SAND2004-6238C

The Structural Simulation Toolkit: A Tool for Bridging the Architectural/Microarchitectural Evaluation Gap

Richard Murphy⋆† Arun Rodrigues⋆† Peter Kogge⋆ Keith Underwood†

University of Notre Dame⋆
Computer Science and Engineering Department
384 Fitzpatrick Hall
Notre Dame, IN 46545

Sandia National Lab* †
PO Box 5800
MS-1110
Albuquerque, NM 87185-1110

{rcm,kogge}@cse.nd.edu, {afrodri,kdunder}@sandia.gov

Abstract

Modern supercomputers involve complex interactions between complex components crossing tens of clock domains — per node. Thousands (and soon hundreds of thousands) of nodes are used to achieve the level of performance needed to address the nation's most challenging problems. This growing complexity makes system design ever more challenging; however, microprocessors are facing mounting challenges in delivering performance on scientific applications. This has led to a drive for innovation in systems where the impact of innovation is hard to predict. The Structural Simulation Toolkit (SST) was developed to explore innovations in systems where the processor and memory interact with the programming model and communications system. It is designed to be an open source framework that unifies hybrid discrete event simulation and time-stepping simulation to enable both detailed and abstracted simulations. The focus on parallel systems has been validated through experimentation with models of everything from processing in memory to conventional microprocessors connected by conventional network interfaces.

1. Introduction and Motivation

The solution of the nation's most demanding problems in science, defense, and intelligence requires massively scalable systems. On modern systems, an application written to a specific programming model runs on a set of processors that communicate through a network. These systems are rarely (if ever) simulated in their entirety because the community lacks the simulation environment needed to perform such an ambitious simulation. Instead, state of the art practice simulates components in isolation under a set of conditions that are believed to be representative. In the best of cases, the interaction of a pair of components is included in the simulation. Unfortunately, each generation of machines makes this methodology less tenable.

Bringing a new parallel computer architecture to market is expensive and time-consuming. Designing the microarchitecture, designing the network, fabricating the necessary silicon, developing the tool chain that goes from program to executable, and evaluating the combination with a believable workload to verify that it is in fact "better" are all significant undertakings. With each generation, the design time (and therefore cost) increases. Unfortunately, the impacts of changes in each of the components is not well understood. Current practice relies on a "rule of thumb" for the relative performance needs of the components and subtle interactions between the components are not discovered until system deployment. This often leads to systems that fail to meet their design objectives — failures that could be prevented with adequate system simulation.

What is needed is a simulation infrastructure that is available to every researcher that will support answering questions about system performance at scale (thousands of nodes). If ¡insert your novel processing technology here¿ accelerated an individual node by a factor of 10, would the system continue to scale efficiently? How will system performance change if we change the network topology? Can we enhance the network interface to improve MPI performance? What would happen if a change in programming model changed the message size and frequency?

Supercomputing systems are much more complex than can be modeled conveniently by existing, open, available research tools. Processing components can include microprocessors, vector processors, CMPs, processing in memory, and stream based architectures. Large scale machines are created by connecting processors through network inter-

*Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000.

face chips (which tend to be complex systems on chip designs) to routers that support a variety of topologies. While these components can be modeled in isolation, even the simulation of small system scale integration tends to require a "one-of-a-kind" effort. Furthermore, it is *impossible* to simulate systems in detail (near cycle accuracy) at the scales (thousands of components) where issues of scale *begin* to appear running applications for a meaningful amount of time (an hour of more of simulated time to capture the evolution of the application over time). There is a real need for a "plug and play" infrastructure where each component can be simulated at every level of detail. Processor, network, and router simulations must range from cycle accurate models to simplified statistical models. High level models must be tunable to match the performance characteristics of detailed models as closely as possible and the various levels of modeling must be easily combinable to enable studies of the impacts of minor microarchitectural perturbations on systems at the largest of scales.

The Structural Simulation Toolkit (SST) discussed in this paper was developed to help solve these problems, and to allow exploration of highly concurrent systems where the ISA, microarchitecture, network, and multi-node system issues need to be explored together. SST is modular and provides the ability to modify one characteristic at a time without having to rip up the rest, or modify support tools such as compilers until necessary or appropriate. The key to this is the explicit separation of program and instruction interpretation (the "front end") from instruction and microarchitectural timing (the "back end"), combined with a hybrid discrete event and time-stepping simulation framework. The interface to the front end supports multiple program formats such as instruction traces and compiled executables. The back end interface supports multiple levels of detail, from simple functional simulation to models of collections of heterogeneous processors. Introduction of new front or back end components can be done independently of the other.

Experiments to date have used this simulation structure to explore software and hardware techniques related to processing in memory, novel NIC architectures to support MPI communication, vector processing, and has recently been equipped with a three dimensional router model. The typical initial experimental costs have been in the range of a handful of days of development time and results in simulation performance comparable to the state of the art. Together, this indicates that such a structure can have a significant effect on future architectural research.

2. Related Work

A variety of simulators and simulation strategies are used in computer architecture, providing a range of features and functionality. A major goal of the Structural Simulation Toolkit is to build upon the success of these previous works.

2.1. Architectural Simulators

Many architectural simulators have been written to explore design issues on the processor or system level. These simulators represent programs by execution-based, trace-based, or stochastic mechanisms. They vary in level of detail, configurability, and focus.

SimpleScalar[7] is one of the most commonly used architectural simulation toolkits. It includes a number of execution-based simulators, ranging from simple execution to cache simulation to full simulation of an out-of-order processor and memory hierarchy (`sim-outorder`). A number of processor parameters can be adjusted, such as issue width, functional units, internal queue lengths, and cache characteristics. Though SimpleScalar only models a single conventional processor, several derivatives have extended its functionality. The SImulator for Multithreaded Computer Architecture [16] (SIMCA) was developed to explore multi-threaded architectures by augmenting the PISA ISA to include instructions for thread creation and control. Another derivative, MP_simplesim[23] presented multiprocessor cache simulation capabilities. However, this simulator only modeled cache interactions and did not attempt to model detailed timing. Other multiprocessor simulators include Solo[12], RSIM[17], ML-RSIM [30], Tango[13], and MINT[34].

Other simulators have been developed to enable specific functionality. For example, SimOS[15], Talisman[6], and ML-RSIM[30] are capable of supporting the execution of an OS. SPIM[20] was developed as a teaching tool for exploring the MIPS ISA. Others, such as `simg4`[25] and `simg5`, were developed to model a specific processor (the PowerPC 7400 and 970) in detail. Both of these simulators are trace-based, and provide a high level of detail, but lack a high degree of parameterization. Additionally, they do not attempt to simulate a full system with multiple processors or network.

The ASIM[8] performance model framework is comprised of a set of modules which can be composed to form different architectures. A novel feature of ASIM is the partial separation of the performance model for system components from the program execution.

Other modular simulation efforts include the Liberty Simulation Environment[33], which has developed a number of modules in its own LSS language, and Microlib which provides a number of modules in SystemC[1].

The message PAssing computeR SIMulator, PARSIM[32] was developed to explore algorithms and network topologies for parallel computers. It models program execution as a generalized algorithm divided into computation and communication. Processor speed and network characteristics

can be parameterized, but no attempt is made to model the internals of the processor.

Some simulators focus on network modeling, such as DaSSF[22] and GTNetS[27]. These simulators can simulate large networks in great detail, allowing for a variety of different topologies, protocols, and link characteristics. However, these simulators do not attempt to model the exact execution of a program or the internal architecture of a system. Instead program communication patterns are generated stochastically.

2.2. Models of Computation

An important characteristic of a simulator is the underlying model of computation[21] which defines how time is advanced and how components interact. Most architectural simulators use one of the following approach:

- **Synchronous**, or time-stepped, models discretize time into fixed increments. At each increment, all components are evaluated and the effects are propagated. Simulators using this approach include SimpleScalar [7], SPIM [20], or PARSIM [32].

- **Discrete Event**, or event-driven, approaches only evaluate transition functions if component inputs change, or if an event is received from another component in the system.

- **Process-model** uses processes to model components. Processes represent the activities of some component, how it interacts with other components and how it advances its own state. The framework provides a facility to schedule execution of processes and switch between them to simulate simultaneous action [31]. Examples of this include CSIM and SSF.

- **Hybrid** simulators combine two or more simulation techniques. For example, `Enkidu` uses both time-stepped and discrete event models and SSF [22] uses both process-based and discrete models.

Discrete event simulation is popular for general simulation, battle space simulation, gate-level microarchitectures, and network simulation, where many components generate events at varied rates. Synchronous simulation is more popular in architectural processor simulation, where every component would often generate an event every clock cycle.

When inter-component interaction is infrequent, discrete event simulations have an advantage, in that they can "skip" intermediate time steps between event arrivals. Note that in a pure discrete event simulation, if a component needs to change its state it must send an event to itself. However, when events are relatively infrequent, asynchronous simulation tends to incur less overhead.

Process-oriented simulation allows components to update their state without an explicit event. As such, they can be an elegant way of expressing frequent, repetitive, behavior (like a time-stepped system) while still allowing non-uniform time steps (like an asynchronous system). However, process context switching can become a significant source of overhead [31, 22].

Many works have looked at simulation speed for given simulations, or at the overhead provided by a particular framework [26]. Much of this work has focused on parallel discrete event simulation[11, 19], often focusing on the benefits of a particular language or framework such as SSF[22] or PARSEC [32].

3. Framework

Structural Simulation Toolkitis an architectural simulator implemented in about 45,000 lines of C++. It is composed of four primary elements (see Figure 1): the **Front Ends**, which model the execution of a program; the **Back Ends**, which model architectural components of the system; the **Processor/Thread Interface**, which allows the front and back ends to interact; and **`Enkidu`**[29], a component-based discrete event and synchronous simulation framework that coordinates communication between back end components and models the passage of time. To provide modularity and reconfigurability, it is possible to select a front end and choose a variety of back end components at run time. This allows the user to explore a variety of hardware configurations while using the execution model best suited to the available toolset.

3.1. Front End

The front end is responsible for generating instructions and threads to be processed by the back end. The front end also defines how the execution of these instructions changes the programmer visible state. Put another way, the front end simulates the "software" or programmer visible side of the system.

The front end provides a loader which loads the program into the simulated memory before the simulation begins. Once simulation begins, the front end provides instruction and thread objects to the back end (see Section 3.3). Most importantly, it determines how the program state (usually memory and registers) are modified by the instruction's execution. This usually involves a lookup of the instruction in a "big case statement" to ascertain how state should be updated. This "big case statement" performs a similar function to SimpleScalar's[7] `ss.def` or `powerpc.def` files, or SPIM's[20] `run.c`. Currently, three front ends exist and can be selected at run-time.

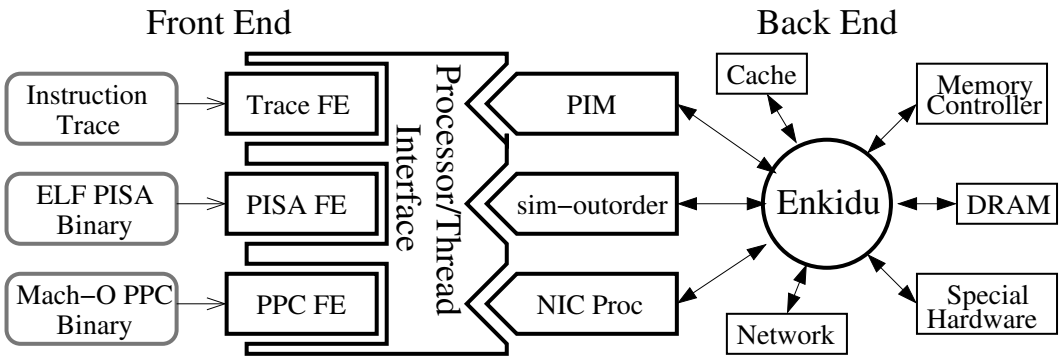

Figure 1. Simulator Framework and Component

- **PISA**: an execution-based front end which uses the PISA ISA and the ELF executable format. The PISA ISA is augmented with extra instructions to provide added functionality required to simulate experimental hardware. These instructions were added to to explore low-cost thread spawning and different synchronization primitives, to allow a processor to communicate with a subordinate processor on a NIC, to access specialized hardware such as network interfaces, DMA engines, or custom components, and to allow a processor to select between different address spaces. The PISA front end allows use of modified versions of the `gcc` or `g++` compilers version 2.7.2.

- **PPC**: An execution-based front end which uses the PowerPC ISA[24] and the MachO[4] executable format. This PowerPC ISA is augmented in a manner similar to the PISA ISA. It also includes a small subset[1] of the AltiVec vector extensions. The MachO format is the standard format for MacOS X executables and allows the use of binaries created by a number of modern compilers. It has been tested with `gcc` and `g++` versions 3.1 and 3.3, g77 version 3.4, IBM XLF 8.1 FORTRAN 77/90 compilers, and Absoft 8.0 FORTRAN 77/90 compilers.

- **Trace**: A trace-based front end which uses the tt6 PowerPC instruction trace format[3] generated by the `amber` trace generator[2]. A tt6 file contains a record of each instruction executed by the target program. This record contains the opcode, registers, immediate and any memory addresses accessed or the branch target. The Trace front end also supports limited speculative execution. Before the simulation begins, the trace is scanned and the address of each instruction, along with that instruction's record, is recorded. If the back end mispredicts a branch and requests an instruction at the wrong address, the front end will return the instruction at the incorrect address, as long as that address is executed somewhere in the trace. Because most mispredicts still return a valid instruction address, the back end is able to simulate the effect of the speculatively executed instructions.

[1] `lvx`, `stvx`, `vspltw`, `cmpequh[.]`, and `vand` instructions

3.2. Back End

The back end is responsible for modeling the hardware of the system. Specifically, it consumes instructions and threads generated by the front end and determines how long it would take for the specified hardware to execute them. The back end is composed of many different `Enkidu` components (see Section 3.4). These components represent physical components such as processors, networks, memory controllers, DRAM chips, or other system elements. Components can communicate through `Enkidu`'s discrete event message passing system.

Several back end components are designed to interact (Figure 2). Current components include:

- **"Level 0" Basic Processor**: Execution-only, no timing. Similar to `sim-fast`.

- **PIM Chip**: a Processor-in-memory(PIM) chip that contains one or more PIM processors. It also models instruction caches, on-chip DRAM or SRAM memory, inter- and intra-chip communication between PIM processors, and optional connection to a backing memory controller.

- **PIM Processor**: A simple processor, suitable for combination with memory. Models a multithreaded in-order pipeline without branch prediction. Can be configured for different pipeline lengths, forwarding, and thread scheduling mechanisms. Supports non-local memory accesses by remote instructions or thread migration.

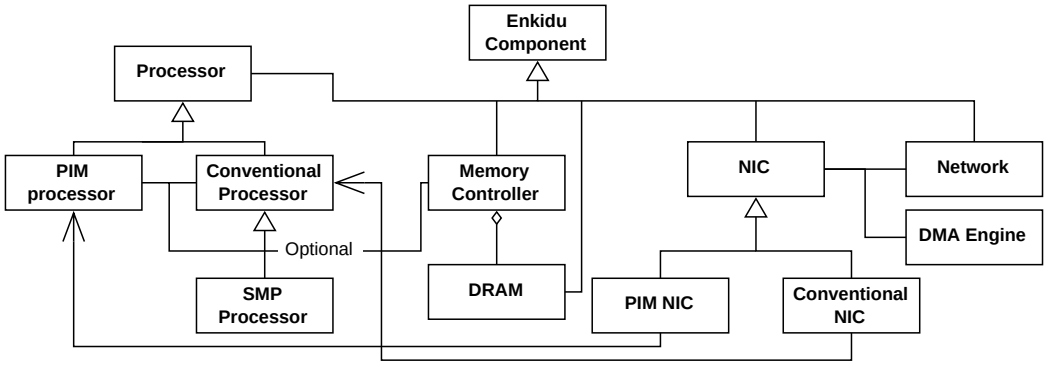

Figure 2. Some back end component classes and their relationships

- **Conventional Processor**: A conventional, out-of-order, multi-issue processor based on the model in SimpleScalar's `sim-outorder`. It can use SimpleScalar's memory model or can connect to a memory controller component. It can be configured to handle multiple threads with simple run-to-completion scheduling.
- **SMP Processor**: An SMP variant of the conventional processor with a simple snooping coherency policy.
- **Memory Controller**: A simple memory controller model which simulates bus contention, bandwidth constraints, communication latency, and DRAM interleaving.
- **DRAM**: A model of a DRAM chip with a configurable number of resizable DRAM banks. Open row (open page) latency and contention is modeled, and the number of open rows is adjustable.
- **Conventional NIC**: a NIC controlled by a conventional out-of-order processor. It also models a separate NIC bus, local memory and a Direct Memory Acess (DMA) component.
- **PIM NIC**: Similar to conventional NIC, but using a PIM Chip instead of a conventional processor.
- **Simple Network**: A simple network model connecting NICs on different systems. Latency and bandwidth effects are modeled, but no topology.
- **Mesh Network**: A configurable mesh topology network with detailed routers and link components. Routers modeled internal buffers and contentions.

3.3. Interface

They `Processor/Thread` interface is the key bridge between the front end and back end. This interface defines

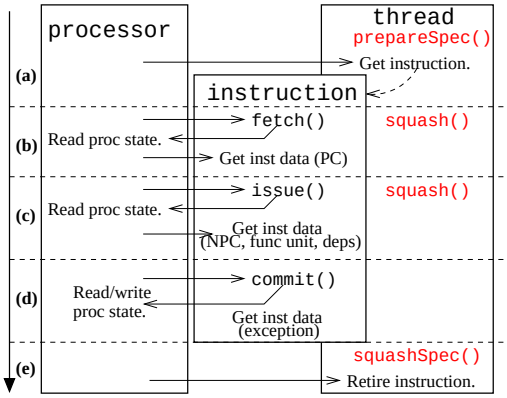

Figure 3. `Processor/Thread` Interface

three abstract classes: `processor`, `thread`, and `instruction`. A `processor` is a back end component which can execute `instructions` belonging to one or more `threads`. Examples of `processors` include the PIM Processor and the Conventional processor. Each front end defines a `thread` class and `instruction` class.

The front end simulates how the program executes from a software perspective — it ignores hardware and timing details and just looks at how the state of its registers and memory is modified by the `instructions`. The back end `processor` simulates how the program executes from a hardware perspective — it ignores the specifics of what values are written where and focuses on modeling the timing details of which components are accessed, how long memory transactions take, and other microarchitectural details. The back end `processor` provides access to a storage area for the front end to store memory and register values. This allows multiple threads to interact on the same data.

The execution of an instruction demonstrating the typical interaction between `processor` and `thread` is shown in Figure 3. In (a), the `processor` begins by requesting an

instruction from the `thread`. This request can be for an instruction at a given address, or can simply be for "the next instruction to execute."

The `processor` simulates the execution of the instruction in three stages: fetch, issue, and commit. These stages can be performed at different cycles, and they can be overlapped with other instructions. First, the `processor` calls the `fetch()` method on the instruction (b). The `fetch()` method may call upon the `processor` to access simulator state, usually to access the instruction memory. After `fetch()` has completed, the `processor` can query the `instruction` for information required to model instruction timing and resource use.

A similar process is performed for the `issue()` stage(c) and `commit()`(d) stages. The processor calls the function which may invoke processor functions. After the `instruction` member function completes, the processor can query the `instruction` to find characteristics such as which functional unit(s) it requires, its register dependencies, and any exception.

Lastly, the `processor` directs the `thread` to retire the `instruction`(e). If the `processor` needs to rollback the execution of one or more `instructions` (perhaps due to an exception) it may due so with the `thread` class' `squash()` function.

3.3.1. Speculative Execution

To model many processors accurately, speculative execution must be simulated. To accommodate this, the SST uses a copy-on-write policy towards memory and register state. If a `processor` detects the start of speculative execution, it informs the `thread`. The thread then treats future `commit()`s as speculative. Any state changes caused by these `commit()`s are written to a copy of the original state. When the speculation is over the `thread` and `processor` discard the speculative state copies.

3.4. Hybrid simulation

Modern processors can be represented as a series of buffers which store instructions and data, separated by logic which acts upon those instructions. Data flows from buffer to buffer according to a strict centralized clock. For architectural simulation, it is possible to say that all events take place in synchronization with this clock. Processors often have dozens of instructions in various stages of execution during each processor clock cycle. As a result, several transition events can be expected to occur each cycle.

Parallel architectures often have two distinct types of transitions. Transitions within a processor occur several times a cycle. Additionally, there will be infrequent inter-processor communication events. For this reason a hybrid simulation framework which combines synchronous time-stepping and discrete event-passing provides the most flexibility and lowest overhead for parallel architectural simulation.

The Structural Simulation Toolkit is built around `Enkidu`, a hybrid simulation framework. In `Enkidu`, `component` objects represent each physical component of the system. Each `component` is evaluated every clock cycle and advances its internal state. In addition, `components` can communicate by passing event messages to each other in an asynchronous manner. Optimizing for the common case in architectural simulation, `Enkidu` provides low-overhead synchronous time-stepping to handle the bulk of the functionality. For the infrequent communication between `components`, the discrete event mechanism is provided.

Table 1. Default SimpleScalar Configuration for validation

Parameter	Value
issue width	4
commit width	4
ialu	4
imult	1
memports	2
bpred	bimod
mem lat	18/2

4. Validation

A simulation framework must be both correct and robust. Correctness requires that the simulator produce results that match a known target. In this case, SimpleScalar is used as one of the back ends, and so SimpleScalar provides a point of comparison. Robustness, in the sense that the simulator provides the functionality needed to simulate a wide variety of systems, was a primary goal of this effort. To verify robustness, a number of system and processor architectures were simulated.

4.1. Correctness

Validation against SimpleScalar used the PISA ISA front end and the default configuration for the SimpleScalar back end (Table 1). Validation used several applications:

- `speedTest`: A simple program consisting of a linked list creation and traversal (a memory intensive operation) and a matrix-matrix multiply (a cache and floating-point unit intensive operation) was used. Several system calls also test underlying system libraries.

- `anagram`: An anagram finder from the Instructor Benchmarks[5] provided on the SimpleScalar website as a precompiled PISA executable. It uses a dictionary of 24,474 words.

- `go`: A go-playing benchmark program.

The output of these simulations was checked against known correct values to show correct execution of the program. All outputs were correct.

To show correct simulation, simulation statistics were compared. The total number of instructions committed, execution cycles, branch prediction hit rate, and cache hit rate for L1s, L2, and TLBs were compared. Some known small discrepancies exist between the SST and SimpleScalar. These include small differences in how certain system calls are handled and stack layout details. However, these differences prove to be minor (Table 2). For all measurements, the SST was within a fraction of a percent.

4.2. Robustness

Demonstrating the robustness of the simulator framework was accomplished by implementing a number of system and processor architectures on the back end and a number of front ends.

- **Initial Framework**: The initial framework included the `processor/thread` interface, basic configuration utilities, a PISA-based front end and a simple multiprocessor back end. The back end modeled thread migration and remote accesses between processors, but not timing.

- **PIM**: A simple Processor-in-Memory (PIM) model includeing PIM chips and network, and thread migration (see Section 3.2).

- **Dynamic PowerPC Front End**: A front end for the PowerPC ISA was developed, including a dynamic loader for the MachO executable format, used on MacOS. This allowed the use of a wider range of modern compilers than the existing PISA toolchain. To simulate large "real-world" scientific applications, it is often necessary to support dynamically loaded librariesThe MacOS X dynamic link editor (`dyld`), was modified to provide dynamic library support. This allowed the simulation of large applications such as CTH[9], a 500,000 line mixed FORTRAN/C/C++ shock physics application with adaptive mesh refinement and visualization capabilities.

- **Altivec**: A subset of the Altivec vector instruction set for PowerPC was added.

- **Simple Scalar**: `sim-outorder` was integrated with the simulator to simulate conventional out of order processors (see Section 3.2).

- **NIC**: A programmable NIC (see Section 3.2) and components with a simple network model. A subset of the Message Passing Interface[10] (MPI) was implemented and tested on this architecture.

- **PIM NIC**: The previously constructed NIC was generalized to allow a Processor-in-Memory as its controller.

- **Associative memory structure**: A specialized associative memory structure to accelerate MPI was integrated onto the NIC. The parameters for this structure were based upon more detailed VHDL simulation.

- **SMP**:Basic SMP functionality[14] was added to allow simulation of multiple conventional processors sharing a common snooping bus.

- **Trace-based Front end**: A front end which accepts PowerPC instruction traces in the tt6 format was constructed. This allowed simulation of programs which were difficult or impossible to compile for other front ends, or of selected sections of a program too large to simulate in its entirety.

- **Mesh Network**: A configurable network was added to allow more scalable and detailed MPP simulations.

5. Results

A simulation infrastructure has two metrics by which it should be evaluated: the speed of simulation and the ease with which it can be used for its purpose. The speed of simulation is a straightforward measurement and is compared below to other architectural simulators. In contrast, "ease of use" is a much more difficult metric to evaluate. To attempt to characterize this metric, the "effort" in terms of both lines of code and days to create that code was captured from multiple implementations of architectural features and is presented below.

5.1. Performance

To measure the performance of the simulator, the `speedTest` program from Section 4.1 was used. Figure 4 compares the performance of the Structural Simulation Toolkit with both SimpleScalar and `simg4`, a trace-based simulator for the PowerPC G4 (7400). The Structural Simulation Toolkit trace-based front end uses the same format as the `simg4` simulator, but the back ends are different. Above, the average

Table 2. Simulation correctness validation: Differences

Benchmark	Instructions	Branch Pred.	L1 Hit Rate	L2 Hit Rate	TLB Hit Rate
`speedTest`	$< 0.4\%$	$< 0.2\%$	$< 0.2\%$	$< 0.3\%$	$< 0.3\%$
`anagram`	≈ 0	$< 0.2\%$	$< 0.3\%$	$< 0.3\%$	$< 0.1\%$
`go`	$< 0.3\%$	$< 0.3\%$	$< 0.1\%$	$< 0.5\%$	$< 0.1\%$

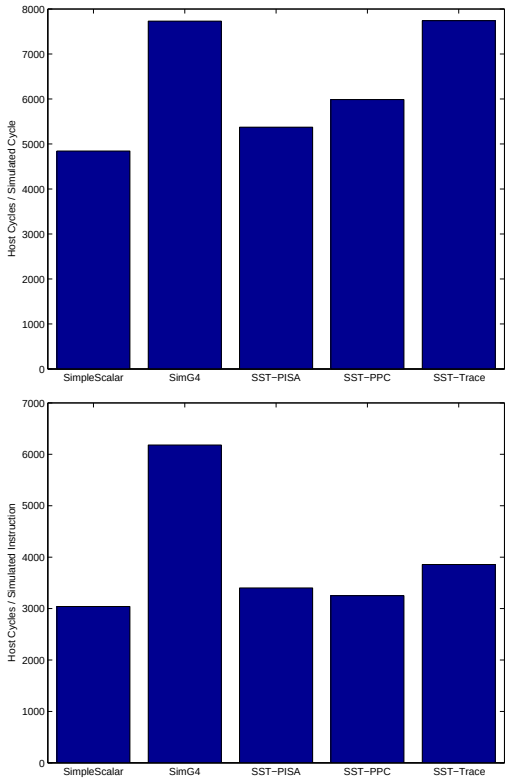

Figure 4. Simulator Comparison: the Structural Simulation Toolkit incurs only a small overhead against SimpleScalar.

number of host processor cycles required to complete a simulated cycle are shown.

Comparing the SST-PISA data point to the SimpleScalar data point indicates that the modularity of the framework adds approximately 10% overhead. This is a relatively low overhead for separating the instruction decode from the microarchitectural simulation. Using the PowerPC front end increases the overhead slightly per simulated cycle and the trace based front end is much heavier weight[2]

Moving to the lower half of Figure 4, gives an "instruction to instruction" ratio between simulated and host instructions. This effectively divides the data on the upper graph by the IPC achieved on the simulated platform. The performance relationship between the PISA based SimpleScalar and the PISA front end of the Structural Simulation Toolkit remains constant; however, the PPC front end and trace based front end show significant relative gains. This occurs because the PPC and trace based front ends use a newer compiler (gcc 3.3) rather than the older compiler (gcc 2.7.2) available for PISA.

The same benchmark was used to compare the three different front ends with a variety of back end configurations. Figure 5 shows the impact of the interaction between the ISA and the microarchitecture. First, SimpleScalar's main memory model was replaced with a more complex model including a memory controller and DRAM chips. This inflicted a very small overhead (0.5% - 1.7%), even though the the new memory model requires the addition of several components and considerable event messaging. This small cost validates the low overhead of the `Enkidu` framework. The front ends were also compared simulating several PIM configurations in which the number of processors on a chip was varied. Because the benchmark program is single threaded, each processor is assigned a region of memory to "own". Whenever the program requires memory owned by a different processor it must emit a remote instruction. Thus, adding extra processors decreases the simulator efficiency because the extra capacity is not used and additional communication overhead is required.

Models of multiple system level configurations were also benchmarked and the results are shown in Figure 6. A 2 processor SMP system was tested with the SPLASH2 non-contiguous LU kernel. The other configurations tested con-

[2]This is currently believed to be a difference in the level of optimization effort applied to the trace based front end.

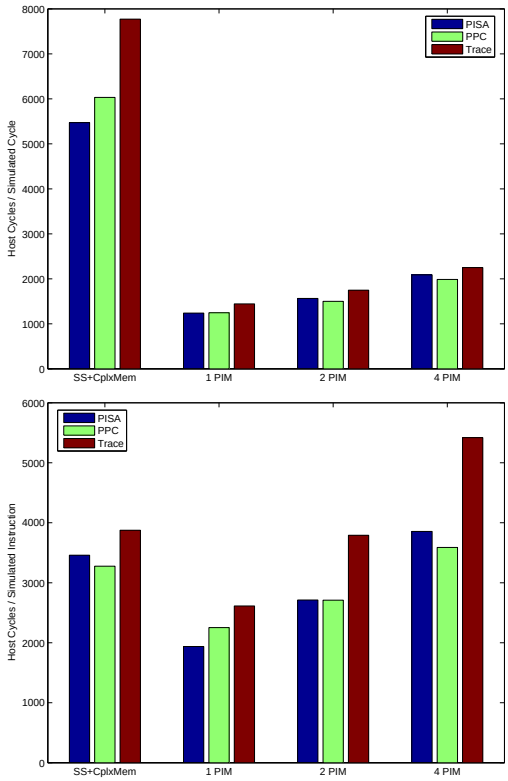

Figure 5. Front End Comparison:

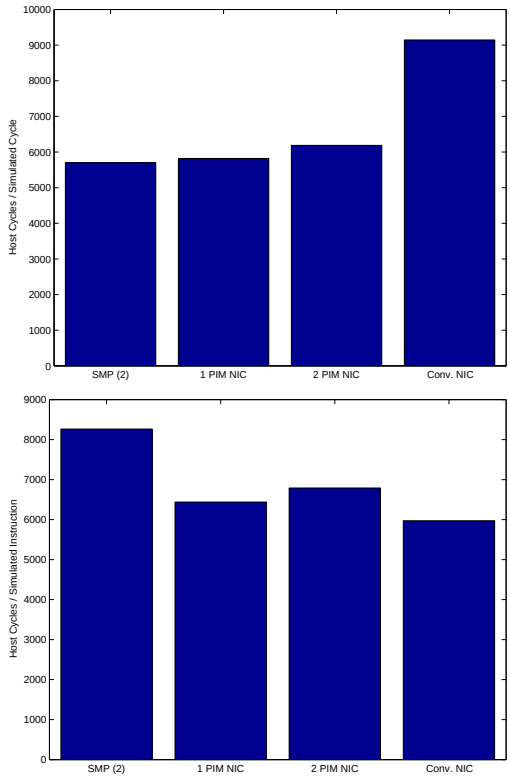

Figure 6. Multi-processor configurations Comparison: All using PPC front end.

ventional processors with programmable NICs attached by a simple network. A small MPI benchmark was constructed that posts several 16K message buffers and then measures the latency of MPI message matching. The back end configurations all used an 8-way out of order processor for the main processors. The NIC's processor was modeled as a 2-way conventional processor, or as one or two PIM processors. The relatively high cost per cycle for the NIC-based systems is because the NIC processor was running at 1/4 of the clock rate of the main processor. These are much cheaper operations in terms of host cycles per simulated instruction.

Another important performance measurement is the amount of simulation overhead incurred by a framework. Overhead becomes especially important when the number of components is high (such as a highly parallel architecture) or if components are relatively simple. Such simulators may become dominated by framework overhead.

Overhead can be measured by the number of instructions the host processor must execute for each simulated component for each simulated clock cycle. Such measurements of various simulators [28] reveal notable contrasts between the overheads of different models of computation. The num-

ber of host instructions per component per simulated cycle can vary from 57 for lightweight hybrid frameworks such as `Enkidu`, to 956 for more complex frameworks such as SystemC — over a 16 fold increase. Pure event-driven frameworks like YACSIM[18] fall somewhere between with 297. Experiments indicate that these overhead instructions execute with a IPC between 0.8 and 1.2 on modern processors.

Put another way, a simulation of 10 components with an overhead of 1000 instructions, executing on a 1Ghz machine with an IPC of 1.0 would be limited to 100,000 simulated cycles per real-world second. Whereas a similar setup with an overhead of 60 instructions would be capable of over 1,600,000 cycles per second.

The Structural Simulation Toolkit, built on `Enkidu`, provides a very low overhead by optimizing for the common case for architectural simulation.

5.2. Modeling Costs

The SST is flexible enough to simulate a range of architectural configurations, but the important metric is how hard it is to create new architectural configurations. New components have been added to the simulator to support a number of projects and the effort level to add those components is captured in Table 3. Many of the smaller projects (ISA extensions, new front ends) took only a handful of days to complete. Even the addition of large system components such as integrating SimpleScalar's [7] out-of-order processor model or adding a NIC and simple network model took less than two weeks.

A primary features of the Structural Simulation Toolkit is the ability to add new front and back ends. Adding a PIM back end complete with communication capabilities and the unique memory model of a PIM took approximately two weeks and slightly over 2000 lines of code. Adding the PowerPC front end required nearly 15,000 lines of code, but the vast majority was in header files enumerating instruction semantics. Indeed, over half of that is the "big case statement" defining the ISA. Similarly, the highly complex `sim-outorder` component of SimpleScalar was integrated in approximately two weeks. This indicates that it is reasonable to integrate existing simulation capabilities into the framework.

The most important capability of the SST is the ability to model multiprocessor systems. A number of examples tested this capability. A two processor basic SMP functionality was implemented (using SimpleScalar as the back end) with a snoopy cache coherency protocol in only two days and with only 320 lines of code. A more complex system connecting two processors through a pair of conventional network interface models (including details such as an embedded PowerPC processor, DMA engines, and a simple network model) with 11 days of effort. This was later extended to use a PIM processor with an additional 7 days of effort (including an updated MPI implementation) and only 686 lines of code. Additionally, a novel hardware feature to accelerate list processing was integrated in a single day.

6. Conclusions

The Structural Simulation Toolkit framework provides a flexible modeling environment for the independent exploration of programming model, ISA, and microarchitectural details. Multiple independent front ends and back ends have been developed that illustrate the feasibility of this approach and that provide a solid foundation for further research. The overhead of the simulator relative to a raw microarchitectural simulator such as SimpleScalar is only 10%, which is an acceptable trade-off for the flexibility offered. Furthermore, the Structural Simulation Toolkit employs a hybrid simulation model to enable low overhead simulation of parallel systems where processor, memory, and communication systems must interact. Multiple examples of these parallel systems (ranging from an SMP to a pair of independent nodes coupled with a standard network interface) were built to evaluate both the ease of use and performance of the simulator. In each case, the extensions were found to require a minimal level of effort while still delivering excellent performance.

7. Future Work

A number of enhancements are being developed for the Structural Simulation Toolkit. A key focus will be more back-end support including such things as chip multi-processing (CMP) and an improved network system. On the front-end, a synthetic workload generator will be added to support faster exploration of parallel system issues. The biggest enhancement in development is support for a parallel version of the system to explore massively parallel system configurations.

Current work is underway to port a full version of MPICH to the simulator. This will allow simulation of a wide range of scientific codes as well as facilitate MPI research.

References

[1] *SystemC User's Guide*, 2.0 edition.

[2] Apple Architecture Performance Groups. *Computer Hardware Understanding Development Tools 2.0 Reference Guide for MacOS X*. Apple Computer Inc, July 2002.

Table 3. Modeling Costs

Component set	Est. Time.	LOC	% .h
Initial Framework, limited PISA Front end, level 0 backend	2 days[a]	≈4000	68%[b]
pipelined PIM with forwarding, inter&intra-chip communication, memory hierarchy (caches, off-chip DRAM)	≈10 days	2260	44%
PowerPC Front End (including MachO loader)	unknown	14965	72%[c]
Addition of subset of Altivec vector ISA to PowerPC Front end	1 day	153	86%[d]
Integrate `sim-outorder` with framework. Combine with memory hierarchy (Memory controller & DRAM modeling).	10 days[e]	17047	≈ 31%
Conventional processor acting as NIC with interface to simple network, DMA engines, NIC bus, access to local and main memory. Simple network model and ability to instantiate multiple systems.	11 days	1116[f]	36%
Processor-in-Memory based NIC, including multi-processor PIM on NIC	7 days	686	15%
Integrate specialized associative memory structure on NIC	1 day	306	25%
Basic SMP functionality [14] for conventional processors.	2 days	320	35%
Trace-based front end	3 days	2088	77%
Mesh Network		≈1000	≈23%

[a] Using considerable preexisting code
[b] Includes `ss.def` file of 1169 lines.
[c] includes `powerpc.def` file of 7858 lines
[d] primarily additions to `powerpc.def`
[e] includes expansion of `processor/thread` interface to accommodate OOE.
[f] ≈15% of which was for network model.

[3] Apple Architecture Performance Groups. Amber trace format specification v1.4. Technical report, Apple Computer Inc, 2003.

[4] Apple Computer, Inc. *Mach-O Runtime Architecture*, August 2003.

[5] Todd Austin. Instructor benchmarks. http://www.simplescalar.com/benchmarks.html.

[6] Robert C. Bedichek. Talisman: fast and accurate multicomputer simulation. In *Proceedings of the 1995 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 14–24. ACM Press, 1995.

[7] Doug Burger and Todd Austin. *The SimpleScalar Tool Set, Version 2.0*. SimpleScalar LLC.

[8] Joel Emer, Pritpal Ahuja, Eric Borch, Artur Klauser, Chi-Keung Luk, Srilatha Manne, Shubhendu S. Mukherjee, Harish Patil, Steven Wallace, Nathan Binkert, Roger Espasa, and Toni Juan. Asim: A performance model framework. *Computer*, 35(2):68–76, 2002.

[9] Jr. E.S. Hertel, R.L. Bell, M.G. Elrick, A.V. Farnsworth, G.I. Kerley, J.M. McGlaun, S.V. Petney, S.A. Silling, P.A. Taylor, and L. Yarrington. CTH: A Software Family for Multi-Dimensional Shock Physics Analysis. In *Proceedings of the 19th International Symposium on Shock Waves, held at Marseille, France*, pages 377–382, July 1993.

[10] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, 1994.

[11] Richard Fujimoto. Parallel and distributed discrete event simulation: Algorithm s and applications. In G.W. Evans, M. Mollaghasemi, E.C. Russel, and W.E. Biles, editors, *Proceedings of the 1993 Windter Simulation Conference*, pages 106–115, 1993.

[12] Jeff Gibson, Robert Kunz, David Ofelt, Mark Horowitz, John Hennessy, and Mark Heinrich. Flash vs. (simulated) flash: closing the simulation loop. *SIGOPS Oper. Syst. Rev.*, 34(5):49–58, 2000.

[13] Stephen R. Goldschmidt. *Simulation of multiprocessors: accuracy and performance*. PhD thesis, 1993.

[14] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., San Francisco, California, 1996.

[15] Steve Herrod, Mendel Rosenblum, Edouard Bugnion, Scott Devine, Robert Bosch, John Chapin, Kinshuk Govil, Dan Teodosiu, Emmett Witchel, and Ben

Verghese. *The SimOS Simulation Enviornment*. Stanford University, 1998.

[16] Jian Huang. *The SImulator for Multithreaded Computer Architecture*. University of Minnesota, Minneapolis, 1.2 edition, June 2000.

[17] C.J. Huges, V.S. Pai, P. Ranganathan, and V. Adve. Rsim: Simulating shared-memory multiprocessors with ilp processors. *IEEE Computer*, 32(2), February 2002.

[18] J. Robert Jump. *YACSIM Regerence Manual*. Rice University, Houston, TX, version 2.1 edition, March 1993.

[19] Pavlos Konas and Pen-Chung Yew. Improved parallel architectural simulations on shared-memory multi processors. In *Proceedings of the 8th Workshop on Parallel and Distributed Si mulation (PADS '94)*, 1994.

[20] James R. Larus. *Computer Organization and Design: The Hardware/Software Interface*, chapter Appendix A. Morgan Kaufmann, third edition, August 2004.

[21] Edward A. Lee. Overview of the ptolemy project. Technical Report M03/25, University of California, Berkeley, July 2003.

[22] Jason Liu and David M. Nicol. *DaSSF 3.1 User's Manual*. Dartmouth, April 2001.

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

[24] Motorola Inc. *MPC7400 RISC Microprocessor User's Manual*, March 2000.

[25] Motorola System Performance Modeling and Simulation Group. *Sim_G4 v1.4.1 User's Guide*, 1998. Available as part of Apple Computer's CHUD tool suite.

[26] Wei Qin. How fast is system c? http://www.ee.princeton.edu/ wqin/systemc.htm, June 2003.

[27] George F. Riley. The georgia tech network simulator. In *Proceedings of the ACM SIGCOMM workshop on Models, methods and tools for reproducible network research*, pages 5–12. ACM Press, 2003.

[28] Arun Rodrigues. A comparison of serial simulation frameworks for simulation of computer architectures. Technical Report TR04-32, University of Notre Dame, 2004.

[29] Arun Rodrigues. Enkidu discrete event simulation framework. Technical Report TR04-14, University of Notre Dame, 2004.

[30] Lambert Schaelicke and Mike Parker. Ml-rsim reference manual. Technical Report TR04-10, University of Notre Dame, Notre Dame, Ind., 2002.

[31] Herbert D. Schwetman. Introduction to process-oriented simulation and csim. In *Proceedings of the 22nd conference on Winter simulation*, pages 154–157. IEEE Press, 1990.

[32] Anthony Symons and V. Lakshmi Narasimhan. The design and application of parsim - a message passing computer si mulator, 1997.

[33] Manish Vachharajani, Neil Vachharajani, David A. Penry, Jason Blome, and David I. August. The liberty simulation enviornment, version 1.0. *Performance Evaluation Review: Special Issue on Tools for Architecture Research*, 31(4), March 2004.

[34] Jack E. Veenstra and Robert J. Fowler. Mint: A front end for efficient simulation of shared-memory multiprocessors. In *Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems*, pages 201–207. IEEE Computer Society, 1994.