Developing Visualization Tools for an Out-of-Order Execution Simulator

Anindo Mukherjee

March 13, 2005

Abstract

Researchers and students require visualization tools in order to efficiently evaluate microprocessor designs as they run in a simulator. The SimpleScalar toolkit, a popular and versatile program for simulating numerous processors, would benefit from such tools. While some visualization tools have been made before for SimpleScalar, few have yet supported the out-of-order execution simulator, where independent instructions are processed in parallel, out of program order. There are several factors that a designer must assess to develop visualization tools that can run on numerous hardware architectures and operating systems: code portability, the graphical user interface (GUI), and running speed. Code portability refers to the choice of programming language/languages and supported compilers and platforms, which determines the system configuration required to run the software. The design of the graphical user interface determines the effectiveness with which a user can interact with the visualization tool, and the underlying simulator itself. Both of these factors will have an effect on running speed and overall program efficiency. The visualization tool, JSimViz, addresses these factors and provides software for the student and research community.

Contents

1 Introduction

Simulation is a critical aspect of microprocessor design for both researchers and professionals. Simulation provides a method for verifying design validity as well as benchmarking efficiency. As microprocessor design has advanced rapidly in recent years, there is a clear need for simulators that can model and analyze the behavior of increasingly complex machines.

Out-of-order execution is one such design feature that was developed to minimize processor idle time and increase throughput. When running a program "in order," the instructions are processed sequentially, in the order in which they appear in the program. Due to the layout of processors, limited hardware resources, and the development of software, data hazards arise and the processor's pipeline must stall, or wait for data from a source with high latency such as memory. Out-of-order execution deals with this problem by executing the instructions out of program order if necessary to execute more instructions over the same period of time. Data hazards and false data dependencies are handled by one of several methods such as reservation stations and register renaming (Sima, 2000).

As is generally the case with processor optimization, checking for such hazards requires a great deal of logic on the chip. The processor must determine which of its fetched instructions can be sent through the pipeline's multiple execution units without conflicting with any already present and executing. The area complexity of processors that use such a feature has limited its use to large scale general purpose

CPUs. However, since the introduction of the Pentium Pro, the incorporation of out-of-order processing into processors has become mainstream, and the design techniques involved with superscalar processors have become essential knowledge for students of the computer engineering discipline. The SimpleScalar toolkit (Austin et al., 2002)

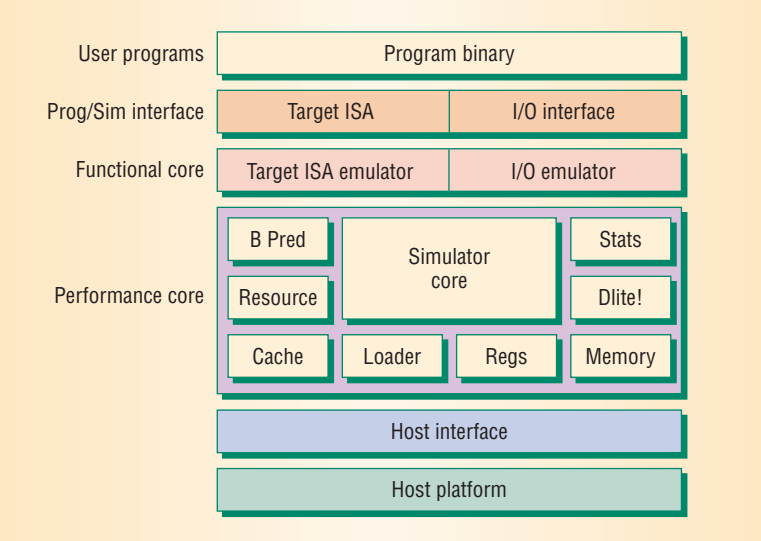

Figure 1: Layout of the SimpleScalar simulator. (Source: Austin, 2002)

is one of several simulators that support out-of-order execution. It is especially notable in that it is a well-maintained project, is widely used, and is adaptable to different architectures. The SimpleScalar simulator itself is compartmentalized, with a virtual machine structure, an Instruction Set Architecture (ISA) emulator, and an ISA interpreter. The user can compile or cross compile a binary for a given ISA, which is then interpreted by the appropriate SimpleScalar module. This then runs on a module that emulates the target ISA, and the emulator in turn interfaces with the simulator, which in turn runs through the instructions as a physical chip would, and

tabulating statistics. Because of the modular design of SimpleScalar, it is possible for a user to add specific functionality by simply modifying the appropriate section of code. This has been done with the implementation of multiprocessing (Manjikian, 2001b) as well as with ISA retargetability (Mong and Zhu, 2003). SimpleScalar can be considered the "*de facto* standard for micro-architecture simulation" (Mong and Zhu, 2003).

Given that there are so many complex issues involved with out-of-order processing and its representation, it is often difficult for student learning the material to grasp the inner workings of the processor being simulated, or the internal state of the instructions and stages. Work has been done to facilitate better usage of SimpleScalar by students (Manjikian, 2001a)(Moure et al., 2002). In "More Enhancements of the SimpleScalar Tool Set," Manjikian writes,

> These enhancements were inspired in part by research needs and in part by a desire to improve the utility of the SimpleScalar tool set in education. Undergraduate and graduate student at Queen's University have used several of these enhancements in both coursework and research, and the software for the enhancements will be released for wider use in the computer architecture community.

As indicated, this work to improve the usability of SimpleScalar by students has generally been done by instructors to aid in specific courses, but released to the community at large.

SimpleScalar can produce a pipetrace file that displays the contents of each stage during execution. This pipetrace however, is a log containing the cycle and state changes as the virtual machine processes instructions; it is machine-readable, rather than human-readable. The file is generated by specifying an optional parameter when executing SimpleScalar from the command line. SimpleScalar then enables pipetracing, and the simulator logs certain values to the tracefile as it is running. It is important to note that since SimpleScalar simulates a machine, none of this information is retained unless logged, since the contents of the virtual components such as the fetch queue and cache change over the course of program execution.

The presentation and layout of the data produced by such a simulator is important in allowing someone unfamiliar with the algorithms involved in out-of-order execution understand such processors. Without generating a pipetrace, SimpleScalar only provides the user with general execution statistics. The pipetrace itself contains information on the status of each instruction in the pipeline, but no information on any other structures within the virtual machine. SimpleScalar can also run in debug mode within a debugger, DLite, but this interface is more suited to one already familiar with the processes involved.

Numerous programs have been written to better present and orient the data the SimpleScalar obtains in a graphical user environment (Austin et al., 2002)(Moure et al., 2002). However, all of these programs have been written for a specific application, and thus have limitations that may not make them as accessible to a wide range of

students. It is therefore the goal of this project to develop visualization tools for SimpleScalar that are easily accessible to students on a variety of platforms.

JSimViz, the software developed in this project, is a Simplescalar visualization application written in Java. Requiring a modified SimpleScalar Toolkit and the Java Runtime Environment 1.4.2 or higher, JSimViz can take a pipetrace and display information from a series of pipetraces in a view that is oriented based on processor cycles. This cycle view allows the user to view instructions' entry into the superscalar processor, which states they occupy, and how they are processed. The visualizer can also display information on the Instruction Fetch Queue (IFQ), Load/Store Queue (LSQ), and Register Update Unit (RUU). The contents of these structures can also aid understanding of superscalar processors. With an emphasis on efficiency, ease of use, and simplicity, JSimViz is a tool designed for displaying useful information about out-of-order program execution. As with the related projects in this field, it is made available to any who desire access by way of an open source license, similar to that of SimpleScalar.

This paper will cover the design and implementation of JSimViz, as well as the decisions made in its development. To provide a basis for the general structure of the visualizer, I will first discuss SimpleScalar and the projects that use or modify its out-of-order simulator, sim-outorder. The subsequent chapters will examine the process used to write and test the software, a guide for the typical use case of the software, and the strengths and shortcomings of the result. Finally, a recommendation will be

made for further work on a similar visualizer or improving JSimViz.

2 Background and Previous Work

This section will discuss the simulator tools released as SimpleScalar, as well as the subsequent work that has built upon it. It will also cover similar visualization applications available for Simplescalar, and compare and contrast their respective interfaces and features.

The SimpleScalar Toolkit is comprised of a simulator, debugger, simulator tests, and the related source code. Binaries are also available from the website, but typically the sources are compiled for the host's architecture either using the SimpleScalar/Portable ISA (PISA) or SimpleScalar/Alpha for Alpha processors. The test binaries are compiled as well, and tested against the known good outputs to verify the simulator is operating properly. SimpleScalar is packaged as a set of simulators which provide different and specific functionality; for example, sim-bpred for branch prediction behavior, sim-cache for cache statistics, and so forth. The simulator that is relevant for the scope of this project is sim-outorder, which provides simulation of an out-of-order processor, as well as the functionality and statistics for the features of the other simulators. Many variables such as translation look-aside buffer (TLB) miss latency, cache size, and a variety of other factors which affect processor operation. Because of its support for out-of-order execution, and its high reconfigurability, it is the most interesting simulator to study and modify. References in this paper to SimpleScalar, or its simulator, refer to sim-outorder unless otherwise specified.

To use SimpleScalar, a user will execute sim-outorder from the command line with

the desired options that specify the parameters that the simulated machine will have. The user can run one of the provided test benchmarks or cross compile their own custom program using a special compiler. The simulator will then run through the program, tallying running statistics such as cache usage/misses, branch prediction accuracy, etc. If a pipetrace was requested at runtime, a file with the specified name will be created, and a flag will be set in the simulator. As it runs, the main simulator loop that performs the appropriate actions every clock cycle will output instruction status to this file. As mentioned earlier, the tracefile itself is not designed to be human readable; it simply contains marks to indicate the type of status change, and information that is not organized into any regular hierarchy. Simplescalar comes with a script written in PERL, pipeview.pl, that will parse a tracefile to standard output in a sort of table format. Because of the constraints of a typical terminal screen, the amount of information displayed for each instruction is kept to a bare minimum, and a fair amount of visualization is left to the user.

SimpleScalar is not the first out-of-order processor simulator, but is currently one of the most widely used for the verification of designs. This is due to the potential to support simulation of multiple ISAs, the extendability of the simulator's features, and the ease with which SimpleScalar can be ported to different systems; only minor changes should need to be made (Austin et al., 2002). Significant work has been done to expand and improve SimpleScalar functionality in the past. An example of this is the multiprocessing extension developed by Manjikian of Queen's University

(Manjikian, 2001b). In that project, support for multiprocessing and memory system was added to SimpleScalar. Also, a multi-threaded simulator, SImulator for Multi-threaded Computer Architecture (SIMCA) was developed using sim-outorder as a base (ARCTiC, 2005).

Visualization tools have been developed for SimpleScalar, including a package called ss-viz, the SimpleScalar Visualizer. Known to work with Linux, Solaris and Windows, ss-viz requires X Windows support, and installation of Tcl/Tk to implement the graphical user interface (GUI). Additionally, the visualizer uses a modified sim-outorder with graphical "hooks" to allow the tool to extract relevant data from the virtual machine. The SimpleScalar Visualizer tool has many features, but lacks an elegant interface. Also, not every user will have Tcl/Tk, and there may be issues for Windows users; the installation procedure recommends a complete Cygwin installation, which may take well over a gigabyte of space. Ss-viz is adequate for some users, but clearly may not be easily configured for any given system.

KScalar, another SimpleScalar-based visualizer with a GUI, (Moure et al., 2002) uses the KDE/Qt libraries to render its interface. The visualizer itself is fairly complete, providing the user with several view types. The user can choose a cycle view, where the focus is on the events that occur in each cycle of the simulator's execution, or an instruction view, where a user can follow an instruction's status as it enters and leaves the machine. While the KDE/Qt libraries provide a good basis for windowing, they may not be available on a wide variety of systems. Furthermore,

according to the included description of KScalar, it uses a "highly modified" version of SimpleScalar, which may impact future extendability if a user wishes to modify the simulator underlying the visualization tool. These are not necessarily shortcomings, but merely choices made to suit KScalar to its application: classroom use in undergraduate and graduate lecture courses. Naturally, KScalar may be used by any other researcher with a system that meets the requirements, but with an x86 processor and Linux required, the systems available may be limited.

3 Approach to Design

The software produced by this project was developed using general guidelines set by the principles of software design. The requirements capture exercise served to establish basic user requirements, and the typical use case for a student fairly unfamiliar with the out-of-order processors and SimpleScalar. I then developed a set of design considerations, or goals that I would use to write the software based on the results of the requirements capture and the needs that have not been addressed by the other SimpleScalar visualization tools already developed. This section will cover those stages of the design process, and conclude with a discussion of the trade-offs that were made in meeting the desired constraints of the software.

3.1 Requirements Capture

To form the basic requirements for use, I familiarized myself with the use of SimpleScalar and the PERL script used to parse pipetrace results. I used the command line arguments for SimpleScalar to specify arbitrary ranges over which to produce pipetraces, and examined the trace files themselves. A student using SimpleScalar would presumably track instructions through the pipeline to understand the program flow in the simulated processor, so I also used the PERL script pipeview.pl to display and format the results and followed the execution. I did this for the example test programs included with SimpleScalar, such as test-fmath, test-anagram, and test-printf.

As discussed earlier, the constraints of a terminal window limit the amount of

information that pipeview.pl can display at one time. The pipeview script assigns a two-letter designation to an instruction when it first enters the pipeline, and refers to that designation in a table thereafter. For example, the instruction *lw r16,0(r29)* would be assigned a value as

```
aa = '0x00400140: lw        r16,0(r29)'
```

and referred to as *aa* in all of the displayed tables. This is conceptually adequate in indicating the location of a particular instruction at a particular cycle, but relies upon the user to keep track of which abbreviations refer to which cycles. This can be somewhat confusing, and the user will probably end up writing out a note of which instructions translate to which on paper. The use of abbreviations also distances the instruction itself from the type of instruction, a factor that is critical in understanding why the processor delayed an instruction from issuing, or several instructions have been sent to execute. Because of this, I deemed it important that a user is able to see the type of instruction while looking at the events that take place in a processor cycle. Balancing the amount of information a typical user would need with the constraints of what will fit on screen in an unobtrusive application, and avoiding the problem of disorienting the user with too much information became a concern in designing the visualizer interface.

3.2 Design Considerations

The focus of the GUI design was on providing a useful but uncluttered interface. The results of the requirements capture indicated that displaying instruction information in the cycle event table itself would be ideal. The SimpleScalar Visualizer, ss-viz, also includes the contents of the IFQ, LSQ, and RUU, which may assist students in understanding the simulator, so I evaluated that to be a useful feature as well.

SimpleScalar pipetrace files are typically used to examine part of a program's execution rather than the entire program's execution; a typical program can easily take hundreds of thousands of clock cycles, and present more information than a human would need to examine personally. Hence, when a user specifies to the simulator that a pipetrace should be generated, the range would usually be small, specifying the range that the user is interested in. However, I thought that it would be useful for the program to be able to run on a large pipetrace file, and navigate to the sections that the user is interested in, rather than force the user to continually rerun the simulator and produce different ranges to examine one by one. By supporting scalability for the input, memory requirements will also decrease; if the amount of memory the visualization application is largely independent of the input, not only will it be able to handle large files, but it will be able to be run on a wide range of systems, and use a reasonable amount of memory. Fewer system requirements will make the visualizer available to a wider audience, which is a goal of the project in general.

Perhaps one of the most important design considerations was maintaining the extendability of the SimpleScalar code. The modifications and extensions done to SimpleScalar mentioned in prior sections of this paper were made by researchers. However, this does not preclude the possibility of a student modifying sim-outorder either as part of a class or for research; to allow for such an occurrence, the software should adhere to to basic requirements. First, the project should use an unmodified version of SimpleScalar as its base so that users working with the version distributed in the simulator will already be familiar with the code. Second, the project should not break any functionality that SimpleScalar already has: the hooks that the JSimViz software uses should not be intrusive and make any previously selectable options unavailable to the user.

3.3 Software Implementation Strategy

The primary software design methodology used in developing and testing the JSimViz visualizer was the spiral approach. In the spiral approach, the program is written and tested incrementally at different stages of its development, and milestones are set to indicate progress. Appropriate software milestones in software engineering are binary; they are criteria that are either met by the deadline, or unmet (Brooks, 1995). I took the time allotted for software design, implementation, and testing, and divided it into three basic stages for development:

1. Make the necessary modifications to the sim-outorder code; add the necessary hooks to SimpleScalar itself, and write additional source in C to interact with SimpleScalar (i.e. use the SimpleScalar pipetrace to generate other necessary information in addition to the normal pipetrace)

2. Develop a window-based GUI to display the data gathered by the code in the previous step, provide the user with an interface to manipulate the data view as necessary, and provide some coherence between the two parts.

3. Test the finished software with some typical cases using the test programs included with SimpleScalar.

3.4 Design Tradeoffs

By electing to use as little memory as possible, a tradeoff regarding speed had to be made. All applications that run on a computer must achieve a balance between memory 'footprint' and running speed. Making the program prohibitively slow was unacceptable, so other methods to improve speed were required. Also, a downside to using pipetraces is that only the information requested by the modified pipetrace will be available to the visualization tool; there is no way to access data in the simulator once its state has changed, or the simulator has terminated.

4 Software Developed

4.1 Overview

Given that data for a given cycle exists within the simulator only while the sim-outorder is within that particular cycle, visualization tools have two ways in which to extract the necessary information: either suspend the simulator itself, displaying only information about the current cycle, or use pipetraces to have the simulator dump data as it is running so the information is available after SimpleScalar terminates. The latter option was chosen for several reasons. First, suspending the simulator, and visualizing a state similar to a debugger would require work with DLite, and would probably elevate the level of complexity of the project, making it unfeasible in the given time frame. Second, a debugger is intended to display current information, and may not be optimal for preserving information from past cycles that may be lost in case the user wishes to view a previous cycle. In a pipetrace file, the state changes for all cycles are recorded. Note that this is not the same as a static list of the contents of each pipeline stage; for a given cycle, there may be instructions that do not change state, and are therefore not listed in a cycle. However, these instructions are still present in the pipeline. Although the pipetrace does not present a convenient table, it is the better option, and was chosen as a means of accessing data for the JSimViz visualizer.

I implemented the design in Java for several reasons: it could be assured to have

similar performance on a variety of systems provided they had an adequate version of the Java Runtime Libraries, and Java is available for almost every architecture and operating system. By using Java, the JSimViz visualizer is theoretically able to run on Windows, OS X, UNIX, and Solaris systems on any architecture that supports SimpleScalar (though JSimViz has not been exhaustively tested). X Windows (the X11 windowing protocol) is an open standard for GUI applications that was considered for this project but not used due to the unavailability of adequate documentation in the project time frame. Java's Swing libraries provide a base for "lightweight" applications– those that interface with the host OS for the rendering of windows, and the interception of user commands. Creating an application using Swing removed a great deal of the technical and logistic issues involved with creating a visual program, and was ideal.

To use JSimViz, the user will run sim-outorder requesting a pipetrace as usual. However, the modifications I made to the simulator will generate additional files that the visualization tool will use to display the information. After opening the JSimViz application, the user can open a file using a standard choose file dialog box. Due to the "lightweight" nature of Swing, the window will use the "look and feel" and widgets of the user's operating system (see figure). The user is then presented with an interface containing the previous cycle contents (top), current cycle contents (middle), and next cycle contents (bottom). A user can press the appropriate buttons to advance one cycle, or go back one cycle, as well as enter a number and go to an arbitrary cycle

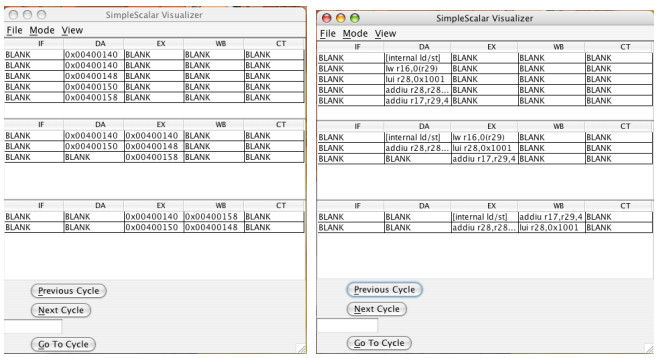

Figure 2: The JSimViz Visualizer displaying information on a program's execution in the simulator. The instructions are identical; instruction number view is on the left, assembly instruction view is on the right.

that exists within the specified pipetrace.

4.2 Software Implementation

The visualizer was implemented using the principles of object-oriented programming. Instructions were made into objects with properties that can be set such as assembly instruction, number, memory address, and so on. These instruction objects are added to pcycle objects which represent the contents of each of the pipeline stages in a given cycle. The dynamically updated tables (JTables) that display information are connected to these pcycles via the AbstractTable construct in Java. All filehandling is done with a filehandler object I wrote. This filehandler opens tracefiles, and parses them into pcycles on demand.

The ability to randomly access a file, that is, the ability to seek to an arbitrary location without having to start at the beginning and search until that point, presented a potential memory problem. A possible solution would be to create a data structure

that contained the contents of each of the pipeline stages. This may not be feasible because the contents of pipeline stages are instruction objects, and the amount of memory required to catalog all of them for each cycle would be prohibitive for very large inputs. Furthermore, a user will not need to access all parts of the file at once, so it is not necessary to keep all of the data easily accessible in memory.

The software solves this problem and meets the memory design constraint two ways. First, upon opening a pipetrace file, it creates a data structure that finds the beginning of each cycle and each new instruction and marks it. This takes O(n), but only needs to be done once for any tracefile opened. This way, whenever a cycle or instruction is needed by the program, or requested by the user, it can be directly accessed, and no further searching through the file is necessary. Second, as this marking is done, the program creates a table of "live" instructions, or instructions that are currently in the pipeline, for each cycle. As previously discussed, the pipetrace file only notes changes in instruction state. SimpleScalar's pipeview.pl PERL script keeps a hash table of which instructions are currently in the pipeline. JSimViz takes a similar approach, but extends it by creating a list of which instructions are live for any given cycle. This results in significant memory savings because though the number of cycles in the input may be high, the list of "live" instructions are simple integers, which consume little memory. These instructions are then parsed from the trace file on demand by the filehandler object.

4.3 IP and Software Licensing

Software licensing was the primary issue for this project with regards to the social and ethical context in which it is to be placed. Defining a license for software is important because it grants users certain rights to examine and modify the source code behind the executable programs. Myriads of software licenses exist for different purposes and different goals. KScalar, for example, is distributed under the GNU General Public License (GPL). The GPL grants users the right to modify and reuse code as long as the resulting code is also available under the terms of the GPL. SimpleScalar, on the other hand, uses a simpler license with fewer guidelines, and its license will be used for JSimViz and the related code produced by this project as well. This is generally termed "open source," where the original source code used to make the programs can be freely distributed, and modified under the terms of the associated license.

5 Significance and Recommendations

With the development of the JSimViz SimpleScalar Visualizer, the research and student community in computer engineering now has access to a portable visualization tool written in Java where none had existed before. The tool has fairly basic capabilities, and is limited in scope, but allows students to view the contents of a processor's pipeline over a 3-cycle span. JSimViz also allows a user to quickly navigate to the information on any desired cycle without consuming a large amount of system memory. The slightly modified version of SimpleScalar included with JSimViz also outputs auxiliary information about processor structures such as the IFQ, LSQ, and RUU. With this information, a student should be able to use JSimViz on programs and aid her understanding of the fundamental operation of out-of-order processors. JSimViz is also notable in that an interested user can run it without installing any other required libraries or utilities other than the Java Runtime Environment 1.4.2, and does not even require the user to compile the source code due to the fact that it runs in Java.

The source code to the visualizer developed in this project will also be released to allow interested parties to examine the code, and create custom modifications or derivative works. The open source software license ensures that the student and research community will benefit from this by guaranteeing access to those groups.

Considerable work could be done in the future to improve or extend the capabilities of JSimViz. Currently, the software supports viewing of three cycles at one time. If

a user deemed it necessary, he could extend this to any number of cycles within the limits of screen size very easily because of the way in which JSimViz is written. The layout of the interface itself could be improved; though Java's Swing libraries allow a look and feel native to the host operating system, the addition of a toolbar or icons could improve ease of use. Over the course of this project, the primary concern was the functionality of the application, though aesthetics often play an important role in improving user interaction speed.

Work could also be done to rewrite the base of the GUI or create a new visualization application that could support an extremely configurable processor. As it currently stands, JSimViz was written to interact with sim-outorder of the SimpleScalar toolkit, and any minor modifications made to it. However, one could create a more flexible GUI that could display a processor with any number of pipeline stages (SimpleScalar's pipeline has 5). Doing so was far beyond the scope of this project and therefore not attempted. Designing such a visualization tool would be complex, but could potentially work for simulated processors that are vastly different from SimpleScalar's default, as well as future simulators other than SimpleScalar.

References

ARCTiC (2005). Arctic labs.

Austin, T., Larson, E., and Ernst, D. (2002). Simplescalar: an infrastructure for computer system modeling. *Computer*, 35(2):59–67.

Brooks, F. (1995). *The Mythical Man-Month: Essays on Software Engineering*.

Burger, D. and Austin, T. M. (1997). The simplescalar tool set, version 2.0. *SIGARCH Comput.Archit.News*, 25(3):13–25.

Chen (2004). Kernel analysis and application of simplescalar simulator. *Journal of the Harbin Institute of Technology*, 36(5):652–654 696.

Espasa, R., Valero, M., and Smith, J. E. (1997). Out-of-order vector architectures. pages 160–170.

Liu, C.-C., Shiu, R.-M., and Chung, C.-P. (1996). Register renaming for x86 superscalar design. pages 336–343.

Manjikian, N. (2001a). More enhancements of the simplescalar tool set. *SIGARCH Comput.Archit.News*, 29(4):5–12.

Manjikian, N. (2001b). Multiprocessor enhancements of the simplescalar tool set. *SIGARCH Comput.Archit.News*, 29(1):8–15.

Manjikian, N. (2001c). Parallel simulation of multiprocessor execution: implementation and results for simplescalar. pages 147–151.

Mong, W. S. and Zhu, J. (2003). A retargetable microarchitecture simulator. pages 752–757.

Moure, J. C., Rexachs, D. I., and Luque, E. (2002). The kscalar simulator. *J.Educ.Resour.Comput.*, 2(1):73–116.

Schnarr, E. and Larus, J. R. (1998). Fast out-of-order processor simulation using memoization. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 283–294. ACM Press.

Shealy, A. R., Malloy, B. A., and Sykes, D. A. (1997). Simx86: An extensible simulator for the intel 80x86 processor family. pages 157–166.

Shen, G., Patkar, N., Ando, H., Chang, D., Chen, C., Chen, C., Chen, F., Forssell, P., Gmuender, J., Kitahara, T., Li, H., Lyon, D., Montoye, R., Peng, L., Savkar, S., Sherred, J., Simone, M., Swami, R., Tovey, D., and Williams, T. (1995). A 64b 4-issue out-of-order execution risc processor. pages 170–171, 359.

Sima, D. (2000). The design space of register renaming techniques. *Micro, IEEE*, 20(5):70–83.

Williams, T., Patkar, N., and Shen, G. (1995). Sparc64: a 64-b 64-active-instruction out-of-order-execution mcm processor. *Solid-State Circuits, IEEE Journal of*, 30(11):1215–1226.