Effective Support of Simulation in Computer Architecture Instruction

Christopher T. Weaver, Eric Larson, Todd Austin
Advanced Computer Architecture Laboratory, University of Michigan
{chriswea, larsone, austin}@eecs.umich.edu

Abstract

The use of simulation is well established in academic and industry research as a means of evaluating architecture trade-offs. The large code base, complex architectural models, and numerous configurations of these simulators can consternate those just learning computer architecture. Even those experienced with computer architecture, may have trouble adapting a simulator to their needs, due to the code complexity and simulation method. In this paper we present tools we have developed to make simulation more accessible in the classroom by aiding the process of launching simulations, interpreting results and developing new architectural models.

1 Introduction

The use of simulation tools in computer engineering is essential due to the time overhead and cost of manufacturing prototypes. To better prepare the student, we and many others have integrated the use of architectural simulation tools into our computer organization curriculum. However, detailed simulators can be very daunting to the beginner, as they typical possess hundreds of options and thousands of lines of code. In this paper we discuss how simulators can be made more approachable to both students who are learning the fundamentals of computer architecture and those that are investigating a particular issue in the field.

In our introductory courses, users who are learning the fundaments are more concerned with running simulations, than understanding or modifying its implementation. We have found the best way to aid novice students, is to provide tools that have a simple interface and an output that allows them to clearly see what is going on. We present two graphical tools (SS-GUI and GPV) and a backend perl script that decrease the complexity of using architectural simulators.

In our more advanced courses, we often ask our students to add performance enhancing features to a microarchitectural simulator. We have found the students are best served by a simulator that is modular and simple to alter. In addition, they require a verification method to ensure their changes do not break the simulator. If bugs are detected the infrastructure should have methods to expedite the detection and correction of the error. We present the features of the Micro Architectural Simulator Environment (MASE) that make it ideally suited for class projects.

The rest of the paper is structured as follows. First we discuss the tools (SS-GUI and perl script backend) that we have developed that simplify the running of a simulation. Next we talk about the graphical pipetrace viewer (GPV) which simplifies the simulation analysis process. We then focus on MASE, which aids more advanced students in developing new architectural models. Finally, we give some concluding remarks on these tools and their use in education.

2 Launching Simulations

SS-GUI, shown in Figure 1, is a user-interface form that contains all of the fields necessary to launch a simulation. The save and load options make it possible for an instructor to setup a template for the class to use as the basis of their simulations. Presently the environment is customized to the SimpleScalar toolset [3], however the only non-generic field is the simulator options field. These fields are constructed by parsing a global configuration file that specifies the options available for the simulator. Additional features of the

GUI are enumerated below with corresponding marks on Figure 1.

1. File options- This menu allows for the loading and saving of the GUI form contents. This allows the system admin or class instructor to fill in a base line form that the student can load and alter.
2. Setting Menu- This menu bring up prompts for the form comments.
3. Simulation Settings- This section contains all the paths to the necessary components to run a simulation. This can be classified as three different types of data: configuration of the simulator, run setup, and benchmark specification. The configuration of the simulator requires the user to supply the path to the actual simulator and any configuration file to use. The run setup requires the user to supply the path of the backend run script (talked about in the next paragraph), where to run the simulation, where to store the results and how to tag the results for later inspection. Finally, the user must supply the benchmark to execute, the path to the executable and type/path of the input set to use.
4. Benchmark Selection window- The user has the option to select the benchmark from a list or type the benchmark and its options in manually. The pop-up window contains information about each of the different benchmarks that are supported (currently spec2000, spec95 and a few others). A global benchmark configuration file specifies how to run the experiments.
5. Simulator Option Scroll Window- This window contains all of the simulator options that

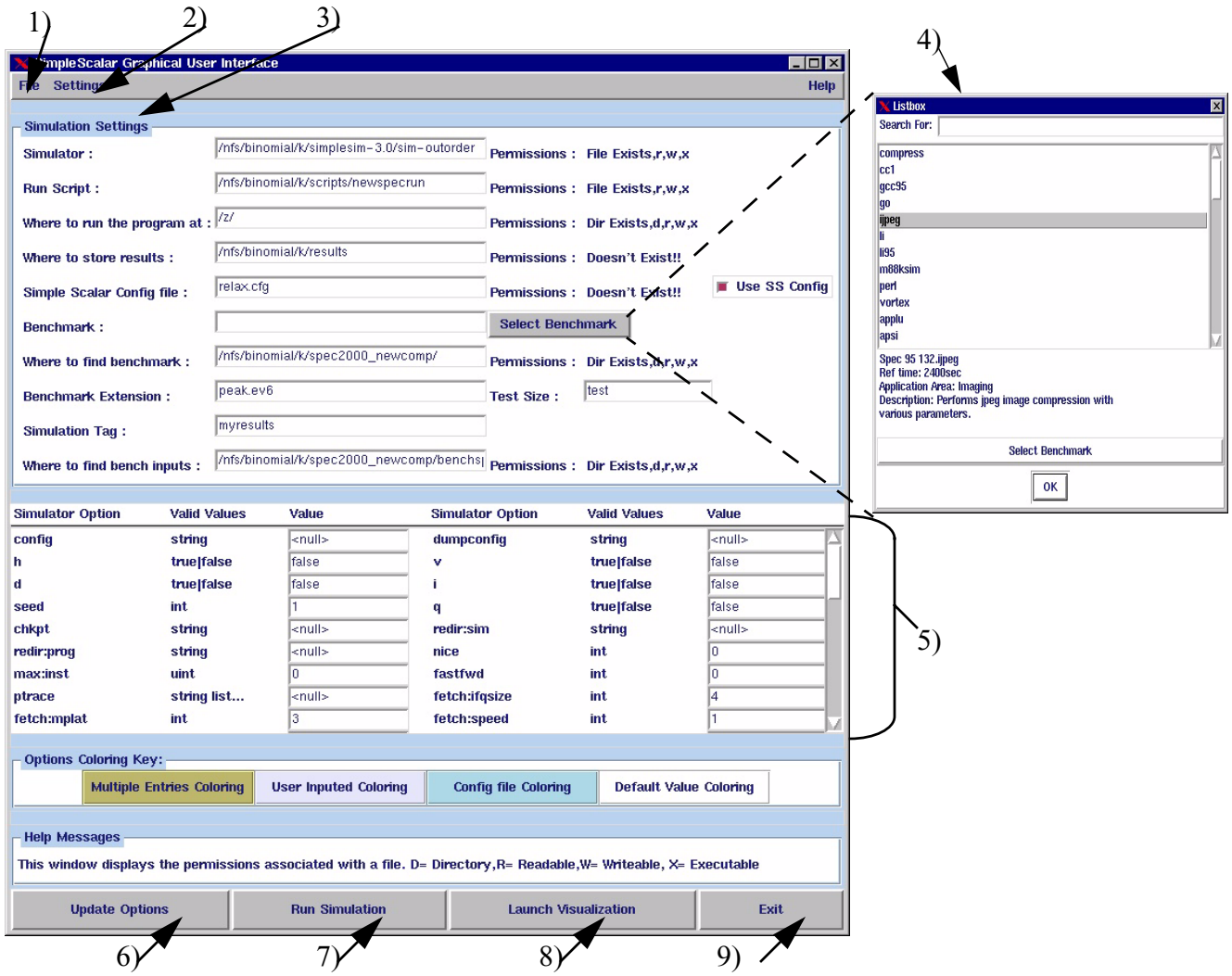

Figure 1: SS-GUI - a frontend for running simulations

are available for the current simulator. If a configuration file is specified, the options will display this value. The entries can also be modified by the user. A color guide is used to illustrate whether the value is the default, specified in the config file, entered by the user or contains multiple entries. The multiple entry fields are reserved for future usage, where the GUI can be used to generate test queues for a variety of simulator options.

6. Update Options Button- This button will run the simulator without any arguments, so that the available options are reported. The reported options are then parsed and reloaded into the Simulator Option Scroll Window.
7. Run Simulation Button- This button will run the backend perl script with the options setup in the GUI form.
8. Launch Visualization Button- The launch visualization option will run the backend perl script with a flag that causes the output to be streamed into GPV (described in the next section).
9. Exit- Exit the GUI environment.

The backend perl script contains a variety of features, however its basic function is to copy all of the simulation files to a experiment directory, launch the simulation, and copy back the results. The script contains all of the arguments need to launch the supported benchmarks (currently spec2000, spec95 and a few other benchmarks). The run script can optionally check that the simulator gave the correct output. The logs generated by the script expedite the diagnosis of run failures.

3 Interpreting Results

Figure 2 gives an overview of GPV, our pipeline viewer. An architectural simulator is used to produce a pipetrace stream. This stream contains a detailed description of the instruction flow through the machine, documenting the movement of instructions in the pipeline from "birth" to "death". In addition, the pipetrace stream denotes various other events and stages transitions that occur during an instruction's lifetime.

The pipetrace stream from the architectural simulator can be sent directly into GPV or buffered in a file for later analysis. GPV digests this information and produces a graphical representation of the data. The graph generated by GPV plots instructions in program order, denoting over the lifetime of an instruction what operation it was performing or why it was stalled. In addition, the tool is able to plot any other numeric statistics on a resource graph.

Multiple traces can be displayed on the screen at any given time for easy analysis. GPV also supports both coarse and fine grain analysis through the use of a zoom function. Color coded events, which are user definable, makes spotting potential bottlenecks a simple task. The remainder of this section will outline the tool in detail, including the main view, advanced features, trace file format, and other infrastructure with which GPV has been designed to communicate.

3.1 Main Visualization Window

The main GUI window of GPV is illustrated in Figure 2. The GUI has two main graphical display windows, the instruction window and the resource window. The instruction window plots instructions in program order on a time axis (measured in cycles). For example, the third instruction bar in Figure 2, shows the execution of an ADDQ instruction on a 4-wide Alpha simulator. As shown in the figure, this instruction is stalled in Fetch (IF) until the stall in the internal ld/st is resolved, after which it continues to completion.

This method for graphing instructions as they flow through a pipeline is a common visual representation, used in many textbooks including Hennessy and Patterson [6]. The instruction axis contains tick marks to indicate the cycle count. Additionally, the vertical axis will also display the instruction mnemonic when the window is zoomed in enough to fit legible text aside each instruction mark (typically two zooms from when the pipetrace is first loaded).

The right panel provides a legend of the coloring that is used to illustrate the instruction's flow through the different stages of the pipeline. Significant events, such as branch mispredictions or cache misses, are displayed in conjunction with the instruction's transitions through the pipeline. The use of color (with a user configurable palette) provides an effective means for spotting potential bottlenecks. A highlight option, which can flash the occurrences of a particular event, can be used as an alternative method of locating bottlenecks.

The bottom window, the resource view, displays graphs of any numeric statistic provided in the pipetrace file. GPV has been designed to plot both integer and real statistics. Up to four data sets (our current development extends this to ten) can be displayed simultaneously with color coded axes that indicate the range of the variable. Since there can be a wide variation in the data range of a statistic, a separate x-axis is provided for each one of the four resources that can be displayed at a time. Both the resource and instruction views are plotted against simulator time on the x-axis. This permits widely varying statistical data sets to be plotted within the same window. To avoid clutter, the GUI allows the selective hiding of individual resource views.

The resource view in Figure 2 is shown plotting the IPC of a simulated program. As shown in the figure, the IPC of the program starts to drop during the cache miss. Once the miss has been handled and instructions start to retire, the IPC begins to recover. The flexibility of the resource view allows the user to chose the statistics that are most valuable for performance analysis and correlate these statistics to instructions flowing through the pipeline. This simplifies the task of identifying bottlenecks, as illustrated by the relationship of the cache miss to the IPC drop in Figure 2.

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

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

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

GPV also allows users to select instructions for more information. Selecting an individual

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

Figure 2: GPV Display Window. This example shows the execution of instructions on a 4-wide Alpha ISA model. (Note: Internal micro-code operations, *i.e.* internal ld/st, are allowed to finish out of program order.)

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

4 Developing New Models

MASE (Micro Architectural Simulation Environment) is a flexible performance infrastructure to model modern out-of-order microarchitectures. It is a novel performance modeling infrastructure that is built on top of the SimpleScalar toolset [3]. MASE is most appropriate for advanced computer architecture courses where students are adding enhancements to a baseline microarchitecture and analyzing their results. MASE simplifies this process by adding a dynamic checker that can detect implementation errors, modularizing the code base improving code readability and understanding, and adding support for optimizations that are difficult to implement. Additional information on MASE can be found in [7].

4.1 Dynamic checker

The dynamic checker is used to verify that any changes or enhancements to the simulator code are indeed correct. Since not all errors directly cause an error in the output, it provides extra security that a model enhancement did not violate any microarchitectural dependencies or program semantics. In most simulators, it is difficult to determine precisely where an error occurred when there is a difference in the output. The checker will pinpoint the first instruction where a mismatch occurs, greatly reducing debugging time.

The checker resides in the commit stage, monitoring all instructions that are committed. It compares values produced from the core to the correct value. The correct value is obtained by the use of an oracle in the fetch stage. The oracle is an in-order functional simulator that has its own architectural state and memory. The oracle data is passed to the checker using a queue. In addition to checking the output value, the checker will also check (if appropriate) the PC, next PC, effective memory address, and any value written into memory. If the results match, the result will be committed to architectural state and the simulation will progress as normal. If the results do not match, an error message is printed out indicating the failing instruction along with the computed and expected values. The simulation may continue or be aborted depending on a user-controlled flag. If the simulation is allowed to continue, the oracle result will be committed to architectural state and a recovery will be initiated. The instruction with the bad result is allowed to commit (with its result corrected) in order to ensure forward progress. The remaining instructions in the pipeline are flushed and the front-end is redirected to the next instruction.

Our experience with the checker has been very positive, starting when we were implementing MASE itself. The first bug we found involved failing instructions that referred to Alpha register $31 (the zero register). Almost immediately, we were able to determine that the processing of this special register was incorrect. Once that problem was flushed out, we noticed that most of the problems dealt with conditional move instructions and how the output was incorrectly zero most of the time. We concentrated our debugging efforts at the conditional move and quickly identified that when the conditional move was not executed, it was not handled properly.

The checker was also useful in implementing a blind load speculation case study[1]. As one might expect, loads were the only instruction that failed so the error message provided by the checker did not provide as much insight as in the previous cases. Instead, we focused on the first error that was signalled. We used *gdb* to debug the simulator and set a breakpoint on the failing instruction. Once we arrived at the failing instruction, we analyzed the state of the machine at that time and were able to isolate the problem relatively quickly.

4.2 Modularized code

The MASE performance model has been divided into several files, summarized in Table 1. The rest of the SimpleScalar infrastructure is well

1. Loads are allowed to speculatively execute once their addresses are known regardless if earlier stores could overwrite the data the load is accessing [9].

Table 1: Description of MASE files

mase-checker.c	Oracle and checker.
mase-commit.c	Backend of the machine: writeback, commit, and some recovery routines
mase-debug.c	MASE-specific support for SimpleScalar's DLite! debugger
mase-decode.h	Macros used for decoding an instruction
mase-exec.c	Core of the machine: issue and execute
mase-fe.c	Frontend of the machine: fetch and dispatch
mase-macros-exec.h	Execution macros for the execute stage
mase-macros-oracle.h	Execution macros for the oracle
mase-mem.c	Memory interface functions
mase-opts.c	File contains all MASE-related options and statistics
mase-structs.h	Common MASE data structures
mase.c	Initialization routines and main simulator loop

modularized with separate files for branch predictors, caches, and memory systems. This organization allows users to focus on the part of the simulator they plan to work on without requiring intimate knowledge of the other sections. It also allows different users to work on different files without having to worry about combining changes within in a single file later[1]. It is straightforward to add enhancements since most of the new code can be placed in separate files usually requiring only slight modifications to the existing code.

Many of the features in MASE were added to make the model more realistic and representative of modern microarchitectures. A side effect of this is that it makes it easier for new users to understand how the provided code works. For example, one of the main obstacles to understanding how sim-outorder works is due to the fact that the core only simulated timing - there is no execute stage. The core of MASE executes instructions, allowing new users to track an instruction from fetch to commit without wondering where the execute stage is. To further improve readability, the execution and decoding macros have been placed into separate file, removing machine-dependent code from the bulk of the core.

1. sim-outorder.c is 4,692 lines long!

4.3 Modernized microarchitectural model

One of the goals of MASE is to modernize the baseline microarchitectural model, allowing for the creation for more accurate models. To accomplish this, we added support for several different types of optimizations or analyses that would be difficult to implement in the previous version of SimpleScalar. This section outlines some of the things we added.

A micro-functional core is added that executes instructions instead of just timing them. This allows for timing dependent computation which is necessary for accurate modeling of the mispeculated instruction stream or multiprocesssor race conditions. Lastly, it is necessary to execute instructions in the core in order to use the checker to find implementation errors such as violating register dependencies.

An oracle sits in the fetch stage of the pipeline and is a functional emulator containing its own register file and memory. Oracles are commonly used to provide "perfect" behavior to do studies that measure the maximum benefit of an optimization. A common case of this is perfect branch prediction where all branch mispredictions are eliminated. In order to provide this capability, the oracle resides in the fetch stage so it knows the correct next PC to fetch.

We added a flexible speculative state management facility that permits restarting from any instruction. The ability to restart from any

instruction allows optimizations such as load address speculation and value prediction to be implemented. In these optimizations, instructions other than branches could be mispeculated, making it necessary to restart at the offending instruction. This approach also simplifies external interrupt handling since any instruction could follow an interrupt request, forcing a rollback. The checker also uses this mechanism to recover from any errors that are detected since any instruction could potentially cause an error.

MASE uses a callback interface is used that allows the memory system (or any resource) to invoke a callback function once the memory system has determined an operation's true latency. The callback interface provides for a more flexible and accurate method for determining the latency of non-deterministic resources.

5 Related Work

There are a number of performance modeling infrastructures available to instructors today that implement various forms of these technologies. The Pentium Pro simulator [12], Dinero [5], and Cheetah [15] are examples of simulators that read external traces of instructions. Turandot [10], SMTSIM [16] and VMW [4], are simulators, like SimpleScalar, that generate instructions traces through the use of emulation. RSIM [11] is an example of a micro-functional simulator; instructions are emulated in the execution stage of the performance model. Unlike MASE, it does not have a trace-driven component in the front-end. This prevents oracle studies such as perfect branch prediction. The idea of dynamic verification at retirement was inspired by Breach's Multiscalar processor simulator [2]. Other simulation environments include SimOS [13] and SimICS [8] which focus on system-level instruction-set simulation. MINT [17] and ATOM [14] concentrate on fast instruction execution.

There are also numerous visualization infrastructures available today. The tools range from pedagogical aids to comprehensive performance analyzers. DLXview [18] is a tool that depicts the DLX pipeline that is outlined in Computer Architecture: A Quantitative Approach by John Hennessy and David Patterson [6]. It was created as part of the CASLE (Compiler/Architecture Simulation for Learning and Experimenting) project at Purdue. Another common method for visualizing the performance of a simulator is to abstract away the architecture and provide statistics based on the actual code running. CPROF [20][21] and VTUNE[19] are two examples of programs that display information such as cache misses or branch mispredictions for specific segments of code. RIVET [22-24] is a powerful display environment developed at the Stanford Computer Graphics Laboratory. The tool provides a very detailed time line view to identify problem areas. This view uses multiple levels of selection to gradually decrease the area of code being viewed, while simultaneously increasing the detail. Further background information on these tools and how GPV differs can be found in [25]. This paper also illustrates how visualization can be used for performance analysis.

6 Conclusion

We have introduced three tools in this paper that aid students using simulation in the classroom. The SS-GUI and backend perl script make it simple to launch simulations, by allowing the user to graphical select the simulator options and benchmark to simulate. The graphical pipeline viewer (GPV) aids the student in analyzing the simulation results. Finally, MASE's modularized code base and built-in checker mechanism make it ideally suited for efficient architectural model generation.

SS-GUI and GPV can be downloaded from *http://www.eecs.umich.edu/~chriswea/visualization/vis.tar* . The MASE toolset and documentation can be downloaded from *http://www.simplescalar.com/v4test.html*.

Acknowledgments

This work was supported under a National Science Foundation Graduate Fellowship and by the NSF CADRE program, grant no. EIA-9975286. Equipment support was provided by Intel.

References

[1] T. Austin. DIVA: A Dynamic Approach to Microprocessor Verification. *Journal of Instruction-Level Parallelism Vol. 2*, Jun. 2000.

[2] S. Breach. Design and Evaluation of a Multiscalar Processor. *Ph.D. thesis, University of Wisconsin-Madison*, 1999.

[3] D. Burger and T. Austin. The SimpleScalar Tool Set, Version 2.0. *University of Wisconsin Computer Sciences Technical Report #1342*, June 1997.

[4] T. Diep. VMW: A Visualization-based Microarchitecture Workbench. *Ph.D. thesis, Carnegie Mellon University,* June 1995.

[5] J. Edler and M. Hill. Dinero IV Trace-Driven Unipro-

cessor Cache Simulator. *http://www.neci.nj.nec.com/homepages/edler/d4*.

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

[7] E. Larson, S. Chatterjee, T. Austin. MASE: A Novel Infrastructure for Detailed Microarchitectural Modeling. *Proceedings of the 2001 International Symposium on Performance Analysis of Systems and Software*, Nov. 2001.

[8] P. Magnusson, F. Dahlgren, H. Grahn, M. Karlsson, F. Larsson, F. Lundholm, A. Moestedt, J. Nilsson, P. Stenström, and B. Werner. SimICS/sun4m: A Virtual Workstation. *Usenix Annual Technical Conference*, June 1998.

[9] A. Moshovos and G. Sohi. Memory Dependence Speculation Tradeoffs in Centralized, Continuous-Window Superscalar Processors. *The 6th Annual Int. Symposium on High Performance Computer Architecture*, Jan. 2000.

[10] M. Moudgill, J. Wellman, J. Moreno. Environment for PowerPC Microarchitecture Exploration. *IEEE Micro*, May/June 1999.

[11] V. Pai, P. Ranganathan, and S. Adve. RSIM Reference Manual. Version 1.0. *Technical Report 9705, Department of Electrical and Computer Engineering, Rice University*, July 1997.

[12] D. Papworth. Tuning the Pentium Pro Microarchitecture. *IEEE Micro*, April 1996.

[13] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta. Complete computer system simulation: the SIMOS approach. *IEEE Parallel & Distributed Technology: Systems & Applications*, Winter 1995.

[14] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. *Proc. of the 1994 Symposium on Programming Language Design and Implementation*, June 1994.

[15] R. Sugumar and S. Abraham. cheetah - Single-pass simulator for direct-mapped, set-associative and fully associative caches. *Unix Manual Page*, 1993.

[16] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. *Proc. of the 22nd Annual Int. Symposium on Computer Architecture*, June 1995.

[17] J. Veenstra and R. Fowler. MINT: a front end for efficient simulation of shared-memory multiprocessors. *Proc. of the 2nd Int. Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems*, Jan. 1994.

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

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

[20] A.R. Lebeck, "Cache Conscious Programming in Undergraduate Cmputer Science," ACEM SIGCSE Technical Symposium on Computer Science Education, SIGCSE '99.

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

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

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

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

[25] Chris Weaver, Kenneth C. Barr, Eric D. Marsman, Dan Ernst, and Todd Austin, "Performance Analysis Using Pipeline Visualization," *2001 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2001)*, Nov 2001.