SIMULATING AN ADVANCED SUPERSCALAR ARCHITECTURE

Adrian FLOREA, Lucian N. VINTAN,

"Lucian Blaga" University of Sibiu, Computer Science Department, No. 4, E. Cioran Street, Sibiu-2400, ROMANIA,
E-mail: aflorea@vectra.ulbsibiu.ro, vintan@jupiter.ulbsibiu.ro

Abstract: There are two paradigms that contribute for increasing the processor's performance: one based on software and the other one based on hardware. For following the evolutionary path from the last 25 years in computer architecture it is necessary to realize an integrated approach based on a synergism between technology and architecture, concepts, algorithms and methods, hardware and software applications. This paper wants to be a useful software guide dedicated to anyone who should realize a simulator - a virtual machine that evaluates and optimizes the performance of a parallel architecture - and doesn't know how to start. The simulator offers the user the opportunity to define the parameters that govern many details of the machine model's behavior.

Key words: Software Simulation, Benchmarking, Architecture, Instruction and Thread Level Parallelism, Hardware and Software Resources.

1. Introduction

The design of next generation processors is based mainly on software architecture design. Researchers traditionally use simulation techniques, to evaluate different processor pipeline configurations. First a parameterized simulator is written for the processor model; then a suite of benchmarks is executed to evaluate the performance of different configurations. Also, the processors should be designed together with the compilers that are using them: the code generated by the compiler must exploit the architectural features, otherwise the code will be inefficient.

Improving processors performance could be realized statically, helping by the compiler (through global optimization scheduling techniques) or dynamically, based on hardware methods (forwarding, combining). The actual conclusions of computer architecture researchers are that three phenomena (the clock speed, integration on single chip and exploiting the instructions and threads level parallelism - ILP and TLP) contribute for increasing the overall processor performance. The general purpose of any research about uniprocessor architecture is to extract and realize higher degrees of ILP from benchmarks compiled (optimized) for the respective architecture.

The testing programs considered (Stanford, SPEC) represent common applications computationally intensive with higher dynamic instruction counts, some of them pure recursively. These benchmarks reflect the advances in chip technologies, compilers and applications and include graphics, multimedia, compilers, sorting problems, image and sound compression, games. For example, one of the applications that runs on the last commercial processors Intel Pentium IV is the encoding of real time image captured by a digital video camera. The Stanford benchmarks were proposed by John Hennessy in 1981 and constitute one of the first generation of testing programs (≈1 million dynamic assembly instructions). In the last 12 years the Standard Performance Evaluation Corporation (SPEC) released at every three years new benchmarks reflecting the changes in technology (≈2,5 dynamic billion instructions).

The simulation methodology could be *execution driven* or *trace driven*. The first methodology is characterized by knowing, in every clock cycle, the content architectural resources (registers, memory location, reservation stations, functional units, etc). As outputs from this simulator are considered hit ratios from caches, processing rate, using degree of resources, or even traces of instructions executed. The main aim of trace driven simulation is to determine the optimal instance of architecture – *the processor* – which will be hardware implemented. In this sense, the instructions from traces generated by execution driven simulator are sequentially analyzed, helping to cache-processor's interface simulation, statically or dynamical branch prediction.

Figure 1 shows the simulation, comparison and establishing phases of an optimum computer architecture, starting from high level languages source of benchmarks until to architecture's hardware implementation (VHDL description, silicon implementation). The original source code (C, C++, Fortran 90, etc.) is first passed through a cross-compiler (e.g. GnuCC) that produces the correct format of assembler mnemonic code, together with assembler

directives and data allocation commands. The assembler code is relocatable, with branch targets and data references being expressed as label rather than actual addresses in memory. Optional, this code could be rearranged using an instruction scheduler that packs the instruction in independent groups. The resulting object code represents an input parameter for the execution driven simulator, highly parameterized, which generates a lot of interesting results and the trace file.

This paper tries to help anyone who wants to write an architecture software simulator, a trace driven in this case [1]. As the complexity of superscalar architecture increases, the underlying concepts of dynamic scheduling and speculative execution become increasingly difficult to explain without some form of visualization. So, the simulator was developed using Microsoft Visual C++, version 6.0, and could be run under Windows 9x or NT operating systems. The choice of authors relies on the fact that C++ language offers a strong support for object oriented programming: multiple inheritance, polymorphism, friend functions and classes. All these concepts create the premises for future development (extension) of actual simulation version. The implementation should be realized in so manner, that any change (adding) in hardware or software to be made with a minimum effort. From the user point of view it is very necessary a visual friendly interface, based on menus, dialog boxes, graphical images, although not every simulator provide some kind of interface (e.g. *Simplescalar Tool Set*). The simulator must be easy to use and the results must be efficiently interpreted and processed (eventually transferred to some utility application such Microsoft Graph, Excel, PowerPoint, Internet).

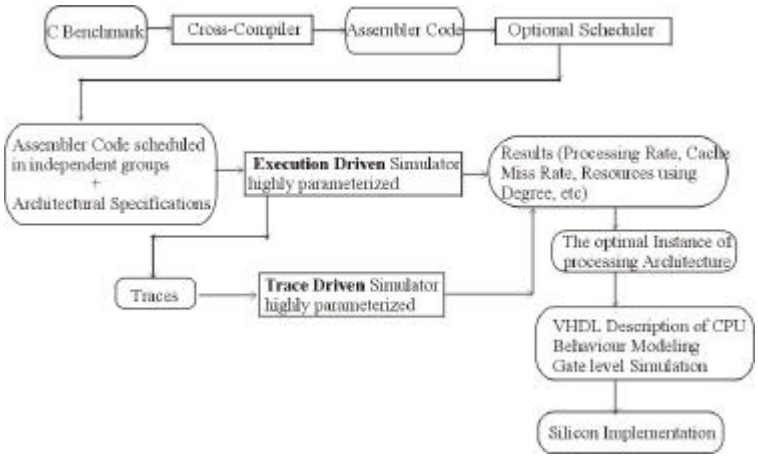

Figure 1. The simulation and establishing phases of a computer architecture

2. Software Principles of Implementation

Besides the instruments for debugging, editing and resource generating, Developer Studio environment from Visual C++ provides to programmers three kind of wizards used to simplify Windows programs development.

- *AppWizard* is used to create the basis structure of a Windows program.
- *ClassWizard* helps to define the classes inside a program created with AppWizard and to manipulate the controls or other resources from dialog boxes, etc.
- *OLEControlWizard* is used to create the base mainframe of an OLE Control.

The main stages in developing Visual C++ programs are:

- Creating the backbone of program using AppWizard.
- Creating the software resources required by program.
- Adding the classes and functions for treating the messages using ClassWizard.
- Creating and developing the functional kernel of program.
- Compiling and debugging the programs using the Visual C++ Debugger.

Starting the simulator execution, (e.g. *cache.exe* – simulator dedicated to cache memory integrated in a superscalar architecture) on the host computer screen appear a window main *menu.* The simulator (the program application) relies on simple menus nested or float, depending on operations executed on specific architecture elements.

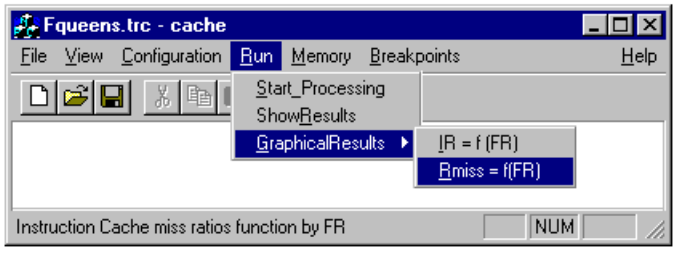

Figure 2. The simulator main menu

The *File* menu contains options like *OpenTraces* (for selecting and reading the benchmark), *ResetAll* (reset the architectural configuration), *CancelBenchmark* (close the active benchmark) and *Exit* (close the simulator and pass the control to operating systems). The *Configuration* menu includes *ExecutionUnits* branch (for setting the main input parameters except those related by cache), *Forwarding* (for enabling respective disabling of this mechanism), *Load_Config* respective *Store_Config* (necessary for loading or storing an existing configuration). The *Run* menu consists in the options: *Start_Processing*, *ShowResults* (exhibit the statistical results in edit boxes) and *GraphicalResults* (function by executed simulation). Choosing *Memory* menu the user should specify the type, size and architecture of caches. Also, it should be set the cache block size, penalty due to cache misses, writing cache strategy, using or not the Data Write Buffer. The *Breakpoint* menu contains the options *Set*, *Reset* and *ResetAll*, for establishing and removing partially respective totally of interrupting points. The *Help* option must contain explanation and detailed references about architecture, hardware and software resources, facilities offered by simulator, how it works, and statistical results.

Another software resource, frequently used in implementation of simulator is the *dialog box*. Their lifetime is very short and they are used to present information or to collect data from input devices. The dialog boxes have different forms, varying from simple *messageboxes* to very sophisticated dialog boxes that contains controls such as: *editboxes*, *buttons*, *listboxes* or *comboboxes*, *listviews*, *OLEControls* (*grids* and *graphics*), *progress bar* controls. A very important dialog box presented here illustrates the results under graphical form (figure 3). The main component of this dialog box is the ActiveX control (user interface element build in ActiveX technology) that helps for representing the graphical results of simulation without asking help from other applications such Microsoft Excel, MS Word Graph or MS Access. ActiveX represents a set of technologies that enables software components to interact with one another in a networked environment, regardless of the language in which they were created. ActiveX™ is built on the Component Object Model (COM). Additionally, on pushing the Save button it is created a text file containing the results simulated till then, under the following format:

	sort	tree	matrix	bubble	queens	tower	perm	puzzle	Hmin
FR=4	3.4333	6.0045	0.0506	0.0454	13.2642	0.0000	0.0267	7.6752	0.0000
FR=8	2.8733	8.1122	0.0572	0.0346	14.7214	0.0000	20.3828	7.7634	0.0000
FR=16	3.8167	9.3909	0.0630	0.0345	12.6638	0.0000	26.1302	6.9828	0.0000

The zero values suggested that respective benchmark wasn't simulated yet and implicit it wasn't computed the average performance value. Pushing the OK or Cancel button the box will be close and the results will be store in the used software structure. Selecting ResetAll option from File menu will graphic reset the ActiveX object.

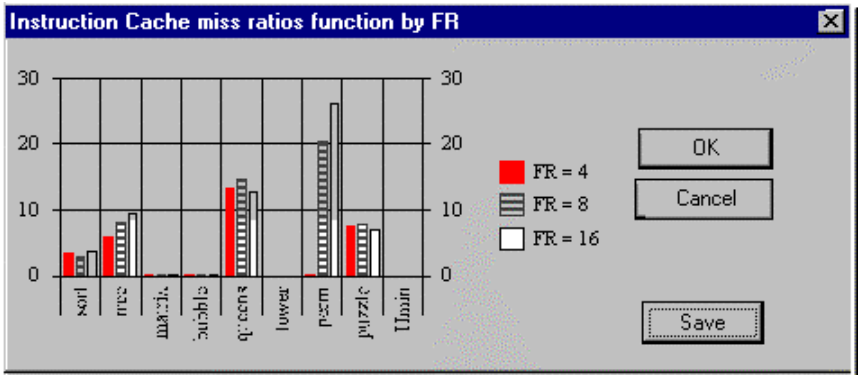

Figure 3. Graphical representation of results using ActiveX objects

From among other used software resources we mention:

- Very flexible button controls used to close dialog boxes, starting search process, asking for help.
- Edit control (singleline and multipleline) for introducing/showing data from/to I/O devices.
- Listboxes that allows the user to select from many options (simple or multiple selection). Multiple selection may be choosing when it wants to simulate simultaneous on more than one benchmark (multithreading).
- A "*progress bar control*" is a window that an application can use to indicate the progress of a lengthy operation. It consists of a rectangle that is gradually filled, from left to right, with the system highlight color as an operation progresses.
- The listview control displays items using one of four different views. The user can arrange items into columns with or without column headings as well as display accompanying icons and text.

3. The User Interface. The Functional Kernel of Simulator

High performance superscalar processors organizations divide naturally into an instruction fetch mechanism and an execution mechanism (Figure 4). These two mechanism are decoupled by an instruction issue buffer (queues, reservation stations, etc). Conceptually, the instruction fetch mechanism acts as a "*producer*" which

fetches, decodes, and places instructions into the buffer. The instruction execution engine is the "*consumer*" which removes instructions from the buffer and executes them. The branch instructions and their hardware predictors provide the feedback mechanism between the producer and consumer. So, in a missprediction case, the buffer must be emptied at least partial and the access address to the Instruction Cache must be modified accordingly with the target address of branch.

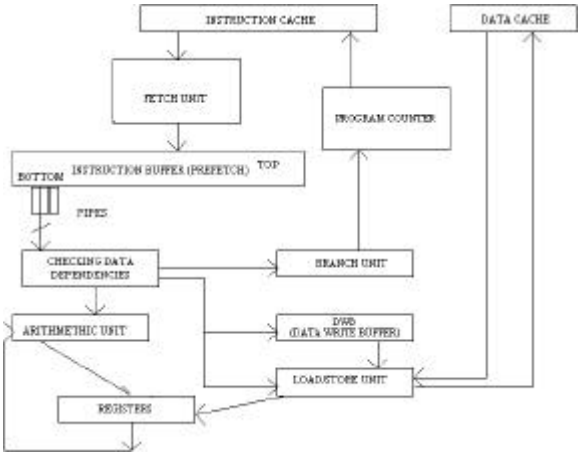

Figure 4. The Simulated Superscalar Architecture

The main parameters of the architecture, choose accordingly with the latest technology are [2]:

- FR – fetch rate – the instructions number that is fetched from Instruction Cache
- IBS – instruction buffer size
- IRmax – parallel issue capability – the maximum number of instructions that can be dispatched concurrently from the Instruction Buffer
- Instruction Latencies (measured in CPU cycles)
- Type and Number of Functional Units
- Type and Size of Cache Memories
- Using or not the *Data Write Buffer* (DWB). DWB is a small write buffer, which contains the virtual address and data that must be written in Data Cache, offering then multiple virtual writing ports. Using DWB it is eliminated the need of serializing Store instructions with afferent penalties and besides, through bypassing will be eliminated many "Load after Store" hazards.
- Cache writing strategy in Data Cache. There are two basic options when writing to the cache: *write back* and *write through*, both of them having advantages. I/O devices and multiprocessors are fickle: they want write back for processor caches to reduce the memory traffic and write through to keep the cache consistent with lower levels of the memory hierarchy.

The machine model should be "*fine-tuned*" to remove redundant or little hardware features and to investigate possible tradeoffs of performance against the functionality provided. The realized simulator must remove the bottlenecks that limit the processor performance and search for possible changes (architectural or optimization techniques) for improving it. Providing a highly parameterized model for every processor instance, the performance obtained by simulation will represent a quick feedback mechanism related to proposed changes. The simulator execution consists in the following sequential steps:

1. *Configuring the microarchitecture* with the input parameters (FR, IR, IBS, etc) and *selecting the benchmark* that will be simulate through the graphical interface described before.
2. *Initialization phase* (caches, Program Counter register, and clock counter).
3. *Starting the trace processing*: FR instructions are extracted from cache/memory and copied into the top of the Instruction Buffer, if there is room for them. IRmax instructions from the bottom of the Instruction Buffer are selected for in-order dispatch to the Functional Units. Successfully dispatched instructions are marked as "*squashed*" in the Instruction Buffer and contiguous squashed instructions are removed from the bottom of the buffer at the end of each cycle. The unsquashed instructions can only be dispatched at the same time if there are no data dependencies between members of the group that is formed. The clock counter is incremented accordingly with the maximum time consumed by each mechanism: fetch and issue. The simulator generates a large amount of statistical data about program profiles or the way in which the machine's resources have been utilized. Also, one of the outputs could be the performance gain obtained by implementing of different advance techniques (selective victim cache, dynamic instruction reuse, value prediction, trace cache, etc).

References:

[1] L. Vintan, A. Florea – *Microarchitectures for Processing the Information* (in Romanian), Technical Publishing House, Bucharest, 2001.

[2] R. Collins – *Exploiting Instruction Level Parallelism in a Superscalar Architecture*, PhD Thesis, University of Hertfordshire, October 1995.