EMSim: An Extensible Simulation Environment for Studying High Performance Microarchitectures

Daniel Ortiz-Arroyo[†], Ben Lee[†], and Chansu Yu[‡]

†{dortiz, benl}@ece.orst.edu
Electrical and Computer Engineering Department
Oregon State University
Corvallis OR, 97331, USA

‡{c.yu91@csuohio.edu}
Electrical and Computer Engineering Department
Cleveland State University
Cleveland OH, 44115, USA

Abstract

Modern microprocessors achieve high performance through the use of speculative execution and mechanisms to exploit instruction level parallelism. Performance evaluation of such architectures is generally made using detailed, cycle-by-cycle simulation. Since detailed simulation is slow, the design of recent simulators has been focused on developing fast simulation engines. However, these optimized simulators are difficult to modify or extend. In addition, intensive benchmarking is required to validate simulation performance results. This task consumes a significant amount of time even if very fast simulators are used.

This paper presents a novel simulation environment to study high performance microarchitectures. This environment consists of an extensible simulator for superscalar architectures and a group of utilities to perform benchmarking in parallel. The new simulator developed has features that are not found in other simulators reported in the literature. These features include extendibility, on-the-fly value passing, and distributed architecture.

Keywords: Microarchitecture, superscalar, simulation, object oriented, generic programming, parallel and distributed computing.

1. Introduction

Simulation of modern, high-performance microachitectures is carried out using trace or execution-driven simulators. Trace-driven simulation is fast because traces typically contain the opcodes as well as all memory addresses referenced when the trace is created, i.e., the simulator does not need to calculate these values [25]. However, trace-driven simulators are unable to emulate the speculative actions performed by modern microarchitectures [3][4]. In contrast, execution-driven simulation is generally slower than its trace-driven counterpart. The reason is that memory is accessed and instruction values are calculated and processed at each stage of the pipeline. This type of simulation accurately reproduces the dynamic behavior of modern microarchitectures, including speculative execution. However, accuracy is obtained at the expense of simulation speed.

The development of a detailed simulator for a superscalar architecture is a complex task [6]. For this reason, researchers usually modify an existing simulator to study advanced architecture issues. However, existing simulators are designed for speed but not to ease modifications. Moreover, the structural complexity of many fast simulators makes them difficult to understand. To address these issues, this paper presents a new simulation environment designed to study modern microarchitectures—the *Extensible Microarchitecture Simulator* (EMSim). EMSim's design incorporates some of the latest developments in object oriented software technology aimed to handle complexity and modifiability. The simulation environment includes a new generic superscalar processor simulator, which can be extended to simulate more complex architectures. EMSim has the following features not found in other simulators reported in the literature: Truly object-oriented design, distributed and multithreaded architecture, implementation based on generic programming, and *on-the-fly* value passing.

Once a simulator has been developed, results from the simulated architecture are evaluated using intensive benchmarking. Execution of benchmarks takes a significant amount of time even when a very fast simulator is employed; hundreds of millions of simulated clock cycles are required to validate simulation results. To perform benchmarking efficiently, EMSim also includes tools to carry out performance evaluation in a *Beowulf* cluster of computers.

This paper is organized as follows. Section 2 describes the related work on microarchitecture simulators for superscalar architectures. Next, the main component of the simulation environment, i.e., the simulator of a superscalar architecture, is described in detail. In section 4, the tools used to perform benchmarking on a cluster of computers are described. Finally, section 5 provides a brief conclusion and future work.

2. Related Work

There exist a number of simulation tools that contain detailed models of today's high performance microprocessors. *SimpleScalar* (SS) tool suite [6] is a popular simulation platform that provides several classes of simulators of varying accuracy/speed. Among these, *sim-outorder* simulates a superscalar microarchitecture and is the most complex simulator of the tool suite; it is a hybrid of functional and trace simulators. Traces are generated on-the-fly by the front-end simulation engine. This engine issues and executes instructions in-order, modifying the values of registers and memory. In the back-end, these traces are used to emulate an out-of-order processor, without modifying registers or memory. Sim-outorder handles system calls by passing them to the host operating system. The host OS

executes the system calls and passes the results back to sim-outorder. SS tool suite is being widely used in computer architecture research. SS is written in C, executes only user-level application programs, and has been ported to many different platforms. A large percentage of the research published in major conferences and journals is done using SS. However, sim-outorder is not easy to modify due to its structure and complexity.

In contrast to the SS approach, SimOS simulates all the hardware in a computer system, including I/O devices such as hard disks and network interfaces [21]. SimOS simulates all the hardware components in sufficient detail to boot and execute a complete OS. Using this simulator, it is possible to study the effects of more realistic workloads on the performance of a complete computer system. SimOS is written in C, models the MIPS R4000, R10000 and Digital Alpha processor families and executes IRIX and Digital Unix OS. SimOS comes with an in-order processor simulator but an out-of-order version (MXS) [11] is also available.

PSim [7] is a simulator for the PowerPC architecture. PSim implements the three levels of the PowerPC *instruction set architecture* (ISA): User, virtual, and operating environments. In the user mode, PSim can run static programs compiled for any of the following operating systems: NetBSD, Solaris or Linux. This simulator comes integrated with the gdb debugger.

Other superscalar processor simulators were designed as teaching tools. Examples of this type of simulator are: SuperDLX [17], and SATSim [27]. There are also simulators that are variations of SS, such as SIMCA [13], which has multithreading capabilities. This special purpose simulator requires support from the compiler to generate threads. In addition, some simulators run only on specific platforms or require special compilers such as MIPS [8] or SMTSim [26]. All these simulators are execution-driven.

In contrast, there are simulators that are both, event-driven and execution-driven, e.g., RSIM [20]. RSIM simulates an out-of-order processor similar to MIPS R10000 and is partially written in C and C++. RSIM is also capable of simulating a multiprocessor system using event-driven simulation.

Another hybrid simulator is fMW [3], which is a descendent of the trace simulator VMW [9]. This simulator contains a trace engine called MW that directs the order of instruction execution of PSim. PSim calculates results and sends the data back to MW, which calculates IPC and processor utilization.

Most aforementioned simulators are written in C for fast execution. However, since the main goal of these simulators is to provide correct functionality and speed, their code structure is complex. Therefore, modifications to such simulators are difficult to perform. Moreover, the use of centralized data structures in these simulators increases the risk that modifications in one section of the simulation code could cause unintended side effects in other parts of the code. Side effects are undesirable, since their presence (1) complicates the total understanding of a simulator's actions, (2) makes code reutilization difficult, and (3) causes bugs that are difficult to detect.

Due to its modular design centered on class hierarchies, EMSim structure is easy to understand. In addition, EMsim's design employs generic containers and *virtual* functions, which allow minimizing the impact of modifications. Moreover, the objects defined allow new components to reuse the existing functionality included in the simulator. Thus, the simulator can be tailored to new architectures by refining parts of its code. Next section describes in detail EMSim's internal structure and design.

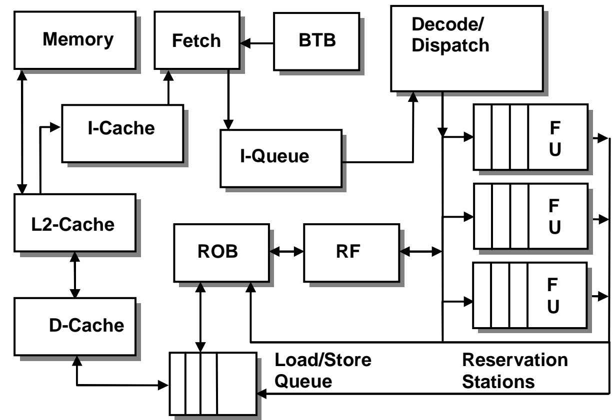

Figure 1. EMSim's superscalar processor model.

3. EMSim Superscalar Simulator

The specific superscalar architecture that EMSim simulates is shown in Figure 1. It consists of six main stages: Fetch, Decode/ Dispatch, Issue, Execute, Write-Back, and Commit. In the Fetch stage, instructions are fetched from memory and placed in the *instruction queue* (I-Queue). In this stage, branch prediction mechanisms are employed to avoid stalling the fetch unit. Using branch prediction, the fetch unit can continue fetching at the most probable path of execution, *speculatively*. Later, if the speculation turns out to be wrong all mispredicted instructions are flushed from the pipeline and fetching resumes at the correct path of execution [12] [14]. A special mechanism in the Fetch unit called the Branch Target Buffer (BTB) provides the prediction value (i.e., *taken*, *not-taken*) and the target address of a branch.

During Decode/Dispatch stage, instructions are decoded and renaming mechanisms resolve *false* dependencies (WAW and WAR hazards) [12], which allow independent instructions to be dispatched for execution. On the other hand, instructions with *true* (RAW) dependencies are placed in the Reservation Stations (RSs), i.e., *instruction window*, where they remain until their dependencies are resolved. Once this occurs, the instructions are issued to the Functional Units (FU) for execution. Superscalar processors execute instructions *out-of-order* to exploit instruction level parallelism (ILP). However, instructions are committed *in-order* to preserve the semantic content of a program. Control of instruction retiring and dependency handling is performed by the Reorder Buffer (ROB) using a special tagging mechanism that eliminates WAW and WAR hazards [14] [23].

Once instructions are executed their results are written back to RSs, during the Write-Back stage, enabling dependent instructions that were waiting for those values to become ready for execution. Finally, during the Commit stage, instructions are retired from the ROB in-order and their results committed to the Register File (RF).

Speculation is actively researched to predict data values from registers or memory [23]. In other approaches, speculation is used to dynamically generate threads from a sequential flow of control [15]. Furthermore, some recent architectures support the overlapped execution of multiple, independent threads using Simultaneous Multithreading (SMT) [26]. Therefore, it is obvious that to simulate these complex architectures, flexible simulation tools are required.

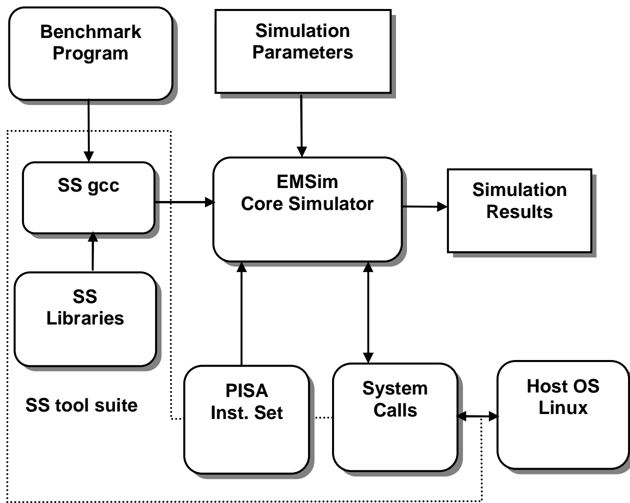

Figure 2. EMSim's programming environment.

EMSim was designed using *object oriented* (OO) techniques. The advantages of an OO approach to software design in general are well documented [5]. They include many well-accepted design goals of quality program development, such as modularity, modifiability, and maintainability [16]. Moreover, designs centered on objects are especially suited for simulation.

Simulation speed is obviously an important factor in a simulator. However, the features that provide the OO approach to software design are equally or perhaps more important in a simulator. Languages such as C++ provide useful OO mechanisms, such as *inheritance*, *polymorphism*, and *templates*. Templates support the design of software using *generic programming* [2] techniques. In generic programming, software components are created so that they can be easily reused in a wide variety of situations. The data structures and algorithms in the Standard Template Library (STL) [2] are examples of the application of generic programming. In this library, software components such as queues, sets, lists, etc., are able to handle different types of objects employing different algorithms.

EMSim provides modularity, code reutilization, and extendibility through the use of classes, inheritance and generic programming. Modifications to EMSim are easily integrated since these features are available to a developer. To obtain fast execution speed, EMSim was developed in C++. In addition, the implementation of EMSim was carried out employing STL's generic containers and iterators. Moreover, the *interfaces* defined using virtual functions allow subclasses to specialize methods with their particular implementation.

Figure 2 shows EMSim's programming environment. As this figure illustrates, the simulator in its current version is compatible with the compiler, linker, assembler, and libraries of the SS tool suite. As a result, EMSim shares with SS the way in which data, stack, and code areas are mapped into memory. Parts of the macros that define the implementation of the instruction set and the system calls of SS were modified to make them compatible with EMSim. EMSim's simulation parameters can be configured from the command line or from a text file. The configuration parameters of EMSim include: size and associativity of cache memories and BTB; size of the instruction queue, ROB and reservation stations; and number and type of functional units.

EMSim superscalar simulator is execution-driven. The simulator is able to operate in different modes of execution, which are (a) fast, in-order simulation and (b) detailed wide-issue, out-of-order simulation. The fast simulation mode allows the user to quickly place the simulator in a particular section of the benchmark code, skipping uninteresting parts like initialization. During fast mode simulation, instructions are read directly from memory and executed in sequence. Conversely, in the detailed simulation mode, all the memory hierarchy and the pipeline stages of the simulator are exercised. In this mode, EMSim loads a binary program into its internal memory and then simulates in detail, cycle-by-cycle, all the processing performed by the pipeline. During instruction processing, register values are calculated and passed from producer instructions to consumer instructions or memory on-the-fly. EMSim's on-the-fly value passing closely emulates the behavior of real superscalar processors, and is also used as a means for checking the correct operation of the tagging and out-of-order execution mechanisms inside the simulator. In this way, EMSim ensures correct manipulation of instruction values during speculative execution.

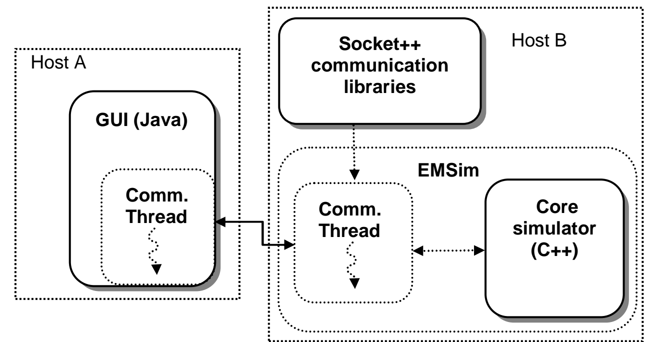

Figure 3. EMSim's distributed architecture.

Functional validation of EMSim was performed in two ways. First, the contents of EMSim's memory and registers were compared on a cycle-by-cycle basis with the corresponding values obtained by sim-outorder during the execution of the same benchmark. In addition, special benchmarks, which perform intensive mathematical calculations, were executed on a real machine. The results obtained by these benchmarks were compared with the results obtained from EMSim. In both cases, EMSim obtained the same results as sim-outorder and the real machine.

EMSim has a distributed architecture that consists of a graphical user interface (GUI), a core simulation engine, and communication facilities. This is in contrast with existing simulators that consists of only a single executable with a simple text mode user interface. The Java language provides, through the *Swing* library, abundant graphical elements to build complex user interfaces that are portable across different platforms [10]. To make use of these capabilities and without sacrificing speed of execution, EMSim was designed using Java's Swing in the user interface and C++ in the core simulator. This architecture allows decoupling the GUI from the simulator, which is a useful feature during testing or when benchmarking is performed. Figure 3 shows EMSim's distributed architecture.

EMSim's user interface handles user events and creates a special thread for communication with the main simulator through TCP/IP sockets. On the other hand, the core simulator creates a Posix thread (*pthreads)* to execute the communication routines. The communication thread in the GUI receives commands from the user to control the simulation execution. A simple protocol was designed to transfer data between the simulator and the GUI. The communication library socket++ [24] is employed to receive/send the C++ input/output streams from/to

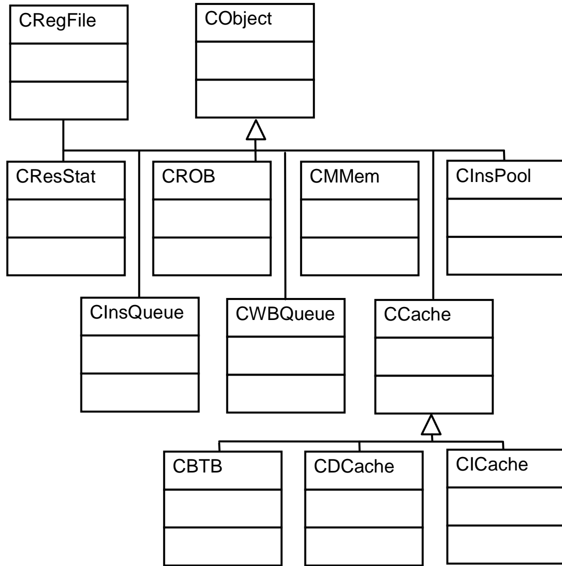

Figure 4. UML diagram of EMSim's main classes.

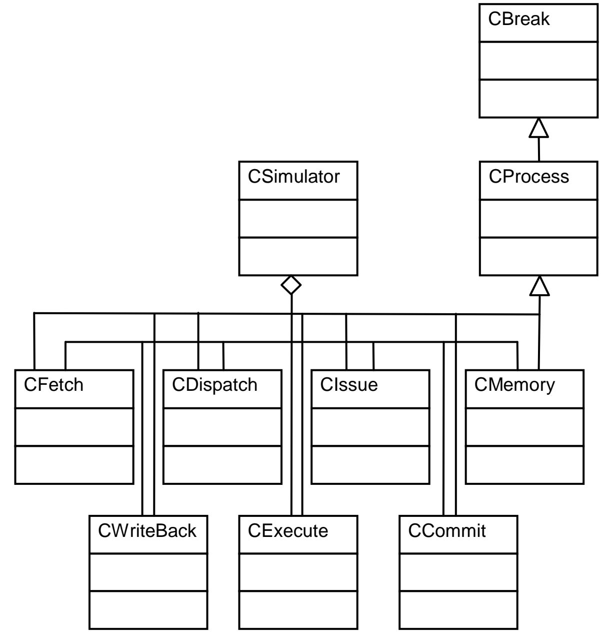

Figure 5. UML diagram of EMSim's pipeline classes.

the simulator. EMSim's distributed architecture makes it feasible to use the simulator with the GUI on a single computer or remotely over the network. EMSim is also able to run on a single computer without the GUI in text mode. In the latter case, the output streams generated by the simulator are re-directed to the standard output instead of through the TCP/IP sockets.

EMSim's design is organized in a hierarchy of classes. A partial UML diagram of the main classes in EMSim is shown in Figure 4. The base class CObject provides common methods and variables to all other classes derived from it. Instruction cache (CICache), data cache (CDCache), and BTB (CBTB) classes are specializations of the base class CCache to store instructions, data, and branch information, respectively. Classes are also defined to keep the state and provide the actions performed by the register file (CRegFile), main memory (CMMem), reservation stations (CResStat), instruction pool (CInsPool) and writeback queue (CWBqueue).

EMSim pipeline was modeled using the UML class hierarchy diagram shown in Figure 5. The class CSimulator contains the classes used to model the pipeline stages and the main simulation loop. As is illustrated in Figure 5, classes were defined to emulate a processor pipeline consisting of fetch (CFetch), decode/dispatch (CDecode), issue (CIssue), execute (CExecute), write-back (CWriteBack), and commit (CCommit) stages. In addition, memory instructions are processed during the memory stage (represented by CMemory). The base class CProcess provides the features that are common to all derived classes representing the pipeline stages (e.g., bandwidth). Utility classes (not shown) were also designed to handle statistics, exceptions, memory data, clock, timers, etc.

The design of EMSim also provides support for debugging. The class CBreak in Figure 5 allows breakpoint conditions to be declared. This class also contains methods to dump the state of a pipeline stage when a breakpoint condition occurs during simulation. The information dumped includes the state and contents of the queues handled by the stage. The dump method can be overridden by a new class derived from CBreak to print any other information required by the user.

In its design, EMSim utilizes different STL generic containers such as sets, lists and queues. Although the design of STL was optimized for fast execution, this library adds an overhead to the total simulation time. Hence, to minimize this overhead, STL's generic containers were employed only in those parts of the simulator that do not negatively impact EMSim's performance. Therefore, containers of objects that are used very frequently were implemented using templates and arrays instead of STL generic containers.

EMSim's main loop processes all the pipeline stages at each simulation cycle in the following way:

```
obj_list_iterator p;
for(;;;){
   for(p=s_RunList.begin();
        p!= s_RunList.end();
        p++)
     (*p)->Run();
   clock.Tick();
}
```

As this code segment illustrates there are no specific references to particular pipeline stages in the main loop. Pipeline objects are stored in the STL generic container `s_RunList` during initialization, and then, accessed through the iterator `p`. This approach reduces the amount of changes and at same time, simplifies modification to EMSim. As an example, to add a new pipeline stage into EMSim, a new class derived from CProcess is created. This class will include the implementation of the *virtual method* `Run` shown in the segment of code above. Virtual methods are the interface that is implemented in different ways at each pipeline stage. Then, an instance of the class would be stored in `s_RunList`. Pipeline objects are stored using the `push_back()` method of the container `s_RunList`. The iterator in the main loop of EMSim will automatically process the new pipeline stage by calling its `Run` method. Using this generic programming approach, the main loop of the simulator remains unchanged regardless of how many pipeline stages are added (changes occur mainly in the new class code).

In EMSim, pipeline stages communicate through instruction queues. Hence, after being processed, instructions must be placed in the appropriate output queue so that the next pipeline stage can access those instructions. Thus, no global structures are accessed during this process. In contrast, adding a new pipe-

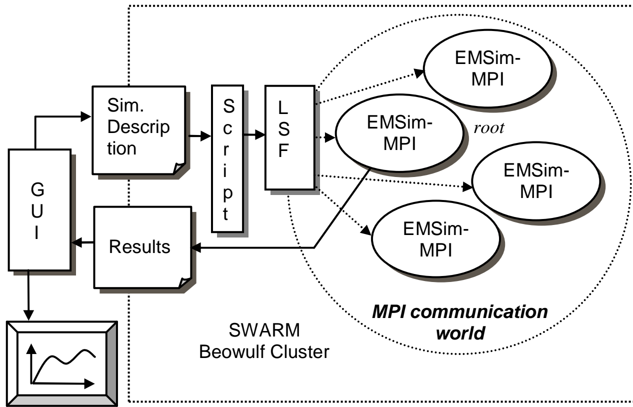

Figure 6. Benchmarking on a cluster of computers.

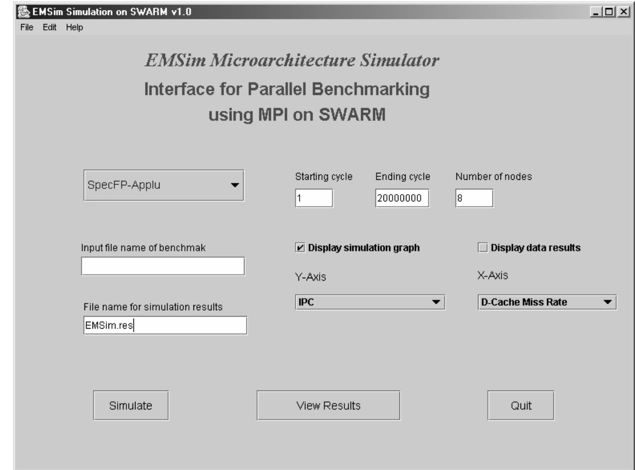

Figure 7. GUI to configure benchmarking parameters.

line stage in SS's sim-outorder requires changing the main simulation routine. In addition, the new procedure must update, in an appropriate way, the global structures and variables (e.g., queues and flags) that keep track of the processor's state in the RUU unit [6]. However, since the RUU unit is a centralized structure, this process must be performed very carefully to avoid causing unintended effects in other parts of the simulator.

The OO structure of EMSim facilitates extending its capabilities to simulate other types of architectures and allows visualization of the changes that will be required to perform those extensions. As an example, extending EMSim to simulate a SMT processor entails creating additional instances for the register file (or processor's *context*), instruction queue, ROB, and Load/Store Queue (LSQ). In addition, using the generic programming approach, such instances will be stored in generic containers that the appropriate pipeline stage would access in order to process instructions corresponding to different contexts. Other changes involve extending the cache memory to support multiple ports, overriding the fetch method that access the instruction cache memory to support a new fetching policy, overriding the method that is used to dispatch instructions, etc.

4. Efficient Benchmarking

Performance evaluation of new microarchitectures requires intensive benchmarking. During benchmarking, overall simulation time can be substantially reduced if a group of networked computers is employed. Each computer in the network is programmed to execute a simulation using different parameters. However, in this environment it is difficult to achieve good load balancing among computing nodes, especially if nodes are not dedicated exclusively to simulation, i.e., nodes may finish at unpredictable times. Moreover, there is no way to synchronize simulations without explicit communication support. Synchronization is required when all the nodes are executing the same benchmark using slightly different parameters. Once all the nodes finish a simulation, they synchronize to calculate the group of parameters that will be used for the next simulation run.

A computing cluster, such as the *SWARM Beowulf cluster* [18] can ameliorate these hurdles. In SWARM, load balancing is obtained through specialized software, such as *Load Sharing Facility* (*LSF*) [18], which analyzes the load assigned to each node, i.e., nodes with lesser load are assigned more computation.

Executing synchronized simulations in a cluster of computers requires communication among simulation nodes. The *Message Passing Interface* (MPI) [19] library provides the communication primitives required for this task. Using MPI, special communication worlds are created for each simulation [19]. Simulation nodes that participate in a communication world are able to perform a simulation cycle coordinately. Using MPI it is even possible to define several communication worlds where participating nodes work on different benchmarking tasks.

Figure 6 shows EMSim parallel benchmarking environment. This environment consists of (a) a GUI application written in Java, (b) a collection of Perl and shell scripts, and (c) EMSim simulator modified with MPI calls. EMSim employs MPI functions to send, receive, and synchronize with other simulation nodes in the cluster of computers.

Figure 7 shows the GUI that allows configuring simulation parameters for parallel execution of benchmarks on SWARM. Some of the information provided by the user includes measurement variables such as IPC, cache miss rate, branch prediction accuracy, etc. The user also enters the name of the benchmark program to execute, the number of simulation nodes, the number of simulated clock cycles to execute, and the format of the output results. Once all these information is entered, the GUI creates a text file automatically. Then, a script written in Perl, reads the simulation information and calls LSF, passing it the required number of simulation nodes. Then, LSF sends the simulation processes to SWARM, and finally the EMSim simulation nodes are initialized by the MPI system to work in parallel. A special root node coordinates all other nodes in the cluster to initiate a new simulation run.

During initialization, each simulation node calculates its own *identification* number. This number is used by a node to obtain its particular simulation parameters. Then, through the network file system (NFS) installed on SWARM, it obtains the object file corresponding to the benchmark program that the node will execute. Afterwards, it executes a simulation run independently. When a node finishes a simulation run, it synchronizes with all other nodes in its communication world sending its results back to the root node. Once this occurs, the node is free to start a new simulation cycle. Finally, when all nodes terminate their simulation workload, each one sends its final simulation results to the root node. With these results, the root node creates a single file with the complete simulation results. At this point, the LSF system automatically sends an email to the user indicating that a simulation cycle has finished. The user can then execute the GUI to read the file containing simulation results and optionally generate a graph with those results.

5. Conclusions

A new simulation environment designed to study modern microarchitectures was presented in this paper. This environment contains a generic superscalar simulator model that provides the basic elements to simulate other more complex architectures.

Traditionally, the development of simulators for modern microarchitectures has focused on producing fast simulators. Even though simulation speed is important, other features such as modularity, reusability, ease of debugging, and modifiability have been neglected or only partially addressed in existing simulators. We believe that such features are essential to ease the development of advanced simulators for computer architecture research. EMSim's design provides modularity and reusability through the use of class hierarchies. Moreover, the impact of modifications is minimized through the use of generic programming. In addition, the structure of the simulator is easy to understand since there is a one-to-one correspondence between elements in the processor and the classes defined. EMSim provides all these capabilities without sacrificing execution speed since it is written in a fast OO language, e.g., C++. Finally, EMSim's distributed architecture allows decoupling the GUI during benchmarking.

The simulation environment presented in this paper also provides the tools required to facilitate the coordinated execution of benchmarking in a cluster of computers. Moreover, software utilities allow automatic data gathering and presentation of simulation results.

EMSim and the tools presented in this paper are currently in its alpha version. At this time, the GUI and debugging facilities of EMSim provide minimum functionality and are still under development.

Our future work includes porting of EMSim to a different ISA, replacing in this way the SS tools used during development. In addition, later versions will include more elaborated debugging and visualization facilities. Specifically for debugging, we plan to implement *checkpoints* [21], which will enable EMSim to save the entire state of the simulator to disk. Using this feature, the simulator can restore the processor's state at a checkpoint, and continue execution from there, saving simulation time. Finally, a *Dynamic Simultaneous Multithreaded processor* (DSMT) simulator is also planed for development by extending EMSim.

Acknowledgments

The authors would like to thank Mark Daley for his contribution during the initial development phase of this project.

6. References

[1] K. Arnold and J. Gosling, *The Java Programming Language*, Addison-Wesley, Reading Massachusetts, May 1996.

[2] M. Austern, *Generic Programming and the STL*, Addison-Wesley, 1998. ISBN 0-201-30956-4.

[3] C. Bechem *et al.*, "An Integrated Functional Performance Simulator," *IEEE Micro*, May-June 1999.

[4] B. A. Bryan *et al.*, "Can Trace-Driven Simulators Accurately Predict Superscalar Performance?," *Proc. of International Conference on Computer Design*, Oct. 1996, pp. 478-485.

[5] T. Budd, *An Introduction to Object Oriented Programming*, (2nd Edition), Addison Wesley, Reading Massachusetts, 1997. ISBN 0-201-82419-1.

[6] D. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0," *Computer Architecture News,* Vol. 25, No. 3, June 1997, pp. 13-25. http://www.simplescalar.com

[7] A. Cagney, PSim PowerPC simulator. http://sources.redhat.com/psim

[8] Clemson University. MIPS Superscalar Simulator (for Sparc-Solaris big-endian platform). http://www.eng.clemson.edu/~ilp/sim.html.

[9] T. A. Diep and J. P. Shen, "VMW: A Visualization Based Microarchitecture Workbench," *IEEE Computer*, 28(12), December 1995, pp. 57-64.

[10] R. Eckstein, M. Loy, and D. Wood, *Java Swing*, O'Reilly Ed., 1998.

[11] M. J Flynn, P. Hung and A. Peymandoust, "Using simple tools to evaluate complex architectural trade-offs," *IEEE Micro*, July-August 2000.

[12] J. Hennessy, and D. Patterson*, Computer Architecture-A Quantitative Approach*, 2nd Edition, Morgan Kaufmann Publishers Inc., 1996.

[13] J. Huan, The Simulator for Multithreaded Computer Architecture (SIMCA). http://www.mount.cs.umn.edu/Research/Agassiz/simca.htm

[14] M. Johnson, *Superscalar Microprocessor Design*, Prentice Hall, Englewood Cliffs, NJ., 1990.

[15] P. Marcuello, J. Tubella, and A. González, "Value Prediction for Speculative Multithreaded Architectures," *Proc. of the 32nd. Ann. Int. Symposium on Microarchitecture (MICRO-32),* Haifa (Israel), November 16-18 1999.

[16] H. Mili, F. Mili, and A. Mili, "Reusing software: Issues and research directions," *IEEE Transactions on Software Engineering, vol. 21*, June 1995.

[17] Cecile Moura, "SuperDLX a generic Superscalar Simulator," *ACAPS technical memo*. McGill University, School of Computer Science. April 13, 1993.

[18] Oregon State University, Computer Science Department. SWARM cluster. http://www.cs.orst.edu/swarm

[19] P. Pacheco, *Parallel Programming with MPI*, Morgan-Kauffman Publishers Inc., 1996.

[20] V. S. Pai, R. Parthasarathy, and S. V. Adve., "RSIM: An execution-driven simulator for ILP-based shared-memory multiprocessors and uniprocessors," *IEEE TCCA Newsletter*. October 1997. http://www-ece.rice.edu/~rsim

[21] M. Rosemblum *et al.,* "Using the SimOS Machine Simulator to Study Complex Computer Systems," *ACM TOMACS Special Issue on Computer Simulation*, 1997.

[22] E. G. Sirer, "Measuring Limits of fine grain parallelism," Undergraduate thesis, Princeton University, 1993. http://www.cs.washington.edu/homes/egs/mipsi/mipsi.html

[23] J. E. Smith, and G. S. Sohi, "The Microarchitecture of Superscalar Processors," *Proceedings of the IEEE,* December 1995.

[24] Socket++ source code and user manual. http://www.cs.utexas.edu/users/lavender/courses/socket++/

[25] A. Srivastava and A. Eustace, "ATOM: A system for building customized program analysis tools," *Proc. of the 1994 Conf. on Programming Languages Design and Implementation*, 1994.

[26] D. M. Tullsen, "Simulation and modeling of a simultaneous multithreading processor," *Computer Measurement Group Conference*, December 1996.

[27] M. Wolf, and L. Willis, "SATSim a Superscalar Architecture Trace Simulator Using Interactive Animation," *Workshop on Computer Architecture Education*, June 2000.