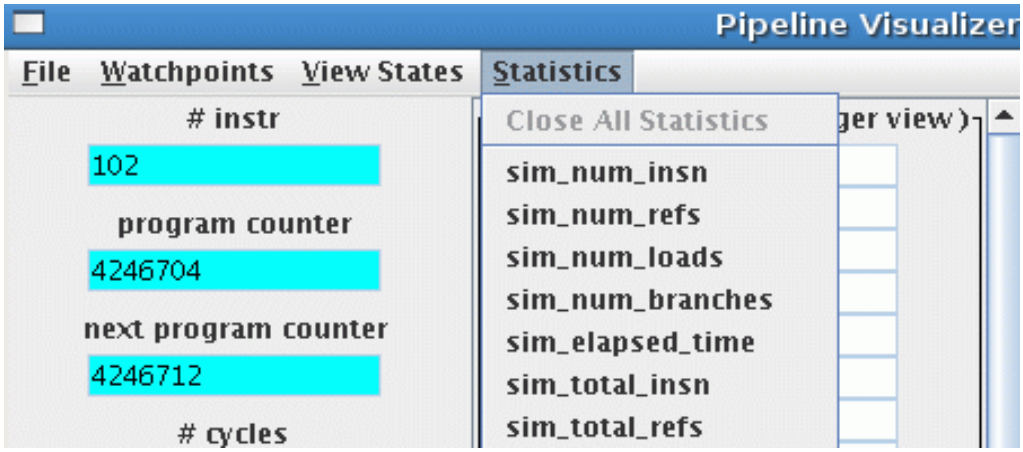


Figure 2.11: The statistics menu.

the instruction, in assembly. The row labeled *CPC* contains the current program counter instruction. In a traditional, five-stage pipeline, this is the instruction in the fetch stage of the pipeline. The row labeled *NPC* contains the next instruction to be executed. The entries above the *CPC* entry are the last few instructions executed. Or, in a pipelined architecture, they are the instructions that are in later stages of the pipeline. These entries are in order. So, the entry one above *CPC* is the instruction that will retire immediately before it, and so on. The entries below the *NPC* entry are the upcoming instructions. These instructions are based off of the predicted next program counter value, so they may never actually reach the pipeline.

While being able to look at individual state contents can be helpful in understanding the detailed behavior of a program through a pipeline, sometimes information can be lost when looking at such a low-level representation. This panel presents instructions in a human readable form. By looking at both the high level representation (disassembled instructions) and the low level representation (individual state contents), the user can connect the two, and have a better understanding of what is going on within the pipeline. In addition, the disassembled instructions alert the user to the upcoming instructions.

CHAPTER 3

IMPLEMENTATION

The implementation of the pipeline visualization tool relies on two main components: the visualizer and the simulator. The visualizer is the GUI with which the user interacts, and its functionality was discussed in detail in Chapter 2. The visualizer and the simulator run as two different programs concurrently. However, the two processes need to communicate, as the pipeline visualizer is user driven. To accomplish the sharing of information, the two processes utilize interprocess communication.

The interaction between the visualizer and simulator are reminiscent of a client-server relationship. The user performs some action, which produces a request for information or some type of action. The visualizer relays this request to the simulator, which then responds to the request appropriately. All interaction is initiated by the visualizer. Therefore, throughout the rest of this section, we refer to the visualizer portion of the tool as *client-side* and the simulator portion of the tool as *server-side*.

The *server-side* contains more than just the simulator. It is composed of two main portions. There is the simulator, which can be any simulator, however within this discussion we refer to the SimpleScalar simulator. There is also the interface code needed to interact with the visualizer. These two portions are run as one program. The interface code hooks into the simulator code, so that the necessary information from the simulator can be retrieved. The interface code contains all of functionality needed to communicate with the visualizer. When the visualizer sends the server a command, the message is received by the interface code portion of the server, which handles the response. There is no direct communication with the client-side code and the simulator code. All communication goes through the interface code, which is hooked into the simulator framework. Some simulator-specific code is required, making it necessary to modify the simulator code. However, one of the main

goals was to minimize as much additional code required by the programmer as possible. Figure 3.1 gives an overview of the communication flow.

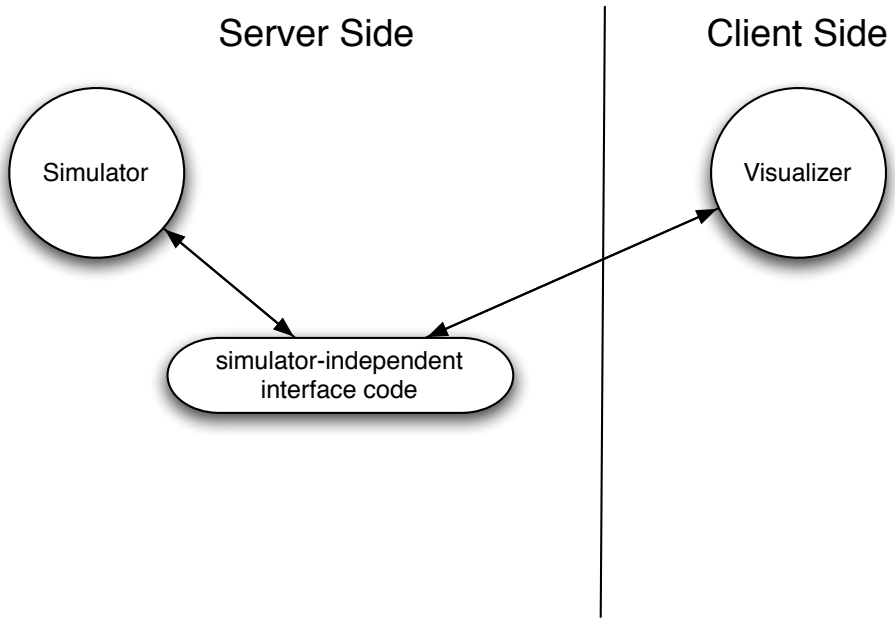


Figure 3.1: The communication paths within the visualization tool.

This chapter focuses on the details of implementing the pipeline visualizer, the interfacing code, as well as some of the issues encountered when interfacing the SimpleScalar simulator set with the visualizer.

3.1 Interprocess Communication

As already noted, the visualizer and the simulator are two separate programs. While it could be possible to combine the simulator and visualizer together, it makes more sense to keep the two entities separate. One of our main goals was to allow the ability to use the visualizer with any simulator. By keeping the visualizer separate from any one simulator, we help facilitate this notion and make the process of porting much simpler. In addition, by separating the simulator and visualizer into two distinct processes, we provide the capability of running the visualizer on a machine other than the one on which the simulation is running. There is a need for information exchange between the simulator and the visualizer. Since we want

to allow the two processes to run on separate machines, we chose to allow communication between the server and client using sockets.

3.1.1 Using Sockets

The two endpoints for the socket communication are the *server-side* and the *client-side*. Communication between the two endpoints is done using the TCP/IP protocol. The server-side socket is implemented not in the simulator code, but instead in the simulator-independent interfacing code. Currently, the interfacing code is implemented in C, and uses UNIX sockets for communication. When the passive socket is created on the server-side, it is bound to a particular port number. The user should be able to specify the port number to bind to at runtime. Therefore, the simulator was modified to allow the user to pass the port number as a flag. After the socket is created and bound to a port number, it listens on the socket, waiting for a connection. This connection is only made once, at the beginning of the program. While many server-client paradigms require a new connection to be made upon each request, we chose to use a persistent connection throughout the span of a particular visualization session, to not only simplify protocols, but also to reduce overhead.

The server receives and sends data by using the `read()` and `write()` functions, respectively. The `read()` function is blocking, allowing us to place a call to `read()` any time we want the server to wait for instruction from the client. Therefore, after the initial connection is accepted, the server is forced to delay initiating simulation until instructed by the client.

The visualizer is implemented using Java 1.5., and all client-side code is in Java. We use a `Socket` object to connect to the simulator, as opposed to a `ServerSocket`, since the visualizer acts a client. To instantiate a new `Socket` object, a host name and port number must be provided. We make both of these command line arguments that the user must provide when invoking a new instance of the visualizer. The requested host name is the name of the machine that the simulator is running on. The port number must be the same port number that the server's socket is bound to. Again, this connection is made only once, during the creation of the visualizer GUI.

The `Socket` class uses an `InputStream` and and `OutputStream` to input and output on the socket. These streams are raw bytes. Since we choose to deal with strings (discussed in Section 3.1.2), we want to access character strings directly, as opposed to dealing with raw bytes. Therefore, we employ the use of `BufferedReader` for input, and `PrintWriter`

for output. The `BufferedReader` is a wrapper for the `InputStream`. It not only converts the input to characters, but also buffers the input to provide for more efficient reading of characters. The `BufferedReader` object *in* is used for reading all input from the server, and the `PrintWriter` object *out* is used for all output to the server.

Once the connection is created on the client side, the client sends a command to the server, then waits for a reply. The method call *in.read()* is placed in a loop, to receive the server's response. The client reads and parses commands within this loop. Each command ends with '\0', allowing for simple delimiting. A server's response can be composed of more than one command, requiring some indication of when the client should finish reading from the socket. Once the client receives an ENDTRANS command, it knows the server has finished the particular information transaction that was in progress, and can stop reading from the socket.

3.1.2 Communication Protocol

In order for the visualizer and simulator to properly communicate, there must be a mutually understood way of representing commands and data. A detailed communication protocol was created for communication between the simulator and visualizer. Since the socket communication can be slowed down due to the network load, the transferring of data between the two endpoints can become a bottleneck. In order to reduce this bottleneck as much as possible, the protocol must be designed to keep messages as small as necessary. For this reason, all of the different possible commands are encoded as single characters. Arguments for each command are delimited by a space. While there are some features which require the ability to send whole string messages between the two endpoints, in general, we keep the command sizes to a minimum.

One of the main design choices taken was to send all messages between the two endpoints using character strings. All data must be converted to character strings before sending the data to the other endpoint. This choice was made to simplify the process of sending messages. Utilizing raw binary values can produce some complications. Some of these complications can be easily overcome, such as the two endpoints utilizing different endian formats. This problem can be addressed by converting all messages to network byte order, then converting them back to the host machine byte order once its destination is reached. However, other complications are not so easy to overcome. For instance, different languages can store the

same data type differently. An integer in Java can be one size, while an integer in C can vary among different architectures. Furthermore, complex data structures can be packed differently on different processors.

We avoid these complications by only using character strings to be used in messages. Another advantage to using character strings is that is easier to diagnose any errors involved with the communication process. A developer can understand the string representation of data types easily, and it does not require parsing of the data stream when debugging errors, either.

Appendix A defines the communication protocol in detail. The protocol covers all commands the client could request, as well as all responses the server may send. Once the communication channel has been created, the client side requests the state registration data from the server side by sending a single, no argument command, BEGCOMM. When the server receives the BEGCOMM command, it knows to send all of the state registration data. As earlier stated, the ENDTRANS command is used to notify the end of the transaction. Once the client receives all state registration data and the ENDTRANS command, it can then request the initial state of the simulator by sending the server the COMPLETEUPDATE command. Again, the server responds with the requested data. At this point, the visualizer is now initialized and can begin receiving input from the user.

3.2 State Registration and Storage

There is a state registration process which requires the server side to send all state descriptions to the visualizer. As mentioned in Section 2.1, this is because the different states used can vary from simulator to simulator. This state registration process allows the visualizer to be generalized and flexible, making it easier for a developer to add new states to the visualizer. In fact, this state registration process is the key to making the visualizer simulator-independent. The visualizer knows the characteristics of each state based solely on the registration information, providing a decoupling of the simulator and the visualizer.

When the GUI is first invoked, the user must select *begin* to allow the visualizer to begin communication with the simulator. At this point, all state is registered with the visualizer. One of the main goals of the visualizer was that it could be flexible enough to handle any new state additions that might be encountered when adding new state. Since computer architecture is always advancing with innovative, new ideas, it is hard to determine what

new microarchitectural state could be used. Instead of limiting any potential new state by assumptions, the visualizer was created to handle any state described by some predefined properties. Each state is identified as having some known properties, so that the visualizer knows the state's behaviors. In this way, no matter what new states are added or even what simulator is hooked in, the client side code does not ever need to be changed. However, due to this, it is necessary to first send the state definitions, or state registration to the visualizer.

This state registration also simplifies many things for a developer that wants to add state to the visualizer. Once the state has been completely integrated into the simulator, the developer is able to look at the provided state registration functions and clearly see everything that must be provided in order to register a state. The state registration function requires function pointers for state-specific actions like partial and complete updates, as well as copying state, and all actions needed to put a watchpoint on a state. In addition to function pointers, there is, of course variable requirements, such as state name, type, etc. By requiring all of these pieces of data to register a piece of state, it is easy to see all that is necessary to add a piece of state to the visualizer in one place, instead of having to read through documentation and code in multiple places.

The only function pointer not required by the state registration process, but required for visualization are the functions which handle state modifications. These must be carefully inserted into the simulator code, which is why they are not a part of the registration process. For instance, if one is incrementing the PC in the simulator, instead of setting PC++ in the code, one should instead create and call a *set_pc()* function, which not only sets the PC, but makes the appropriate function call to a visualizer function. The requirements for this function are simple and straightforward.

3.2.1 State Registration Process

Each time the visualizer is run, the state registration data must be sent to the visualizer. The server side code calls a function which contains all state registration for the particular simulator. It is within this function that the developer must make all state registration calls. To register a state, one must call any of the available state registration functions. A different registration function is provided for each type of state: internal, external, and block.

In Section 2.2, the three main predefined types of states were discussed: internal, external, and block states. When the developer wants to add a new state to the visualizer, he/she

must decide on one of these three choices, and call the corresponding registration function.

While each type of state has different requirements for registration, there are some common aspects of state registration. The developer must provide the state identifier of the new state. It is the developer's responsibility to ensure the state identifier is a unique character string. The state identifier is a character string as opposed to a single character to allow more unique states to be registered with the visualizer. While one could get 256 unique states with a single character value, many of those character would not be letters, or even readable. It is easier to debug the communication protocol if the state identifier is a letter which is somewhat related to the state in question. While '+' could be used for, say, the general purpose registers, when debugging, one may have to go look up what '+' stands for. Whereas, using "gpr", or simply "g", is more intuitive.

At the time of state registration, it must be specified whether the state being registered allows watchpoints. If so, the necessary watchpoint data and function pointers must be provided. These will be discussed in more detail in Section 3.6.

Certain flags and data must be provided during registration time which have no other purpose than describing how the visualizer should treat and display each state. The full state name must be provided as a character string. The full state names are utilized extensively in the visualizer. This string is only sent once, during registration. All other references to a particular state between the simulator and visualizer use the shorter state identifier. This reduces the size of communication messages. In the visualizer the full state name is provided in display menus and watchpoint lists so the user may be able to easily tell what state is being referenced. While the unique state identifiers try to be as meaningful to a particular state, sometimes it is difficult to recall that, for instance, "g" stands for the general purpose registers. Therefore, providing the full name provides convenience for the user.

Dimensions and sizes of the state must also be provided at state registration time. The visualizer supports only one and two dimensional state. We refer to state as one dimensional or two dimensional, because at the low level, most state is stored as some sort of array: an array of 32 general purpose registers, an array of pipeline registers, an array of 1GB of memory, etc. The sizes of each dimension must also be provided. For instance, if one is registering the program counter, or any other scalar state, then the dimension would need to be one and the size of the first dimension should be 1, since the state is composed of a single piece of data. For the registration of the general purpose registers, the dimension would also

be 1, but the size of the first dimension would be something like 32. By specifying the size in this way, it provides for easier modification later. If the simulator is modified so that the general purpose registers now contain 64 registers, the only server-side visualizer code that must be changed is when the state registration function call is made, the user must change the size of the first dimension. For block state, the sizes of each dimension are not required. Only the dimension is necessary.

There are also some function pointers which must be provided at the time of state registration. These function pointers are critical, as they provide the state-specific implementation of updating the visualizer. The three function pointers that must be provided are the complete update function, the partial update function, and the copy state function. The complete update function must send the proper update commands for the specific state. While we provide utility functions to construct many commands, the developer must have a basic understanding of the communication process and know the update state command syntax. Before constructing the update state command, the state's value must be converted to character representation. The update state command then sends the constructed command through the socket using the socket identifier parameter passed into the function. The partial update function performs almost identical actions. However, it only constructs and sends an update state command if the value the visualizer currently has for that piece of state is different from the value of the state now. It then sets the value of the visualizer's copy to the value sent. The final function mentioned is the copy state function. This function copies the current value of the state into the visualizer bookkeeping copy, which tracks what value the visualizer is displaying.

Registering Internal States

Each particular state type (internal, external, block) has extra requirements for registration in addition to the common features mentioned. Internal state registration requires a display flag. This flag specifies whether the internal state is always displayed or not.

Registering External States

The registration of an external group is almost identical to that of internal state. The only additional requirement during registration is creating an external state group. This feature allows one to group multiple states within the same external window. For instance,

in a traditional pipelined architecture, each of the pipeline registers have different entries. The pipeline register that comes before decode may only have entries for the instruction and the program counter. However, the pipeline register after the decode stage will have additional entries for the contents of registers. Due to this, each pipeline register is actually a different state, since they are composed of different entries. Typically, a user views the pipeline registers together. It is desirable to group all of the pipeline registers as one external "state". Therefore, when registering the pipeline registers, a new external group must be registered. To register an external group, a few pieces of information are required. The group name is required, for instance, "pipeline registers". Another key piece of information required is the number of states that will be in this group. The registration of a new external group is not explicitly communicated to the client-side. Instead, the new external group information is embedded in the first piece of state that will be in the external group. An external state registration command has a field for the external group name and the number of states in the group. When the client-side reads this, it knows the next x states will be in the same group. When registering an external state, a pointer to the external group must be passed to the registration command.

Registering Block States

In addition to the common requirements mentioned previously, block state registration requires a few more function pointers. Since the request of a new block of state is created using input from the user, it must be verified that the request is valid and makes sense. We do not want to make any assumptions about state type and cause the visualizer to be less flexible, so we do not want to do any checking on the client-side. Instead, we must provide a function that checks the validity of the user provided parameters embedded in the block state request. This function receives the first entry of the block and the number of entries for the block from the request. The function must then decide if these values are valid for the state in question. It must then create a response command that will specify whether the request was valid or not. Again, the developer must have some familiarity with the syntax of the command, however it is very straightforward.

3.2.2 Server-Side State Database

When a user calls the different state registration functions, two things happen. The first is that a data structure is created and populated with the necessary data that the server may need in the future. The registration function also creates and sends a state registration command to the client-side. Only the information the client needs is embedded in the command.

On the server side, all state data must be stored in an organized way. We chose to create a database of all state. Within this database structure are link lists for each type of state. Since each state type has different function pointers and data requirements, it is most intuitive to separate the different types into their own linked lists. In Figure 3.2, we see the definition for the state database.

```
struct state_db_t {
   struct llist isdb;  /* internal state database */
   struct llist esdb;  /* external state database */
   struct llist csdb;  /* execution control database */
   struct llist bsdb;  /* block state database */
};
```

Figure 3.2: All State Database Structure.

Each of the linked lists contains all states of the specified type. `struct llist` is a linked list of void pointers, allowing it to be used for each type of state. Note that the execution control is stored in the state database, though not necessarily considered a state. This is because the execution control must be registered at the same time as state, and since it is its own linked list, it does not get in the way of other state.

3.2.3 Client-Side State Storage

On the client-side, states are stored in the Java container, `Vector`. Similar to the server-side, each type of state (internal, external, block) is stored within a separate vector. Internal and External states are both stored as `StateContainer` objects. External states' `StateContainer` objects are further stored in `ExternalPane` objects. Block states are stored as `BlockState` objects.

Internal States

Once an internal state registration command has been received by the client, it parses the command, and instantiates a new `StateContainer` object. There are two different constructors for the `StateContainer`, one for one dimensional state, and one for two dimensional state. Each constructor has different parameter requirements. Both constructors require the full state name, state identifier, and the size of the first dimension. The one dimensional constructor also requires the always displayed flag. The two dimensional constructor also requires the size of the second dimension. All of the parameter information is parsed from the registration command.

Figure 3.3 shows the underlying design of a single one dimensional state. Everything is encompassed within a `JScrollPane`. This allows the user to view all state if it is larger than the pane it is being displayed in. Within the scroll pane is a main panel. This panel contains everything necessary to convey the state contents to user. The `JPanel` has a border around it with the full name of the state at the top of the border. We see there are two main boxes within the `JPanel`. The layout of the panel is set to a box layout. The box layout in Java will not wrap when a frame is resized. Since we want everything lined up a certain way, we need to ensure the components will stay in the positions designated to them.

The first box seen in Figure 3.3 contains an array of `JLabels`. These labels contain text describing the row within that state. The row labels are initialized to integers, beginning at 0 and increasing. The row labels can be changed by the server side, using a SETROW command. For instance, each pipeline register is internally regarded as an array of data. The first element is the program counter, the second is the actual instruction, and so on. Each element is in its own row when displayed in the visualizer. However, labeling the rows 0, 1, 2, etc., would not make sense. Instead, it's more intuitive to explicitly label the rows "PC", "instr", and so on.

The second box in 3.3 contains the array of `JTextFields`. This is where the actual state contents are displayed. When the `StateContainer` class is constructed, `JTextFields` are created and initialized. These text fields are what hold and display the actual contents of each state. The text fields are set so that the user may not modify the contents of the field. If the state is one dimensional, then a single array is created of the specified size. If a state is two dimensional, a two dimensional array is created. Whenever there is a state update, the

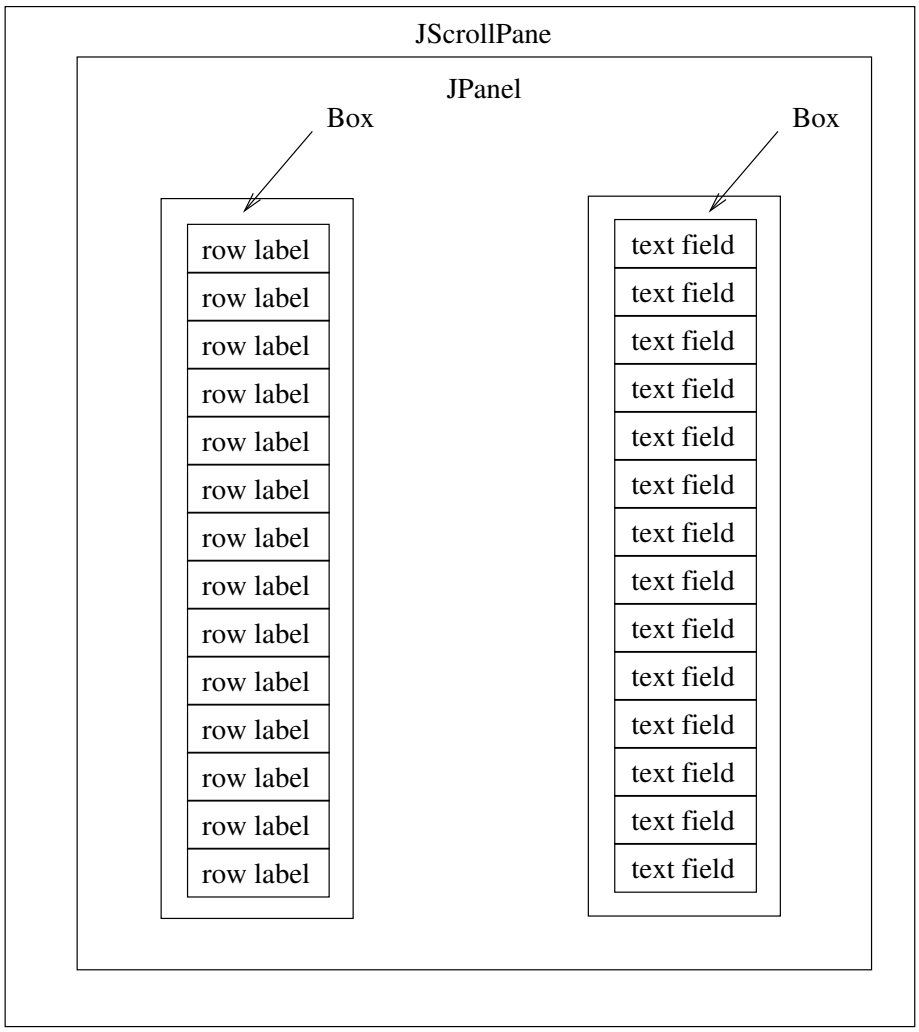


Figure 3.3: Underlying design of a one dimensional StateContainer.

visualizer calls the `setState()` methods in the `StateContainer` class. These `setState()` methods merely change the text of the necessary `JTextField`.

While each type of state contained within a `StateContainer` must display the `JTextFields`, each type has slightly different contents within the main panel. Figure 3.4 shows the underlying design of a scalar state. Since, by definition, a scalar state only contains a single piece of data, row headings and a scroll bar are unnecessary. We see the underlying design of a two dimensional state in Figure 3.5. There are two main differences between the `StateContainer` panel of one dimensional and two dimensional state. The first is the presence of multiple boxes of text fields. Instead of placing all text fields within one box, we

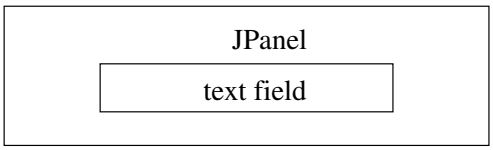


Figure 3.4: Underlying design of a scalar StateContainer.

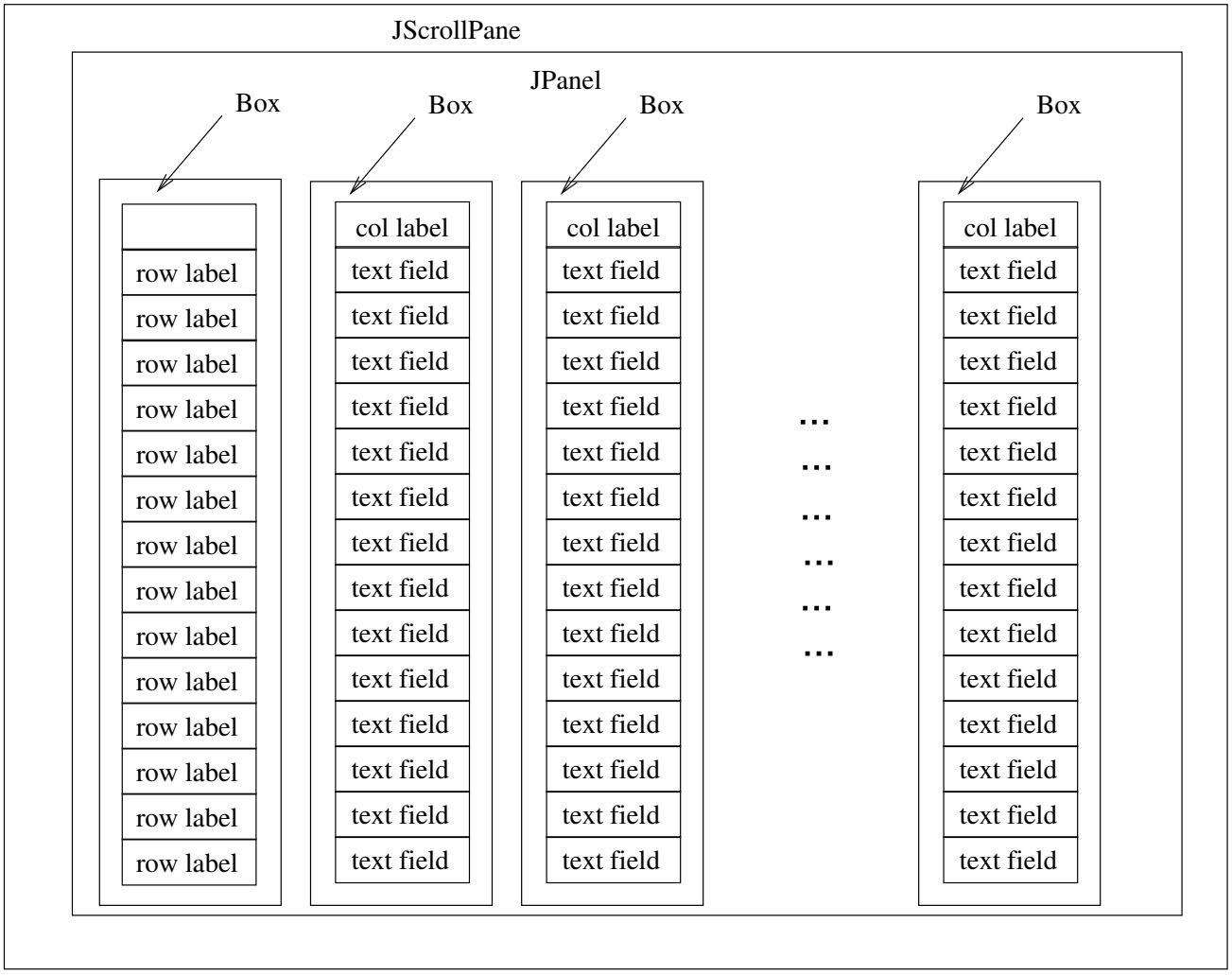


Figure 3.5: Underlying design of a two dimensional StateContainer.

illustrate the fact that the state is two dimensional by having a grid of text fields. The first text field in the first box would correspond to the data in state[0][0]. The second field in the same box would correspond to state[1][0]. Likewise, The first text field in the second box would correspond to the data in state[0][1]. The number of boxes of text fields is determined by the size of the second dimension. The other main difference is the presence of "col label" at the top of each box. These labels are the column headings for each column. The column heading for the box containing the row labels is blank. Just as in the row headings, the column headings are initialized to integers, but can be changed using the SETCOL command.

External States

All external states are stored in a `StateContainer` with the same behavior described in Section 3.2.3. However, the `StateContainer` is further stored within an `ExternalFrame` object. As noted in Section 3.2.1, external groups containing multiple external states are allowed. When a new external group is needed, an object of the class `ExternalPane` is instantiated. The `ExternalPane` class contains a vector of `StateContainers` as well as a single `ExternalFrame`. Upon state registration, an external state is only added to the `ExternalPane`. The `ExternalFrame` is created when a user selects to view an external state. This process is discussed in more detail in Section 3.3.2.

Block States

Once a block state registration command has been received by the client, it parses the command, and instantiates a new `BlockState` object. As with the `StateContainer` class, there are two constructors for `BlockState`: one for one-dimensional and one for two-dimensional. Unlike `StateContainers`, the `BlockState` class does not have an array of `JTextFields`. Due to the nature of `BlockStates`, it is not known what block size a user might request. Instead, the BlockState class merely holds the basic information and characteristics of a block state, such as the full name of the state, unique identifier, and dimension. Once a block state request is made by the user, then a pane similar to the one discussed in Section 3.2.3 is created. However, this process is discussed in more detail in Section 3.3.3.

3.3 Displaying State

Giving the user the ability to view the contents of different states is one of the most important features of the visualizer. In Section 2.2, the steps the user must take to view different states was described. In this section, we look at the underlying process of displaying the different states in the visualizer.

3.3.1 Displaying Internal States

As mentioned, there are two types of internal states: always displayed states and not always displayed states. The always displayed states are found in the left panel of the main GUI window. When an always displayed state is registered with the client-side, it is added to the left panel. The `StateContainer` class has a method called `getStateSet()`, which returns a `JScrollPane` containing the state's panel if the state is non-scalar. If the state is scalar, the JPanel is returned with no scroll bar. Therefore, to add a state to the left panel, the visualizer merely calls the `getStateSet()` method on the `StateContainer` object in question, and adds the returned component to the left panel.

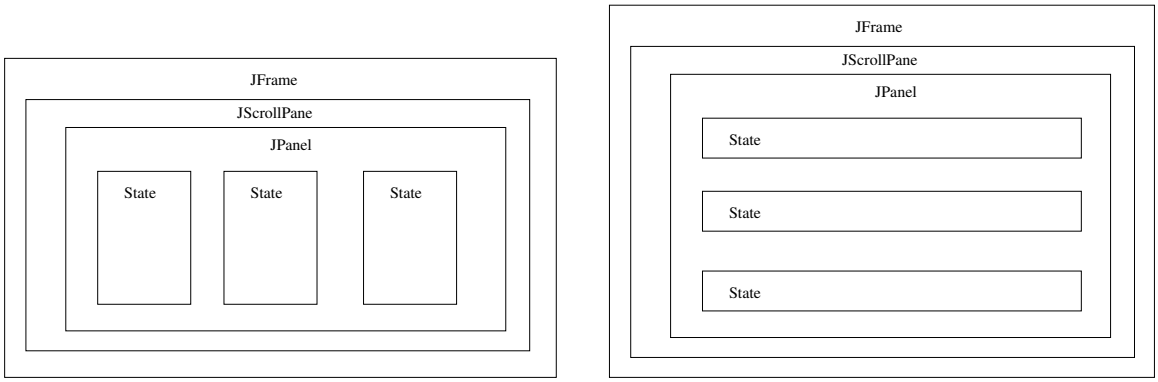
Displaying the other internal states are done in a similar fashion. When the user selects an internal state to view from the memory, the selection's `StateContainer` object is accessed. The visualizer first clears out the component currently in the center panel by calling the `removeAll()` method for the panel. The new state is then added to the center panel by adding the returned component from `getStateSet()`. There is no other work required by the visualizer, since the initial panel was created when the state was first registered.

3.3.2 Displaying External States

Displaying external states requires more work than displaying an internal state. Section 3.2.3 noted that upon registration, all `StateContainers` within a single external group are added to the same `ExternalPane` object. However, since external states are displayed in a separate window from the main GUI, a new window to properly display the external group has to be created. When an external state is selected to be viewed, the `ExternalFrame` object within the `ExternalPane class`, must be instantiated to display the state externally.

Figure 3.6 illustrates the underlying design of an `ExternalFrame`. The `ExternalFrame` class has a `JFrame` that is the new window in which external state will be displayed. The

layout of the contents of the frame are similar to a `StateContainer`. There is a main panel within a scroll pane. This allows the user to resize the frame, but still be able to scroll and see the entire contents of the frame. Within the `JPanel` are the individual states that comprise the external state group.



a) Horizontal Mode. b) Vertical Mode.

Figure 3.6: Underlying design of an external frame.

When an `ExternalFrame` object is created, the frame, scroll pane, and panel are initialized. The `ExternalFrame` class has a public method `add()`, which takes a component as a parameter and adds it to the frame's panel. The `ExternalPane` class, after instantiating the `ExternalFrame`, must call this add method for all of the states within the external state group. It does this by iterating through the vector of `StateContainers`, and calling the StateContainer method `getStateNoScroll()`. This method returns a component similar to that in Figure 3.3, except without a scroll pane. The scroll pane is omitted, since there is one enveloping the main panel in the `ExternalFrame`. Once all states have been added to the `ExternalFrame`, it is made visible to the user.

When an external state group is registered, there is an alignment flag that is provided by the server. The two options are horizontal alignment and vertical alignment. This option is set during the registration process on the server side. When a new `ExternalPane` is created, this alignment flag is passed to the constructor. The flag allows two different display layouts for external state. The first can be seen in a) of Figure 3.6. Here, the states are added to the panel in a horizontal alignment, side by side. A vertical alignment would result in a layout such as the one seen in b) of Figure 3.6. The states in this external frame as aligned

vertically, one on top of the other. The alignment has no meaning when only one state exists in a single external state group.

Whenever a user closes an external state window, the `ExternalFrame` object is freed. However, the underlying `StateContainers` are not deleted. This allows the undisplayed StateContainer to have the most recent data from the server, even when it is not displayed. This further means that when an external state is selected to be viewed, it does not have to get any information from the server. All current state contents are stored in the StateContainer.

3.3.3 Displaying Block States

When the user wishes to view a block state, the user is presented with a prompt. The user must input the portion of the state to view. The prompt is an instantiation of the `BlockPane` class. When first instantiated, the class creates a `JFrame` containing a prompt as can be seen in Figure 2.4. Once the user inputs the necessary information, and clicks the button, the visualizer constructs a MONITORBLK command and sends it to the server. The server must verify that the provided information is valid for the state in question. For example, if the user wanted to view a block of memory at the word level, requesting the block to begin at an address value not divisible by four would be invalid. However, that same address value would be valid at the byte level.

To determine if the first entry and the number of entries are valid, the server must first find the block state entry in the state database. It then uses the function pointer stored in the block state structure to call the function which checks the validity of these values. The function will return a string with the formatted response command. The server will then send the response to the client.

The response can specify that the values were invalid. If so, the response command also contains an error string which the visualizer displays, informing the user what was done wrong. The current `BlockPane` object is also freed. If the response specifies that the request was valid, the visualizer can then create the frame in which to display the block state. The same `BlockPane` object that displayed the query for user input will be modified to now display the block state. First, the frame is cleared of the current query panel. A new `StateContainer` is then created, using the block state information. Since the visualizer knows the number of entries to display from the query to the user, it can now easily create a `StateContainer` of the necessary size. The `StateContainer` is added to the frame of the

`BlockPane`. The final step is the visualizer gets the current values of the block state from the server. It does this by making a complete update request for the block state.

3.4 Relaying Updates to the Visualizer

Relaying updates to the Visualizer is vital for understanding what is happening within the simulator. The visualizer will explicitly request an update from the server-side. There are two types of updates: complete update and partial update. A complete update sends all of the data values of each piece of state. A partial update only sends the data values that have changed since the last update.

3.4.1 Internal and External Updates

When the visualizer sends a complete update command to the server, the server knows to send all data values for internal and external state. Block states and statistics are treated differently and are addressed later. Due to the fashion in which the state database was created on the server side, this is a relatively simple task to accomplish. The server merely needs to call the state-specific complete update function via the function pointers in each state entry. The algorithm for performing a complete update can be seen in Figure 3.7. Lines 1 through 4 perform a complete update for all internal states. On line 4, the call to the state's complete update function is made. The state's unique identifier string and the socket descriptor are passed to the function. The state identifier is passed so that the update state commands can be constructed. The socket descriptor is passed to the function so the function may write the state update commands directly to the socket connected to the client. Lines 8 through 13 perform a complete update for all external states. Performing a partial update for all internal and external states would require the same amount of work, only referencing `partial_update_func` instead of `complete_update_func`.

3.4.2 Block State Updates

Updates to block states must be handled differently. Since only portions of a state need to be updated with block states, it is necessary for the client to specify to the server which portion of state to send updates. Due to the necessary qualifiers, block states require their own update request commands. When the visualizer is ready to get state updates for all

```
1  foreach node in isdb do
2      it ← llist_get_data(node);
3      if it->complete_update_func != NULL then
4          it->complete_update_func(it->sid, sockfd);
5  foreach enode in esdb do
6      et ← llist_get_data(enode);
7      foreach node in et->isdb do
8          it ← llist_get_data(node);
9          if it->complete_update_func != NULL then
10             it->complete_update_func(it->sid, sockfd);
```

Figure 3.7: Sending a Complete Update

states, it sends the COMPLETEUPDATE command for the internal and external states, as described in Section 3.4.1. To request the block state updates, it must generate a state request for each block of state currently being displayed. Since the user may view more than one block state at a time, the `BlockPane` objects being displayed are stored in a vector. The visualizer then iterates through the vector, and generates a block state update request for each block state currently displayed. The block state update request includes a unique block identifier integer, which both endpoints can use to reference a specific block of state, since a state can have multiple active blocks. The block state update request also contains the first entry and number of entries of the block. The visualizer sends the block state update request to the server.

Once the server receives a request for a block state update, it parses the request and extracts the state identifier and block id, which it will use to identify the block. The server goes through the list of all block states in the state database, until it finds an entry that matches the state id. It then calls the state's update function, similar to the approach employed by internal and external states. Figure 3.8 illustrates the algorithm used by the server to send updates to the client of a block of state. On line 4, we see more parameters are passed to the complete update function than were used for the internal and external states. The function needs to know the first entry of the block, as well as the number of entries, so that it may send the state updates only for the block of entries currently being displayed

in the Visualizer. The algorithm for performing a partial state update is the same as in Figure 3.8, except `partial_update_func` is referenced instead of `complete_update_func`.

```
1 foreach node in bsdb do
2     bt ← llist_get_data(node);
3     if strcmp(bt->sid, sid) == 0 then
4         bt->complete_update_func(bt->sid, blk_id, begin_entry, num_entries, sockfd);
```

Figure 3.8: Sending a Complete Update for Block States

3.4.3 Keeping Track of State Changes

In previous sections, there has been mention of a process called a partial update of states. This action is implemented in an attempt to reduce the amount of data that must be sent between the server and client. Since each state can have more than one piece of data to update, sending updates to the visualizer could require an immense amount of data to be transferred. To reduce the amount of data needed, the server keeps track of what data the visualizer already has. This is done by having a copy of each state on the server endpoint. When any updates are sent to the visualizer, the copied state is updated with the new value. This shadow copy of state is not updated during simulation as the original state is. Instead, it is only updated with the contents of the original state when those contents are sent to the visualizer via update state commands. This is accomplished by requiring the shadow copy of the state to be updated within the state specific update functions. While the shadow copies do require more memory, it helps to reduce the amount of data sent over the socket.

3.5 Controlling Simulator Execution

One of the main ways of controlling simulator execution from the visualizer is by using execution control variables. As described in Section 2.3.1, an execution control variable provides the user with the ability to skip ahead x units of time. Typical execution control variables are cycle count and the number of committed instructions. Instead of limiting execution control to these two units, we allow control variables to be registered during the state registration process. While not treated the same way as state, the execution control

variables are stored in the state database in a separate linked list. Each execution control entry has a corresponding state identifier. Since the purpose of the execution control variable is to stop once a certain number of units have elapsed, the execution control entry must have a way of storing the value that needs to be reached. Each control entry may be of a different type, so a void pointer is provided to store the threshold value. Similar to internal, external, and block states, some function pointers must be provided at state registration time to handle the state-specific actions. Three distinct functions are required to implement execution control. A check function must be provided. It receives the value of the threshold as a string, and must verify that it is a valid value for the state. If the value is invalid, the function must construct a formatted response command indicating that the value is invalid. Otherwise, the function will return null. A set function is also required. This function receives the value of the threshold as a string, and must convert it to the correct value. The final function required is a function that checks to see if the threshold has been reached. The function returns true if the current value of the state is below the threshold value, and false otherwise. This function requires a single comparison, and is quite simple.

When the user sets the new execution control variable within the visualizer, a corresponding set execution control command is constructed. The command is sent to the server, which then parses the command. The server searches through the list of registered execution control variables, until it finds an entry with a matching state identifier. Once the matching entry is found, the state-specific check function is called to check the validity of the execution control value. If the function returns null, then the value was valid, and the new execution control value is set, using the state-specific set function. In addition, a variable is set which contains a pointer to the execution control entry of the current active execution control. At any one time, only one execution control variable may be set. A response is sent to the visualizer, specifying whether the command was valid or not. Once the new execution control variable is set, the simulator resumes execution.

There are two cases when simulator execution needs to pause and give control back to the visualizer. The first case is when an execution control threshold is met. The other is when a watchpoint is met (discussed in Section 3.6). Since, it is not known to the simulator which states are execution control, or which states have watchpoints, the simulator must call a server-side visualizer function after each update. This function, `determine_control()`, determines whether any execution control is met, and if so, gives up control to the visualizer.

Otherwise, the function immediately returns, effectively allowing the simulator to continue execution. Within `determine_control()`, we check to see if the current execution control threshold has been reached. Recall that a pointer is maintained, which points to the entry in the execution control database of the currently active execution control. We merely call the check threshold function of the active entry to see if the threshold has been reached. By maintaining the active pointer, we eliminate the need to iterate through the execution control database on each state update. Instead, we make a single function call and compare its return value to TRUE.

3.6 Breakpoints and Watchpoints

Breakpoints and Watchpoints are vital to skipping ahead execution to a specific action. The steps the user must take in adding a breakpoint and watchpoint was discussed in Section 2.3.2. In this section we will look at the underlying server-client communication required to create a watchpoint. We will also look at the way in which the server and client store the watchpoint information. Finally, we will look at the details of how to determine when a watchpoint is triggered so that simulator execution may pause.

3.6.1 Server-Side Watchpoint Storage

The server maintains a watchpoint database, similar to the state database. The watchpoint database consists of linked lists of watchpoint groups. Each watchpoint group has a unique group identifier, as well as a linked list of watchpoints. The watchpoint groups are needed because some watchpoints are dependent upon others. When any state is set, the set state function makes a call to the `determine_control()` function that checks whether or not execution should halt. In addition to checking the execution control variable, this function checks to see if a watchpoint is met. If we check every watchpoint every time, not only would performance decrease, but we would also encounter some false positives on some watchpoints being triggered. For instance, if we only want to stop when the number of instructions committed is 7, the first time the number of instructions is set to 7, the watchpoint for the number of instructions will be triggered, causing the simulation to pause execution and giving control back to the visualizer. However, once the user tells the visualizer to continue, the next time **any** state is set, the watchpoint for the number of instructions will give a

false positive, because the number of instructions is still 7. To prevent these false positives, we only want to check to see if a watchpoint is triggered if the state in question has been modified. For this reason, we need to provide the state identifier and state indices to this check execution function. We only call the watchpoint triggered function if the watchpoint state identifier is equal to the state identifier of the state just set. Furthermore, we require that the state-specific watchpoint triggered functions only return true if the indices match the watchpoint indices. This not only reduces the number of functions we call, but also prevents the false positives.

However, the selective checking causes some problems. For instance, if a word of memory is written, we want to check all watchpoints dealing with memory, not just those dealing with a word of memory. It is for this reason that we introduce the notion of watchpoint groups. The group identifier is also passed to the `determine_control()` function. When iterating through the watchpoint database, we check **all** watchpoints within the specified watchpoint group, not just the one with the same state identifier.

Figure 3.9 illustrates the layout of the watchpoint database. Each of the watchpoint entries are a structure containing all of the necessary information for setting, checking, and deleting watchpoints. The watchpoint structure contains the state identifier which is the object of the watchpoint. There are two index data elements for support of up to two dimensional state. There is a bit which designates whether the watchpoint is conditional or unconditional. There is a unique integer watchpoint identifier associated with each watchpoint. This unique number is provided by the visualizer upon requesting a new watchpoint, and is used to identify the particular watchpoint between the two endpoints. Using the state identifier is not possible, since there can be more than one watchpoint associated with a single piece of state, and it is necessary to be able to distinguish between different watchpoints. Within each watchpoint entry is a pointer which will store the watchpoint value. Since the value can be of any type, depending on the state, we make the pointer of void type, and require state-specific functions to access and modify the pointer.

As noted in Section 2.3.2, two advanced options are implemented for breakpoints. An ignore count can be set, specifying that a watchpoint should be ignored the first n times it is met. The watchpoint structure has a data member for the ignore count. Also, additional comparison types are allowed, aside from the traditional equal to. A field for the comparison type is also in the watchpoint data structure.

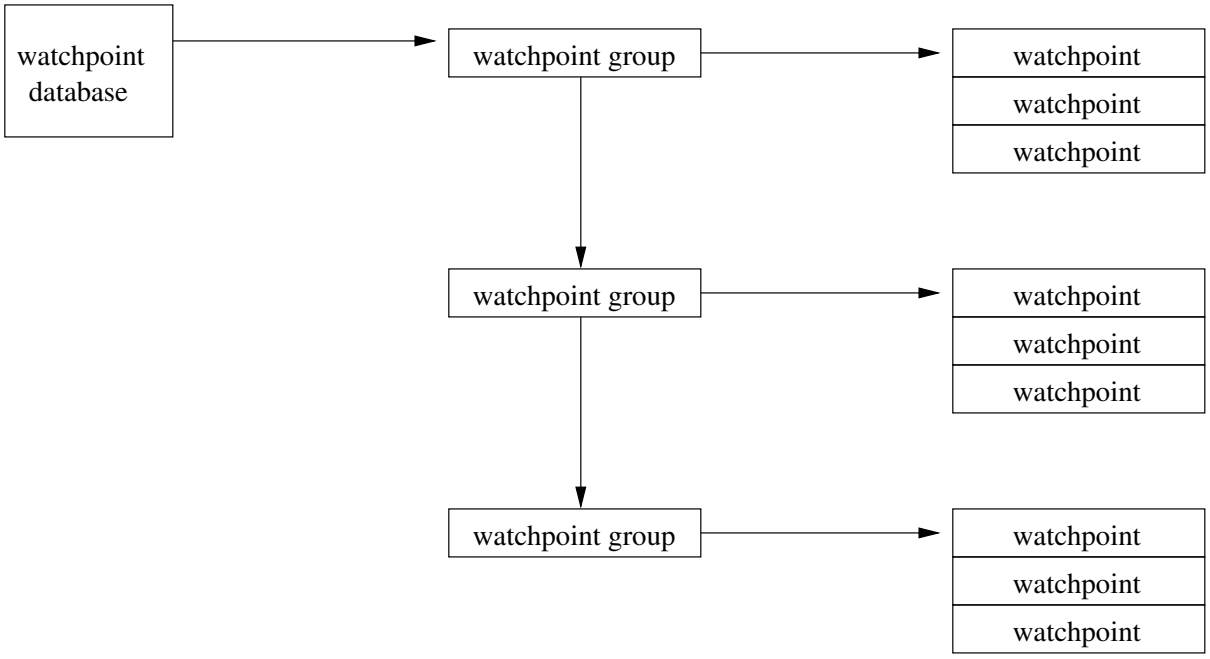


Figure 3.9: The server-side watchpoint database design.

All of the previously mentioned data member values of the watchpoint structure are set when a new watchpoint command is received and parsed. However, there are still other members of the watchpoint data structure which are state-specific, and must be extracted out of the state's database entry. Four function pointers are required to manage the state-specific actions that the generalized server cannot handle. While requiring the implementation of these functions puts more work on the developer, it is necessary to keep the visualizer simulator-independent. The implementation of setting, checking, and deleting watchpoints was done in such a way to make these functions as simple as possible.

As noted in Section 3.2.1, watchpoint related items were required during the state registration process. There are four different actions required for implementing watchpoints that are state specific. We require the developer to provide these four functions. The first of these functions required is a `wp_valid()` function. This function receives the string value of the watchpoint as well as two indices. The function must check the validity of these three values. For instance, if the watchpoint is for the program counter state and the provided value is -1 or even "bob", then the value is invalid. Likewise, if the watchpoint is on a general purpose register, and the index is set out of bounds, then the index is invalid. Though two

indices are required, for some states, such as one dimensional or scalar state, these indices are irrelevant. However, due to the nature of our storage structure, the valid function for all states must have the same function prototype. In cases where the indices are irrelevant, they are ignored. If all three values are valid, the function should return null. However, if any of the three values are invalid, the function should create an invalid response command using a provided function, and return the new string. Some utility functions are provided to the user for the most popular types of variables, like integers, floats, doubles, and their unsigned versions. By calling these utility functions, the state-specific function does not have to implement its own checks for common cases.

The next required state-specific function is the `wp_set()` function. This function must set the watchpoint value accordingly. The function is provided with the string value of the watchpoint command as well as the watchpoint structure. The function must allocate the pointer to the watchpoint value in the structure with the required size. For a conditional watchpoint, the function must convert the string value to the specified type, and set it as the watchpoint value. For instance, if the program counter is an unsigned int, it must convert the string to an unsigned integer using `sscanf()`, or some other method. For an unconditional watchpoint, the current value of the actual state must be set as the watchpoint value.

The server needs a way to decide whether a watchpoint has been triggered. Therefore, a state specific `wp_met()` function is required. The watchpoint entry as well as the indices of the state just set are provided to the function. The function must determine if the state just set is the subject of the watchpoint. If so, the function must make three distinct comparisons of the current value of the state and the watchpoint value. The function must first check if the current value equals the watchpoint value. If so, it must set a flag to the predefined watchpoint equal flag (`WP_EE`). If the current value is greater than the watchpoint value, then the flag is set to greater than (`WP_GT`). If neither comparison is true, the flag should be set to less than (`WP_LT`). The function should then return the flag.

The final state-specific function required for watchpoints is the `wp_clear()` function. This function is called when a watchpoint is being deleted, and must free up the watchpoint value pointer constructed during `wp_set()`. This function was made a requirement in case a watchpoint may later require a more complex data structure than a single variable. However, the function is optional during state registration. If null is passed as the clear function, a simple clear utility function is instead called.

3.6.2 Client-Side Watchpoint Storage

On the client side, all watchpoints must be kept track of as well, so that the user can view all active watchpoints in the visualizer and delete them, as needed. All watchpoints are stored in the `WatchPointList` class. The `WatchPointList` class maintains the list of all active watchpoints. In addition, the implementation of all methods pertaining to watchpoints are located in this class. `WatchPointList` stores all watchpoints in a vector of `WatchPointData` objects. Each watchpoint data object contains all of the necessary information to display a watchpoint to the user. Just as in the server-side structure, `WatchPointData` contains the state identifier, indices, comparison type, conditional bit, ignore count, value, and watchpoint identifier. Here, the watchpoint value is stored as a string, as the visualizer has no knowledge of the underlying type of each state. In addition, the `WatchPointData` object has some auxiliary information needed to properly prompt the user for input. The data for a WatchPointData object is populated when a user selects the *Add Watchpoint* or *Add Breakpoint* options in the Watchpoint menu and enters in the necessary data.

3.6.3 Watchpoint Creation Process

When the user selects the *Add Watchpoint* option in the watchpoint menu, a prompt is presented to the user requesting the details of the desired watchpoint. A set watchpoint command is constructed from this user input. When the user selects the *Add Breakpoint* option, similar actions will be taken since a breakpoint is a special case of a watchpoint. Once the watchpoint command is constructed, it is sent to the server. Before adding the new watchpoint, the server must first verify the validity of the watchpoint request. Since the construction of the watchpoint command was is from user input, it cannot be assumed that the watchpoint is valid. A verification process takes place that determines whether the value and indices for the watchpoint are valid. The server will then send a watchpoint response to the client, specifying whether the request was valid or not. If valid, both the visualizer and the server store the new watchpoint data in their respective storage structures. Deletion of a watchpoint requires similar communication. Once the user has selected a watchpoint to be deleted, a watchpoint delete command is sent to the server. The server will respond with a command specifying whether the watchpoint was deleted or not. In the following sections we look at the process of constructing and responding to these commands on both the server

and client sides.

Server-side Watchpoint Creation

Once the server receives a set watchpoint request from the visualizer, it must parse the command and determine if it is valid. The state identifier is extracted from the command. The server finds the entry in the state database with a matching state identifier. Once it finds the state, it must first verify that the state allows watchpoints. A watchpoint enabled flag was provided for each state during the state registration process. This flag indicates whether or not a watchpoint is allowed to be set on the state. This feature is provided because watchpoints do not make sense on all pieces of state within the visualizer. Also, setting watchpoints on some pieces of state add too much complexity to the visualizer with little results. For example, setting a watchpoint on a disassembled instruction would require either reassembling the string back into machine code to perform the watchpoint check, or disassembling the compared instruction every time. This would clearly increase the complexity of a watchpoint check. In addition, putting a watchpoint on a disassembled instruction makes little sense, as the user would most likely just want to know when a particular program address is encountered. If the watchpoint enabled flag is set to false for the state, then the watchpoint request is invalid, and the server responds to the visualizer by sending a constructed invalid watchpoint command.

If the state allows watchpoints, the server then calls the state-specific watchpoint valid function. If the function returns null, then the watchpoint is valid, and can be added to the watchpoint database. A new watchpoint entry is created, and its structure is populated not only with the information provided by the watchpoint command, but also with the watchpoint function pointers found in the state's entry in the state database. The state's set watchpoint function is called to set the watchpoint value. The new watchpoint entry must now be added to the watchpoint database. Recall from Figure 3.9, that the watchpoint database contains a linked list of watchpoint group entries, which, in turn, contain a linked list of actual watchpoints. To add a watchpoint entry to the database, we must first determine if there is already an entry for the state's watchpoint group. If so, we merely add the new watchpoint onto that group's linked list. Otherwise, a new watchpoint group is created and added to the group linked list. The watchpoint entry can then be added to the new group's linked list.

Client-side Watchpoint Creation

There are two phases of creating watchpoints on the client side. The first requires input by the user. When the user selects the *Add Watchpoint* option from the Watchpoint menu, a prompt is presented to the user. The *Add Breakpoint* option presents the same prompt to the user, but with some values pre-filled in, since a breakpoint is a special case of a watchpoint. Once the user enters all specified information, a new `WatchPointData` object is created and added to the `WatchPointList`. A set watchpoint command is constructed from the `WatchPointData` object and sent to the server.

The second phase of creating a watchpoint depends on the response from the server. The server will reply to the set watchpoint command with a valid or invalid response. If the response informs the client that the watchpoint was valid, nothing more is done. However, if the response says the request was invalid, the watchpoint is deleted from the `WatchPointList`. In addition, the user is presented with an error message.

3.6.4 Watchpoint Deletion Process

The deletion process of a watchpoint is similar to the creation process. When the user selects *Delete Watchpoint* from the watchpoint menu, the user is presented with a prompt box. The prompt contains a list of all watchpoints, in an easily readable form, with the unique watchpoint identifiers associated with them. Once the user selects a watchpoint identifier to delete, the client constructs the corresponding delete watchpoint command and sends it to the server. The server parses the command, and searches through the watchpoint database, until it finds a watchpoint entry with the same unique watchpoint identifier. It then deletes the watchpoint from the database, and also removes the watchpoint group entry, if its linked list is now empty. Upon completion, the server responds to the visualizer with either a success or failure. Once the client receives the response from the server, it deletes the `WatchPointData` from the `WatchPointList`.

3.6.5 Determining When a Watchpoint is Triggered

The server needs to be able to determine when a watchpoint has been triggered, so that it may pause execution of the simulator and give control back to the client. After each state update within the simulator, the `determine_control()` function is called. At this point,

all active watchpoints must be checked. This is accomplished by calling the watchpoint triggered functions for each watchpoint. Since this check is made on each state update, we want to make the algorithm as efficient as possible. As noted earlier, we provide the state identifier, indices, and watchpoint group identifier for the state that was just set. This provides us the capability to call fewer watchpoint triggered functions.

The algorithm for determining if a watchpoint is set can be seen in Figure 3.10. Using the provided watchpoint group identifier, `gid`, we iterate through the linked list of watchpoint groups until a match is found (lines 1 through 3). Once a matching group identifier is found, only then do we iterate through the list of actual watchpoints, calling their respective watchpoint triggered functions (`wp_met()`). The watchpoint triggered functions return a mask defining the relationship between the current value of the state and the watchpoint. However, it still needs to be determined if the watchpoint has been met. This is accomplished with the `WP_CMP_TRUE()` macro. This macro utilizes the mask and the watchpoint comparison type to determine if the watchpoint has been triggered. The macro returns `TRUE` or `FALSE`. If the watchpoint was triggered, the ignore count must be decremented (line 8). Only if the ignore count for the watchpoint is below zero is should the simulator execution be paused. Note that on line 14, we break from the outer loop. This means that we **only** call the watchpoint triggered functions for the watchpoints in a single watchpoint group. In most cases, this will be a minimal number of watchpoints, in comparison to all currently active watchpoints.

3.7 Data Highlighting

Highlighting is implemented in two ways: it can be explicitly invoked by the server-side, enabling the use of the SETCOLOR command, or it may be implicitly invoked by the client. The most often used type of highlighting, data highlighting, is implicitly invoked on the client side. Whenever any type of state update is received by the visualizer, it is understood that this piece of data needs to be highlighted. If data highlighting was only possible by the server sending a command to change the color of a state, then the amount of data sent between the server and client would double. Since we want to reduce as much communication as possible, we instead implement data highlighting on the client side. The visualizer will automatically change the color of any updated state to the data highlighting color. The

```
1  foreach gnode in grpll do
2      wpgrp ← llist_get_data(gnode);
3      if gnode->gid == gid then
4          foreach node in wpll do
5              wpe ← llist_get_data(node);
6              ret ← wpe->wp_met(wpe, sid, i1, i2);
7              iret ← WP_CMP_TRUE(wpe->wp_cmp, ret);
8              if iret == TRUE then wpe->wp_cnt −−;
9              if wpe->wp_cnt < 0 then
10                 wpe->wp_cnt = 0;
11                 grp_ret = TRUE;
12         break;
13 return grp_ret ;
```

Figure 3.10: Determining if a watchpoint is triggered.

visualizer also needs to be able to undo all data highlighting when the next update stage occurs. Therefore, the visualizer stores all state updates and set color commands received during an update stage. Immediately before the next update stage occurs, the visualizer goes the list of all previous updates, and resets the color of each state. If this is not done, then a piece of state could falsely be highlighted as being updated even if it had not been recently updated.

Forwarded data highlighting and error highlighting are both implemented by the server explicitly invoking set color commands and sending them to the visualizer. Even though forwarded data is specific to pipelined architectures, it is handled by the visualizer in a generic manner. The server-side code takes special note when a data value will be forwarded to the execute stage, creating a SETCOLOR command. It adds this newly created command to a linked list of color commands maintained by the server-side visualizer code. After all updates are sent, the set color commands are also sent to the client. When the visualizer receives a set color command, it parses the command, to determine which state and which color to set the data to.

Error highlighting is implemented in a similar manner. As described in Section 2.4.3, within the five-stage, in-order pipelined simulator, there is checking built into the simulator

to ensure that an instruction was executed correctly in the pipeline. This is accomplished by performing a functional simulation and saving the results in the front of the pipeline. During the last stage of the pipeline, the results are compared to the functional results. If any piece of data is incorrect, a SETCOLOR command is created for that piece of data, and is added to the set color linked list. Due to the nature of the error detection, simulation is immediately halted, and the updates, including the set color commands, are sent to the visualizer.

Since there is a framework in place which can store set color commands, and automatically sends them on state updates, it allows for additional highlighting types to be later implemented. Any new highlighting types will require no modification to the client-side code, since the highlighting is handled using a generic set color command.

3.8 Statistics

Statistics are registered with the visualizer during the state registration process. Statistics are extracted from the simulator defined statistics. There is no way that a statistic can be explicitly defined with the visualizer. If the developer wants to register a new statistic with the visualizer, it must be done through the simulator.

When the user wants to view the value of a particular statistic, they must select it from the Statistics menu in the visualizer. A statistic request is then sent to the server, requesting the value of a statistic. When the statistic was first registered with the visualizer, a unique id was assigned to it. All references to statistics use the unique id. The server iterates through the statistics database, until an entry with a matching identifier is found. It then converts the statistic to a string value, and sends the statistic to the visualizer. The visualizer will then create a new window to display the statistic and its value. Whenever the client requests a state update, it goes through its list of statistics, and checks to see if any are currently being displayed. For each statistic currently displayed to the user, a request is made for an update of the statistic. However, if no statistics are currently being viewed, no updates for the statistic are requested. In this way, we can reduce communication costs between the two endpoints of the visualizer.

CHAPTER 4

SIMULATOR MODIFICATIONS

Thus far, we have discussed the creation of the visualization tool. While making mention of some required modifications to the simulator, we have deferred discussion of it. In this chapter, we will discuss the modifications required when interfacing the visualizer into a new simulator framework. As an example, we will briefly discuss the modifications made to the SimpleScalar Simulator suite [1], as well as some challenges involved when interfacing SimpleScalar with the visualizer.

4.1 General Requirements

While we try to keep as much of the visualizer code simulator-independent as possible, the visualizer requires some simulator-specific code to properly function. The most important aspect of interfacing the visualizer into a new simulation framework is the ability to access states as well as handle their updates. The proper functionality of the visualizer relies on the assumption that the visualizer is given control via a function call after each single piece of state is set.

Therefore, when interfacing a new simulator, a collection of set state functions must be created, one for each type of state. For instance, there should be something similar to `set_gpr()` to set general purpose register values, `set_pc()` to set the program counter, and so on. Within these set state functions, the value of the state must be set just as originally done within the simulator. After the value is set, the function must call the visualization function which determines whether simulation control should continue or not. Again, this is required after any state is set because it is not known which states have watchpoints or which states are execution control variables.

Another requirement is that duplicate copies of all state need to be created so that the

visualizer can access the state when sending updates. These copies need to be kept up-to-date with the actual pieces of state. Each time a piece of state is set, the duplicate copy also needs to be set. The simulator must initialize these states before simulation begins. We choose to require that there be duplicates of states instead of pointers to the original to aid in ensuring that the set state functions are the only place where state is changed. If we merely used pointers of the simulator state, then it would be difficult to determine that all state changes are done through the set state functions.

Due to the need of up to date duplicate copies, the set state functions have to perform four actions. The function must first verify the duplicate state value is the same as the simulator state value before the state is set. In this way, we can easily detect if the state was changed outside of the set state function. The next step requires the function to perform the state change on the simulator state, followed by performing the same state change on the duplicate copy to ensure consistency. Finally, the function should call the visualizer control function. In Figure 4.1, we see an example implementation of a set general purpose register function.

```
void set_gpr(regs_t* regs, int index, int value)
{
   /* consistency checks */
   if(regs->regs_R[index] != check_regs.regs_R[index])
      consistency_error(``gpr'');

   /* perform state change on original state */
   regs->regs_R[index] = value;

   /* perform state change on duplicate copy of state */
   check_regs.regs_R[index] = value;

   /* call execution control function in visualizer */
   determine_control(``g'', ``g'', index, 0);
}
```

Figure 4.1: Example function to handle all general purpose register state changes.

The simulator must be modified to require a port number as a command line argument. Once the simulator is instantiated, it must call the visualizer initialization function and pass

the port number. This function will create the socket, and begin communication with the visualizer.

In addition to the interfacing requirement, each piece of state must be registered with the visualizer, requiring the implementation of the necessary functions discussed in Chapter 3.

4.2 SimpleScalar Modifications

When creating the visualization tool, we chose to first integrate it with the SimpleScalar tool set. SimpleScalar is widely used in computer architecture research to gather accurate performance data [1]. The tool set allows one to write a new simulator, or use one of the provided simulators, ranging from a simple functional simulator, to a detailed and complex out-of-order pipeline simulator.

In the SimpleScalar tool suite, instructions are simulated by breaking them down to individual state changes. For instance, an `add r1,r2,r3` instruction, when simulated, would be a single state change of register `r1`. These state changes are handled by macro function calls that change the state. For the most part, we were able to simply modify these macros to instead call the set state functions created for each type of state. We then implemented a base set of functions for each type of state that perform the actions described in Section 4.1. Each of these set functions not only perform the action originally performed by the macro, but also write the value to a separate copy of the state, which the server-side visualizer code utilizes when sending updates. After this bookkeeping state is also set with the value, each set state function calls the visualizer's `determine_control()` function to control simulation execution.

We first interfaced a simple functional simulator, *sim-safevis* to work with the visualizer. *sim-safevis* is identical to the provided simulator *sim-safe*, with only the slight modifications necessary to work with the visualizer. After verifying its proper functionality within the visualizer, we then sought out to interface a traditional five-stage, in-order pipeline. The interfacing process of a simple five-stage pipeline would only require the additional set state functions for the pipeline registers, as all other state functions were created for *sim-safevis*. As we can see, once a base set of functions are created to hook into the visualizer for a particular simulation framework, extension to other simulators, or adding other states requires less work.

With the release of SimpleScalar 3.0, there was no longer a provided in-order pipeline

simulator. Instead, an in-order option was provided for the out-of-order simulator, *sim-outorder*. However, using the in-order option for *sim-outorder* would not produce the desired effects. The out-of-order simulator simulates timing and scheduling correctly, however state updates are done during an early stage of the pipeline. If hooked in with the visualizer, the data in registers and memory would reflect the early execution and would not make sense.

Therefore, we chose to implement *sim-inorder*, an in-order simulator which updates memory and writes back to the registers during the relevant stages. To ensure that we implemented the pipeline correctly, we made two copies of state within the simulator: one that was cycle-accurate, and one that was functional. In the early stages of the pipeline, the instruction is executed, and the functional shadow states are set. The results are saved, and the pipeline continues normally, accessing the cycle-accurate states. When an instruction is ready to commit, a comparison of the functional simulation results is made against the cycle-accurate simulation results to ensure correct implementation of the pipeline. In this way, we are also able to provide a framework for detecting errors in the pipeline. When a state is found to be invalid, the error is reflected in the visualizer with highlighting. This functional versus actual check allowed us to also be sure we were forwarding data correctly and other details involved with a simulator.

The implementation of *sim-inorder* also allows the length of the pipeline to be of variable length. When instantiating the simulator, the user is able to specify how many cycles each main stage (fetch, decode, etc.) is broken into via command line arguments. This allows one to easily determine the effects on performance the length of a pipeline may have, without requiring extensive modifications to a simulator.

4.3 Challenges in Interfacing SimpleScalar

Unfortunately, not all state changes are made through the macros in SimpleScalar. The simulators emulate system calls, and during emulation, the macros are bypassed for some state. We found that the register file and any memory could be modified. We had to implement a way of detecting these changes, without drastically modifying their system call emulation function. Determining register file changes were simple, as we merely compared the duplicate copy versus the simulator's copy of the register file after a system call instruction was executed. If a register is discovered as having been changed, we save the new value, reset the old value directly, then call our own set functions to set the new value. This is

acceptable, as there are a small amount of registers to compare. However, memory is much larger, and comparing each byte of memory would severely hurt performance. Instead, we chose to slightly modify the memory structure within SimpleScalar. A flag for each memory page was added that signals whether a byte within the page has been set. For each page, a bitmap was created which indicates if a particular byte has been set. We also modified the macros that write memory to add a piece of code that sets a page flag, and the byte flag whenever any memory is written to. After a system call is emulated, each memory page's flag is checked. If it has been set, only then is the bitmap checked to copy the changed bytes. Now, a much smaller portion of memory is searched, making it a acceptable check after each system call.

The inconsistent macro use was discovered by consistency checks built into each set state function. Before setting any state, the duplicate state is compared against the original state, allowing easy detection of state being changed outside of the set functions. The use of these consistency checks was found to help greatly when initially interfacing the simulator and the visualizer, which is why they are required for all set state functions. Once a simulator framework is interfaced with the visualizer, adding new state to work with the visualizer is rather simple, due to the careful implementation of the visualization tool. However, the initial interfacing process requires extremely thorough checking to make sure all state changes are done with the set state functions.

CHAPTER 5

RELATED WORK

There have been numerous tools developed to graphically display different pipeline architectures. DLXView [2] is a tool designed to display a graphical representation of the DLX pipeline as described in *Computer Architecture: A Quantitative Approach* by Hennesy and Patterson [3]. The tool provides a detailed view of the data path of an instruction as it progresses through the pipeline. However, aside from which instruction is in each stage, the tool does not display the actual contents of the pipeline registers, or any other architectural state. Also, this tool was specifically designed for the DLX pipeline, limiting its ability to be used with other simulators.

GPV [4] is a pipeline visualization tool which displays a graphical view of the flow of instructions. Before the visualization tool is run, an architectural simulator is used to produce a pipe trace stream. GPV then uses this stream as its input to produce a graphical representation of the data. Similar to our visualizer, the stream input into GPV is generic ASCII text, allowing GPV to be interfaced to other simulators. The graph that GPV produces plots the instructions in program order, using a color coding scheme to illustrate how long the instruction spent in each stage and sources of stalls. GPV is also capable of plotting numeric statistics on a separate graph.

However, GPV is incapable of displaying the state of the machine in the same detail that our visualizer allows. While GPV does allow the user to see which instructions are currently in each stage, it cannot, for example, display the result after an instruction completes the execute stage, or even the contents of the pipeline registers, architected registers, or memory. In general, GPV is more focused on performance analysis and general time trends, whereas our visualizer focuses more on the debugging aspect of adding new components to a simulator and then verifying its correctness.

There are two other trace-based visualizers similar to GPV. The first is TraceVis [5], which displays a color-coded graph of instructions in program order, just as GPV does, leading it to have many of the same drawbacks as discussed with GPV. However, TraceVis differs from GPV by adding some extra functionality. TraceVis allows a user to search for a particular instruction. The tool highlights only that particular instruction, then automatically proceeds to the first instance of that instruction. Similar results can be obtained in our visualizer by utilizing the conditional breakpoint feature.

The other trace-based visualizer, Rivet [6], tries to bridge the gap between our visualizer and the other trace-based visualizers, by visualizing more than just a graph of instructions in program order. Rivet goes a little further by visualizing the different components in an out-of-order simulator. Though it does visualize more than GPV or TraceVis, Rivet still lacks the ability to display the actual contents of different architected state, as our visualizer can.

ss-viz [7] is a pipeline visualizer developed specifically for the SimpleScalar *sim-outorder* simulator. Similar to our visualizer, *ss-viz* has detailed information for components of the pipeline. However, the visualizer only has this type of information for a subset of the architectural state, whereas our visualizer is capable of providing this kind of detail for all components of a pipeline, plus any new components a developer may add. Furthermore, it seems this visualizer was developed for one particular simulator, whereas our visualizer can be interfaced with other simulators.

The visualization tool discussed in this paper was modeled after the *vista* framework, a software tool developed to not only display compiler optimization internals, but also provide interactive compilation [8]. Within *vista*, an application developer can view the graphical representation of the current function being compiled, as well as performing more advanced techniques like adding and undoing transformations. Some of the design choices made during the development of our visualization tool was based on the choices made in *vista*. For example, we chose to separate the simulator and graphical visualizer as different processes to allow remote visualization, just as done in *vista*. Furthermore, we chose to have the communication between the two endpoints manipulate ASCII text as opposed to binary values, as it proved effective within *vista*.

CHAPTER 6

FUTURE ENHANCEMENTS

The main features and functionality of the visualization tool have been implemented. These core features include the ability to control execution of the simulator, view the contents of any state, view simulator statistics, and set and delete breakpoints and watchpoints. In addition to these main features, further development is envisioned.

Some extensions to the visualizer include making current features more robust. For example, only certain simulator statistics are displayed within the visualizer. Formulas and distributed arrays are not added to the visualizer. We plan to enhance the statistics code on the server-side to easily handle these other types of statistics. Furthermore, statistics do not currently support watchpoints, however we hope to make this feature available in the future. The ability to stop when a statistic is modified would provide the user with a powerful feature. For instance, this would allow the user to skip through execution until a memory load or store is encountered, or until the first branch misprediction. Also, as noted in Section 2.3.3, when a watchpoint is set, the simulator will stop at either the watchpoint, or the execution control variable, if its threshold is encountered before the watchpoint. In the future, we plan to provide an option to disregard the execution control variables when a watchpoint is set. This option can be easily changed by selecting and de-selecting a check box.

An additional feature to save the communication streams is also planned. This feature may prove to be very useful when debugging the visualizer. Three options are envisioned. The first two allow the user to save either the server's communication stream, which contains all state updates and validity responses, or the client's communication stream, containing all update and execution control requests. If a user wants to ensure that the simulator interacts with the visualizer correctly after a modification is made to the simulator, the user can

compare an older stream of the simulator commands with the current stream. However, the third option makes this process even easier, by allowing the user to load a saved client stream, and use that stream as input to the server. The visualizer will then send all commands in the loaded stream file to the server, receiving and displaying all updates from the server. Once all commands in the file have been sent, the user is then able to again control the visualizer. This option gives the user the ability to save off an input stream, quit the visualizer, then, later quickly go back to where they left off in the program.

We have already integrated the visualizer with the SimpleScalar simulator. However, SimpleScalar lacks the ability to simulate all portions of the software system. Specifically, SimpleScalar uses emulation to perform operating system operations, limiting the capabilities of the tool set. We have found that M5 [9] is a full system simulator capable of simulating the execution of a complete operating system, as well as supporting multiprocessor models. M5 provides a modular platform for architecture research by utilizing a pervasive object-oriented design which represents major simulation structures, such as CPUs and caches, as objects.

We feel that interfacing the visualizer with such a sophisticated simulator framework would be very beneficial. The interfacing of the visualizer with M5 would require some level of porting code. As described in Section 4.1, such things as socket communication and state duplication code would need to be added to M5. In addition, M5 is implemented in C++, whereas the server-side visualizer code is currently in C. While most code should safely compile with a C++ compiler, some portability issues may arise, causing the need to modify the visualizer code. Furthermore, M5 supports the modeling of multiprocessors, which complicates some some aspects the visualizer. The concept of having one main window to control execution and viewing of state may require modification, as each processor would have its own set of states. The process of transferring state information must be carefully modified to properly identify which processor the state belongs to. As trends move increasingly towards multiprocessor design, it may prove beneficial to extend the visualizer to handle multiprocessor visualization.

CHAPTER 7

CONCLUSIONS

An architectural visualizer, such as the one described in this thesis, can be very useful when trying to understand the instruction flow of a program through a particular simulator. Displaying the contents of microarchitected state at any given point during execution allows the user to carefully inspect the current state of the simulator. Controlling the execution of the simulator provides the user with the ability to step through execution one cycle at a time and carefully monitor the behavior of the simulator. Additionally, this also allows the user to skip past portions of execution that are less interesting. The advanced features of setting breakpoints and watchpoints provide even more powerful options in controlling execution. Displaying current statistic values provides context to the user.

The visualizer can prove helpful in many aspects. Simulators are many times used to gather statistics and data on a new compiler optimization or architectural feature. When these simulations do not produce the predicted results, the reason must be determined. In many cases, the statistics produced from the simulation are not enough to diagnose the problem. However, with the use of the architectural visualizer, one could carefully examine the point of interest during simulation and gather a detailed view into the behavior of the simulator. The visualization of the simulator helps in not only the debugging of the simulator itself, but also in understanding why some code optimization does not produce the desired results. This would help expedite the study of new compiler and architecture research.

The visualizer can also be used as a teaching aid for computer architecture courses. In a modern architecture, there are many complex concepts such as cache layouts, fetch and issue mechanisms, out of order pipelines, and load-store queues which can be rather confusing when first introduced to students. With the aid of the visualizer, the student is able to interactively visualize the flow of a program through a pipeline, providing the student

with a greater understanding of the pipeline and all of its complex features.

This thesis also has numerous research contributions. In the process of interfacing SimpleScalar with the visualizer, a non-trivial in-order pipeline simulator was constructed. This simulator performs all state updates in the memory and write back stages, as well as having built in error detection to determine if the pipeline executed an instruction incorrectly. This pipeline model can also be of variable length, expediting research of how lengthened pipelines react to a particular architecture modification or compiler optimization.

The design of the both the communication protocol and the watchpointing system went beyond ensuring their functionality, as the need for efficiency was required. The communication between the client and server must be minimized as much as possible. Therefore, a compact protocol was created to require as little data to be transferred as possible. Furthermore, we employed the practice of only sending data to the visualizer if they had changed since the last update. The design of the watchpointing system took into consideration not only that it must be robust enough to handle advanced features like ignore counts and different comparison types, but also that it must be efficient.

The creation of the state registration process is what provides the visualizer with the flexibility to visualize more than one simulator. The design of this process required careful consideration, the visualizer must be able to handle the addition of any state conceivable. The design, ultimately, relies on the creation of different state categories, which specifies how the visualizer handles the states. This registration process is not only generic to handle new state additions, but also simplifies the steps necessary for a developer to register new state with the visualizer, minimizing the development burden when researching new architecture and compiler ideas.

APPENDIX A

COMMUNICATION PROTOCOLS BETWEEN THE SERVER AND CLIENT

A.1 Messages Sent from the Client to the Server

Below are the syntax definitions for all messages sent from the client to the server. These messages include all state update requests as well as execution control and watchpoint requests. If the user closes out the visualizer, the ENDCOMM command must be sent.

```
<visualization_session> ::== BEGCOMM <requests>
<requests>              ::== <update_requests> | <execution_requests> | ENDCOMM
<update_requests>       ::== COMPLETEUPDATE | PARTIALUPDATE | <blk_state_req>
                             | <stat_req>
<blk_state_req>         ::== MONITORBLK <blk_action> <state_id> <blk_id>
                             <first_entry> <num_entries>
<blk_action>            ::== ENDTRANS | COMPLETEUPDATE | PARTIALUPDATE
                             | BLOCKSTATECHECK
<stat_req>              ::== REGSTAT <stat_id>
<execution_requests>    ::== <exec_ctrl_req> | <watchpoint_req>
<exec_ctrl_req>         ::== EXECCTRL <state_id> <exec_ctrl_value>
<watchpoint_req>        ::== <add_watchpoint> | <del_watchpoint>
<add_watchpoint>        ::== SETWPOINT <wp_id> <state_id> <wp_cond> <wp_cmp>
                             <wp_cnt>  <wp_index1> <wp_index2> <wp_value>
<del_watchpoint>        ::== DELWPOINT <wp_id>
<wp_cond>               ::== TRUE | FALSE
<wp_cmp>                ::== WP_EE | WP_LT | WP_GT | WP_NE | WP_LTE | WP_GTE
```

```
<state_id>              ::== CHARACTER_STRING
<blk_id>                ::== INTEGER
<first_entry>           ::== INTEGER
<num_entries>           ::== INTEGER
<stat_id>               ::== INTEGER
<exec_ctrl_value>       ::== CHARACTER_STRING
<wp_id>                 ::== INTEGER
<wp_cnt>                ::== INTEGER
<wp_index1>             ::== INTEGER
<wp_index2>             ::== INTEGER
<wp_value>              ::== CHARACTER_STRING
```

A.2 Messages Sent from the Server to the Client

Below are the syntax definitions for all messages sent from the server to the client. These messages include all state update commands as well as valid command replies.

```
<visualization_session>   ::== <state_registration>
                               <info_communication>* ENDCOMM*
<state_registration>      ::== <state_registration_cmds>* DONEREGISTER
<state_registration_cmds> ::== <register_intern_state>
                               | <register_extern_state>
                               | <register_block_state>
                               | <register_exec_ctrl_var>
                               | <register_statistic> | <set_column>
                               | <set_row>
<register_intern_state>   ::== <reg_intern_one_dim> | <reg_intern_two_dim>
<reg_intern_one_dim>      ::== REGISTERSTATE <always_disp> <state_id>
                               <state_name> <dimension> <dim1_size>
<reg_intern_two_dim>      ::== REGISTERSTATE <always_disp> <state_id>
                               <state_name> <dimension> <dim1_size> <dim2_size>
<register_extern_state>   ::== <reg_extern_one_dim> | <reg_extern_two_dim>
<reg_extern_one_dim>      ::== REGEXTERNSTATE <ext_grp_info> <state_id>
```

```
                                <state_name> <dimension> <dim1_size>
<reg_extern_two_dim>      ::== REGEXTERNSTATE <ext_grp_info> <state_id>
                                <state_name> <dimension> <dim1_size> <dim2_size>
<ext_grp_info>            ::== 0 | (INTEGER <panel_name> <alignment>)
<register_block_state>    ::== REGBLOCKSTATE <state_id> <state_name>
                                <unit_name> <dimension> <dim2_size>
<register_exec_ctrl_var>  ::== REGISTERSTATE <state_id> <state_name>
<register_statistic>      ::== REGSTAT <stat_type> <stat_id> <stat_name>
<set_column>              ::== SETCOLUMN <state_id> <col_num>
                                <col_label>*
                                // no label value sets the column label to empty
<set_row>                 ::== SETROW <state_id> <row_num>
                                <row_label>*
                                // no label value sets the row label to empty
<info_communication>      ::== <query_response> | <update_transaction>
<query_response>          ::== <exec_ctrl_resp> | <blk_state_resp> |
                                <add_watchpoint_resp> |
                                <del_watchpoint_resp>
<exec_ctrl_resp>          ::== EXCONTROLCHECK <state_id> <valid> <err_msg>*
<blk_state_resp>          ::== BLOCKSTATECHECK <state_id> <blk_id>
                                <valid> <err_msg>*
<add_watchpoint_resp>     ::== SETWPOINT <valid> <wp_id> <state_id> <err_msg>*
<del_watchpoint_resp>     ::== DELWPOINT <valid> <wp_id> <err_msg>*
<update_transaction>      ::== (<update_cmd> | <set_color>)* ENDTRANS
<update_cmd>              ::== <blk_state_update> | <scalar_update> |
                                <1dim_update> | <2dim_update>
<blk_state_update>        ::== <2dim_blk_state_update> |
                                <1dim_blk_state_update>
<2dim_blk_state_update>   ::== UPDATESTATE <state_id> <blk_id>
                                <row_num> <col_num> <value>
<1dim_blk_state_update>   ::== UPDATESTATE <state_id> <blk_id>
                                <row_num> <value>
```

```
<scalar_update>          ::== UPDATESTATE <state_id> <state_value>
<1dim_update>            ::== UPDATESTATE <state_id> <row_num> <state_value>
<2dim_update>            ::== UPDATESTATE <state_id> <row_num>
                             <col_num> <state_value>
<set_color>              ::== <set_color_2dim> | <set_color_1dim> |
                             <set_color_scalar>
<set_color_2dim>         ::== SETCOLOR <state_id> <color_code>
                             <blk_id> <row_num> <col_num>
<set_color_1dim>         ::== SETCOLOR <state_id> <color_code>
                             <blk_id> <row_num>
<set_color_scalar>       ::== SETCOLOR <state_id> <color_code>
                             <blk_id>
<color_code>             ::== FWDCOLOR | ERRCOLOR | NRMCOLOR
<blk_id>                 ::== INTEGER
<state_value>            ::== CHARACTER_STRING
<wp_id>                  ::== INTEGER
<valid>                  ::== VALID | INVALID
<err_msg>                ::== CHARACTER_STRING
<row_num>                ::== INTEGER
<row_label>              ::== CHARACTER_STRING
<col_num>                ::== INTEGER
<col_label>              ::== CHARACTER_STRING
<stat_id>                ::== INTEGER
<stat_name>              ::== CHARACTER_STRING
<alignment>              ::== VERTICAL | HORIZONTAL
<unit_name>              ::== CHARACTER_STRING
                             // words delimited by ‘‘:’’ instead of spaces
<panel_name>             ::== CHARACTER_STRING
                             // words delimited by ‘‘:’’ instead of spaces
<state_name>             ::== CHARACTER_STRING
                             // words delimited by ‘‘:’’ instead of spaces
<dimension>              ::== 1 | 2
```

```
<dim1_size>                ::== INTEGER
<dim2_size>                ::== INTEGER
```

REFERENCES

[1] Doug Burger and Todd M. Austin. The simplescalar tool set, version 2.0. Technical Report 1342, University of Wisconsin - Madison, Computer Science Dept., June 1997. 4, 4.2

[2] Y. Zhang and G.B. Adams III. An interactive, visual simulator for the dlx pipeline. In *IEEE TCCA Newsletter*, pages 9 – 12, Sept 1997. 5

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

[4] Chriss Weaver, Kenneth C. Barr, Eric Marsman, Dan Ernst, and Todd Austin. Performance analysis using pipeline visualization. In *ISPASS*, pages 18 – 21, 2001. 5

[5] James Roberts and Craig Zilles. Tracevis: An execution trace visualization tool. In *Workshop on Modeling, Benchmarking and Simulation*, 2005. 5

[6] Chris Stolte, Robert Bosch, Pat Hanrahan, and Mendel Rosenblum. Visualizing application behavior on superscalar processors. In *InfoVis*, October 1999. 5

[7] Doug Burger, Todd M. Austin, and Stephen W. Keckler. Recent extensions to the simplescalar tool suite. *SIGMETRICS Perform. Eval. Rev.*, 31(4):4–7, 2004. 5

[8] Wankang Zhao, Baosheng Cai, David Whalley, Mark W. Bailey, Robert van Engelen, Xin Yuan, Jason D. Hiser a nd Jack W. Davidson, Kyle Gallivan, and Douglas L. Jones. Vista: a system for interactive code improvement. In *Proceedings of the joint conference on Languages, Compilers, and T ools for Embedded Systems*, pages 155–164. ACM Press, 2002. 5

[9] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The m5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, 2006. 6

BIOGRAPHICAL SKETCH

Kelley C. Jones

Kelley C. Jones was born on December 19, 1982 in New Orleans, LA. She attended the Florida State University, graduating with magna cum laude honors in the Summer of 2005 with a Bachelor of Science Degree in Computer Science. In the Spring of 2007, she received her Master of Science Degree in Computer Science from the Florida State University. Following the completion of her degree, Kelley joined the working world, and began her career in industry. Her main area of interests are compilers and computer architecture.