NetSim: An Object-Oriented Architectural Simulator Suite

Abstract

NetSim is an object-oriented based architectural simulator suite written in C# and uses Microsoft's .NET Framework. NetSim consists of several libraries that contain various architectural modules that can be combined to form many different computer architectures. The approach to NetSim was to concentrate on accuracy and flexibility for any given architecture at the cost of simulation speed. This approach ultimately leads to quicker development time that yields more accurate results than past architectural simulators. Using NetSim, a two-person team was able to develop, debug, and validate a PowerPC-like superscalar processor running SPEC95 and SPEC2000 benchmarks in approximately eight weeks.

Keywords: Microarchitecture, cycle-accurate simulator, superscalar architecture, object-oriented.

1. Introduction

Cycle-accurate, execution-based architectural simulators provide a cheap and reliable way to prototype and test the performance capabilities of new architectures. Researchers in both industry and academia rely on these powerful tools to estimate the viability of a new architecture before the design is even brought down to the silicon level. Evaluating a proposed architecture with an accurate simulator might take several weeks, but creating the simulator can easily take several months.

This paper explores NetSim, an architectural simulator suite written in the object-oriented programming language, C#, and utilizes Microsoft's .NET Framework to increase portability and flexibility. NetSim has been designed to drastically decrease the time necessary to develop new simulators, while maintaining and, in some cases, improving the architectural accuracy of the simulator. NetSim removes the usage of global variables and cryptic C code by relying on an object-oriented model. Architectural components such as registers and functional units are implemented as individual classes, or objects, each containing their own private data and operators. This allows for quicker development time since designers can now think of the simulator as an actual hardware implementation and not as a software estimation.

There are many architectural simulators available, most of which can be grouped into three categories, (1) trace simulators, (2) functional simulators, and (3) cycle-accurate simulators. Trace simulators are useful for characterizing performances of Instruction Set Architectures (ISAs) by using pregenerated data and instruction traces. Functional simulators execute a program without considering the details of the system's architecture, and are useful for creating emulators. A cycle-accurate simulator attempts to mimic the exact behavior of an architecture by modeling the characteristics of instruction- and data-flow through a system with regard to simulation execution cycles. NetSim fits into the third category as a cycle-accurate simulator suite.

One of the most popular cycle-accurate simulators is SimpleScalar [1]. SimpleScalar is C based and relies on global variables and a cryptic coding style to boost the speed and performance of the simulator. This lends to a simulator suite that is very difficult and time consuming to modify in order to create a new simulator or gather different statistics.

SimCore [2] is another simulator suite that, like NetSim, uses an object-oriented approach. But, SimCore is solely a functional simulator and is not intended to perform cycle-accurate simulations.

NetSim, on the other hand, is a simulator suite that can be used to quickly implement a wide array of simulators. Currently, NetSim has been used to create a fast, functional simulator, FastSim, a fully cycle-accurate superscalar simulator, SuperSim, and a fully cycle-accurate multithreaded simulator, DSMTSim. The PISA instruction set [1] was chosen as the initial

target of NetSim due to the availability of the PISA compiler provided by SimpleScalar.

The rest of the paper is organized as follows. Section 2 provides a detailed overview of NetSim and how NetSim is structured. Section 3 discusses the major benefits and the pitfalls involved with using NetSim. Section 4 briefly describes SuperSim, a superscalar simulator created from NetSim. A performance comparison between SuperSim and SimpleScalar's sim-outorder is discussed in Section 5. Finally, Section 6 concludes the paper and discusses what the future holds for NetSim.

2. NetSim

NetSim is comprised of three main libraries that contain the building blocks to create a variety of simulators. The libraries are designed as a tier structure, where one library is an expansion of the previous. The three libraries are NetSimBase, NetSimHP, and NetSimMT that contain the base components, high performance components, and multithreading components, respectively.

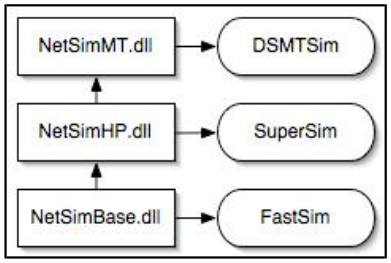

Figure 1: Relationship between the NetSim libraries and the simulators.

Figure 1 illustrates the relationship between the libraries and the actual simulators. The figure shows that the libraries rely on other libraries. For example, DSMTSim will need the NetSimMT library. Since the NetSimMT library needs both the NetSimHP and NetSimBase libraries, DSMTSim will also need those libraries in order to operate correctly.

2.1. NetSimBase

The base library for NetSim contains the core architectural modules that almost any simulator will need. Figure 2 illustrates the main modules and interconnections for NetSimBase. These main modules include the Instruction Database, Loader, Syscall, Statistics Database, Register File, Memory, Resource, Branch Predictor, and Cache modules. With the exception of a few of the modules, all of these basic modules are needed to successfully simulate a microarchitecture. A few of these modules will be described below.

A basic, functional simulator that executes one instruction at a time has been created as the self-sufficient module, FastMod. The purpose of FastMod is to be integrated within more advanced simulators as a way to fast forward or skip instructions within a program while maintaining the state of the simulator. This shortens the overall execution time by skipping the non-essential setup instructions, allowing most of the simulation time to be spent analyzing the critical components of a particular benchmark.

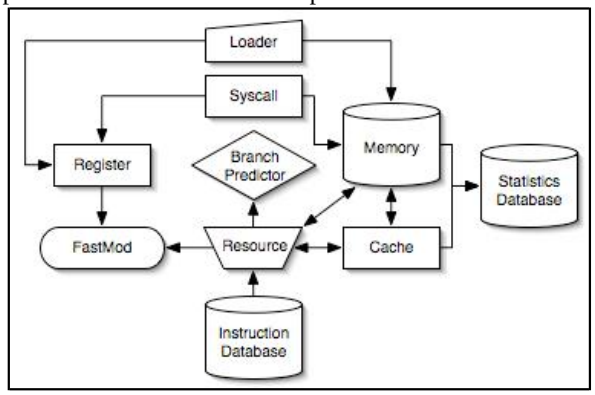

Figure 2: The modules of the NetSimBase Library.

The Instruction Database is simply a code file that contains each instruction as a child of the Instruction object. Upon initialization, the database uses reflection to instantiate every child class of the Instruction class and places the new instance object into a hash table. A binary instruction is then passed to the Instruction Database, which then compares the binary instruction identifier against the hash table and returns a pointer to the instruction object. Moreover, the instruction object contains the code necessary to execute itself. A new ISA may be implemented by reflecting on a different code file during the instantiation of the Instruction Database. The details of the retargetability are discussed in more detail in Subsection 3.1.

Figure 3: Creating a memory hierarchy.

NetSimBase contains Memory and Cache modules that may be used to simulate a multi-level memory hierarchy with cycle–accurate delays. Figure 3 demonstrates how one such memory hierarchy can be created. The Memory and Cache modules all inherit the same IMemoryBus interface, which allows for very complex memory hierarchies to be modeled independent of the rest of the architecture. Multiple simulation runs can then be used to compare the performance of a

basic architecture with any number of different memory hierarchies.

2.2. NetSimHP

NetSimHP contains the high-performance modules, or rather, those modules that are aware of simulated clock cycles. The NetSimHP modules can be used to create more advanced simulators such as a superscalar simulator or a five-stage pipeline simulator. Unlike other simulator suites that are available today, NetSim uses state machine model to evaluate the cycles and gives a more accurate representation of how the actual hardware will perform. As such, each component inherits the ISimComponent interface, which contains two methods, Update and Tick. The Update method is used to calculate all of the next state data and to communicate with any external modules, while the Tick method is used to transition the next state data to present state data and to communicate with internal objects. The state machine model allows developers to concentrate more on the interaction between the simulator components, and not the order in which the components are executed. Figure 4 shows the necessary code needed to execute a simulator clock cycle.

```
ISimComponent[] sim = Init();
while( true ) {
  /* Execute a simulator clock cycle */
  foreach( ISimComponent com in sim )
    com.Update();
  foreach( ISimComponent com in sim )
    com.Tick();
}
```

Figure 4: C# code needed to simulate the clock cycles.

Some of the modules in the NetSimHP library are wrapper classes around modules in NetSimBase that allow them to be cycle accurate. These wrapper modules include the Branch Predictor, Register File, and Functional Unit modules, which inherit the Branch Predictor, Register, and Resource modules from NetSimBase, respectively. Other modules are more specialized for superscalar processors such as the Fetch Unit, Reservation Station, Dispatch Unit, Reorder Buffer, and Common Data Bus. The modules and the basic interconnections within NetSimHP are illustrated in Figure 5.

Currently, these modules are used to create a superscalar processor simulator, SuperSim, which is equivalent to SimpleScalar's sim-outorder simulator. Section 4 will describe SuperSim in more detail, and Section 5 will compare some of the output statistics and timing result between SuperSim and sim-outorder.

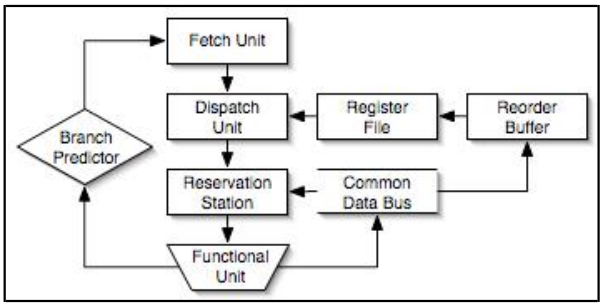

Figure 5: The Modules of the NetSimHP Library.

2.3. NetSimMT

The NetSimMT library contains components that are configured for a multithreaded environment. This is handled by including an additional field in the instruction object that identifies which thread the instruction belongs to. Many of the modules within NetSimMT are child wrapper classes for modules within NetSimHP. These classes can handle the extra thread ID field within the instruction, making them viable component for a multithreaded environment. The modules for NetSimMT and their interconnections are illustrated in Figure 6.

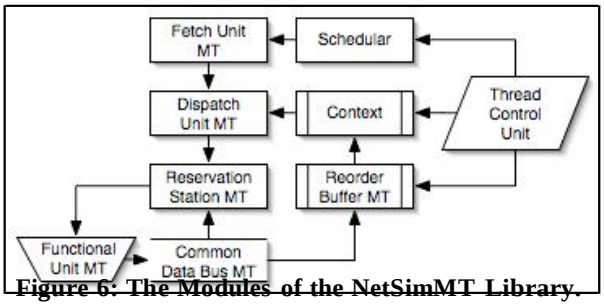

Figure 6: The Modules of the NetSimMT Library.

In addition to the multithreaded superscalar components, other components are available that can monitor, control, and create threads. These new components include the Loop Detection Unit, Thread Control and Initialization Unit, Contexts, and a new Dispatch Unit. The modules in this library are currently being used to create a simulator for a new multithreaded architecture, called Dynamic Simultaneous Multithreading (DSMT), and thus aptly titled DSMTSim. Discussion of DSMT is beyond the scope of this paper. However, interested readers may refer to [7] for further discussion on DSMT.

3. Benefits and Pitfalls

This section will detail the benefits derived from creating an architectural simulator from NetSim and why certain choices were made with regard to NetSim.

3.1. Benefits

The major benefits are (1) programmability, (2) architectural accuracy, and (3) flexibility.

Programmability

One of the major advantages of developing NetSim with C# is the increase in programmability. C# is an object-oriented programming language that is a cross between C++ and Java, and blends the advantages of both languages into one. Among these advantages are accessing low-level function calls and garbage collecting. In essence, C# removes the responsibility of programmer to handle memory clean-up and provides libraries full of basic, fast, and efficient data structures that are common in most programs. This level of abstraction allows the programmer concentrate on the algorithms and objects for the simulator while relying on pre-built methods to handle the mundane programming tasks [3][4].

C# is built on Microsoft's .NET Framework, which becomes a powerful tool for NetSim. In fact, the 'Net' in NetSim is a tribute to the .NET Framework. Programs written for .NET are inherently compiled to a .NET assembly, which is a type of interpreted language. When ran, a Just-In-Time (JIT) compiler will compile the assembly into the machine code. Thus, like Java, any compiled .NET assembly can be executed on any platform that supports the .NET Framework. So far, NetSim has been successfully run on both the Windows XP and Linux environments. Most other simulators, SimpleScalar included, are targeted for a specific OS environment and processor and trying to port the simulator to another environment requires extensive modifications to the source code and several recompiles. NetSim does not suffer this penalty and can be run in any environment where there the .NET Framework is supported without modifying the source code.

The .NET Framework also has the ability to compile multiple programming languages into a single assembly. This feature allows a developer to use some legacy C/C++ code in conjunction with NetSim to create a new simulator. Therefore, .NET lends a great deal to the increase in programmability within NetSim.

Architectural Accuracy

The increase in the amount of architectural accuracy can be derived from the object-oriented nature of NetSim. Microarchitectures are inherently object-oriented, and any architecture, from a five-stage pipeline processor to a superscalar processor, has a high-level block diagram that represents the overall architecture of the system. The block diagrams illustrate the major components, structures, and communication paths. By this nature, it is very easy to imagine each component in a block diagram as a class object and the communication lines as public methods.

In addition, each component can be broken down into smaller components, and if need be, to the gate level. This allows for a level of architectural accuracy that cannot be matched in a non-object-oriented simulator. Therefore, NetSim can be used to create a simulator with a direct one-to-one mapping between a block diagram and the NetSim components. The source code can be easily followed and the architecture can be accurately simulated. Moreover, the modules are usually small and easy to implement

In an actual microprocessor, each component is responsible for only a handful of tasks and manages their own data. This is analogous to a class with a protected data structure and private methods that manipulate that data. Simulators based on C and even C++ will globalize all data structures and expect certain global functions to interact with the data correctly. Because of this, the simulators will often only mimic the behavior of the architecture on the back-end, but perform all the calculations and data manipulation ahead of time. With NetSim, the calculations and data manipulations are done within the correct modules and at the correct times.

An added benefit to using an object-oriented approach is the ability to easily gather statistics from simulations (see Section 5 for examples). These statistics can be used to quickly identify and evaluate bottlenecks within an architecture as a means to compare different schemes and algorithms for performance. Although, it is possible to gather similar statistics in other simulators, significant additions and changes are required. With the use of the Statistics Database module in NetSim, only 2-3 lines of code are required.

The Statistics Database keeps track of each function delegate, or function pointer, within any module that produces metrics. This enables each object to keep track of their own data and statistics and to only pass the information along to the Statistics Database when needed. This method maintains data abstraction within the object-oriented programming environment while allowing the Statistics Database to be the sole provider of statistics for the entire system.

Flexibility

Due to the object-oriented techniques used to create the NetSim environment, it is relatively easy to alter architectural components for comparative analysis. Modifying the behavior of various components in most of the currently available simulators can be a very difficult task requiring numerous global changes to the code. With NetSim, changes can be confined to only the components affected.

Changing a single component can be as simple as creating a new object that inherits the original component and overrides any functions that will behave differently. If extensive changes need to be made to the module, then a new object can be created which uses the same interface. The new module can then replace the old module within the existing architecture.

If the changes would require additional information to be exchanged between units then it will be necessary to add new functions to the components involved. But even in this case, the changes are confined solely to these components. Global changes would be unnecessary.

When creating an entirely new architectural model using NetSim, the amount of work is significantly less than with other simulators. Instead of starting from scratch, base components would be reused and only new or heavily altered components would be written from scratch. For example a 5-stage pipeline simulator could reuse components for the fetch stage, the register file, the memory, and the functional units. The only components needed to complete it would be the decode and write-back stages. Forwarding could be accomplished by using the Common Data Bus object supplied in NetSimHP. With a minimal amount of new code, an entirely new simulation model can be implemented.

NetSim can also be retargeted without making major changes to the simulator. A new Instruction Database can be created based on the target ISA. This handles instruction decoding and execution so none of the core components will require changes. Currently the Loader supports ECOFF and ELF file formats. If an additional format is required then a parsing routine will need to be created for it. There may also need to be changes made to the Syscall module. If the new system calls are similar to the existing Unix style calls then the changes would most likely be minor. Depending on the target this could be a very involved task.

3.2. Pitfalls

With all the benefits involved with using NetSim, there is one major drawback, the time it takes for a simulation to complete. Since NetSim is object-oriented, there is additional overhead involved in the actual execution of programs. This overhead comes from the additional function calls, dynamic object instantiation, and garbage clean up. Therefore, it will not be possible to achieve the same speeds that C-based simulators can run.

But this is only a pitfall when looked at in terms of simulation time. When the time to develop the simulator is added to the simulation time, NetSim becomes a faster simulator. For example, a simulation using NetSim might be an order of magnitude longer to run than using SimpleScalar, but creating a simulator from NetSim can be almost an order of magnitude faster than creating one from SimpleScalar. A comparison of the simulation time is discussed in Section 5.

4. SuperSim

The first major cycle-accurate simulator developed using NetSim is SuperSim. It models a PowerPC-style superscalar architecture with reservation stations distributed by functional units. A version of the Tomasulo algorithm is used to handle forwarding and dependencies [5].

Figure 7 illustrates how SuperSim uses the NetSimHP components to create the superscalar processor. Note that this figure is almost identical to the block diagram of the superscalar architecture Supersim is based on. Not only are the block diagrams similar but the interaction between the various modules are also handled in the same manor as the original architecture.

SuperSim also supports a fast forward capability using FastMod. This allows the simulator to switch to a functional execution mode for any number of instructions and fast-forward through portions of a program. For example, it is possible to fast forward though the first 100,000 cycles, run four million cycles in full simulation mode, and then fast forward to the end of the program. This allows large benchmarks to be executed more quickly while still gathering detailed statistic from the core of the benchmark.

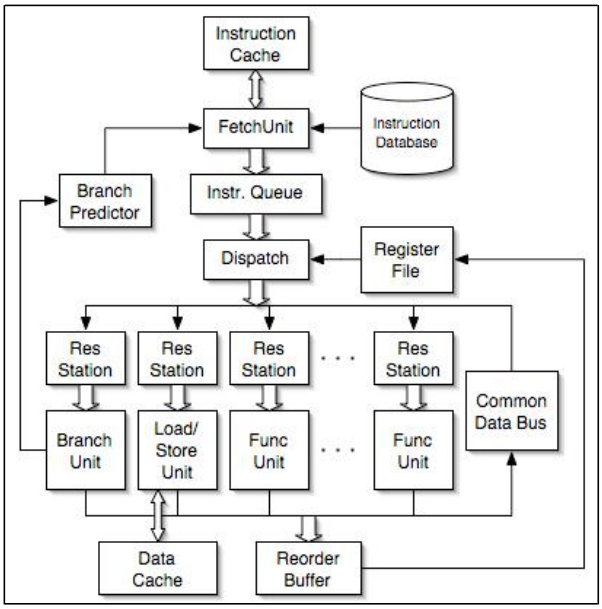

Figure 7: SuperSim block diagram using NetSim components.

SuperSim was validated on a large number of SPEC95 and SPEC2000 benchmarks [6]. The entire

mately six weeks for FastSim, and another two weeks

5. Simulation Results

To compare the NetSim environment to SimpleScalar, simulations were run using the compiled binaries supplied by Postiff, et al. [8] and used a small subset of the SPEC 2000 benchmarks [9]. The simulators were run on similar Pentium 4 computers with Linux used for sim-outorder and Windows XP used for SuperSim. Figure 8 shows the performance of each simulator for the four benchmarks in terms of instructions per second (IPS) and cycles per second (CPS). On average, sim-outorder was around 18 times faster in executing instructions than SuperSim but only about 7 times faster for each clock cycle. It is important to note that the cycles per instruction (CPI) of the architectural

outorder, thus requiring additional clock cycles to execute the benchmark and creating a lower IPS value.

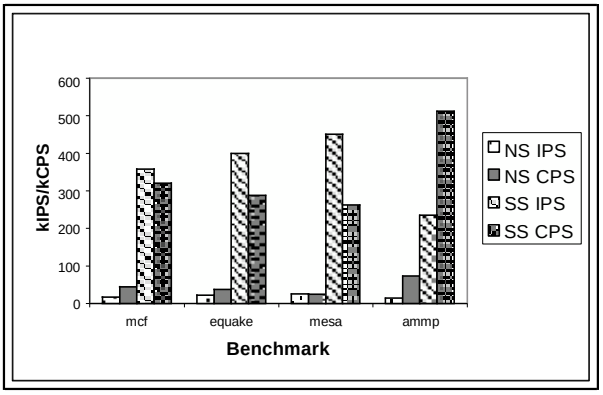

Figure 8: NetSim's (NS) SuperSim and SimpleScalar's (SS) sim-outorder speed performance.

The detailed architectural model used by SuperSim allows for a wide range of performance statistics to be gathered. This includes data that would be difficult to determine in simpler or less accurate simulators. An example of this is seen in Figure 9. The chart shows the issue performance of the dispatch unit as well as a detailed breakdown of the causes of all unused issue slots. The failed issue attempts are tracked based on which other component caused the failure, either an empty instruction queue, a full reorder buffer or no free reservation stations of the correct type. Figure 10 shows another example where keeping track of the busy status of functional units provides a metric for the overall function unit utilization. These metrics can be tracked with only a few additional lines of coded in each of the respective units.

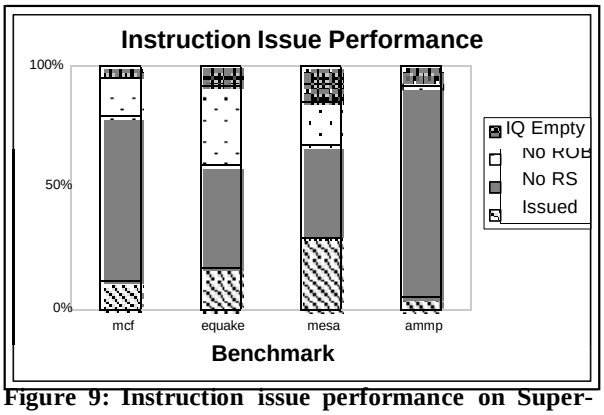

Figure 9: Instruction issue performance on SuperSim.

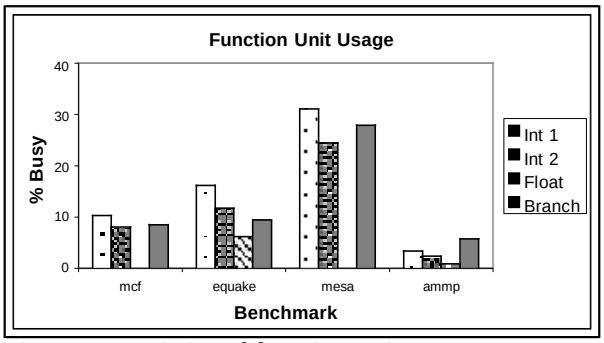

Figure 7: Statistics of function unit usage on SuperSim.

6. Conclusion and Future Work

NetSim was not intended to be the fastest simulator suite, but rather the most flexible and accurate simulator suite. By using the object-oriented model of C#, designers can use NetSim to cut down the simulator development time by orders of magnitude and still develop extremely accurate simulators. This allows designers to concentrate on the evaluation, rather than the development, of new design ideas.

Our future plan is to dramatically improve the speed of NetSim. It would be ideal to start optimizing the base components and attempt to get NetSim's FastSim to perform as fast as SimpleScalar's sim-fast simulator. Additional performance improvement to the simulators can be achieved by reducing the number of objects constructed and deconstructed during each cycle. The removal of legacy code and objects as well as minimizing the amount of data conversions will also enhance the performance.

Our ultimate goal is to develop a cycle-accurate multithreaded simulator, DMSTSim, using NetSim. NetSim will play an important role in researching key issues in exploiting thread level parallelism, such as

loop detection, thread management, inter-thread dependence speculation, and memory parallelism. Because of the architectural accuracy of NetSim, new and different types of metrics can be used to determine performances and bottlenecks. This will lead to a better multithreaded architecture design.