The Liberty Simulation Environment, Version 1.0

Manish Vachharajani Neil Vachharajani David A. Penry
Jason A. Blome David I. August

Departments of Computer Science and Electrical Engineering
Princeton University
Princeton, NJ 08544
{manishv,nvachhar,dpenry,jblome,august}@princeton.edu

Abstract

High-level hardware modeling via simulation is an essential step in hardware systems design and research. Despite the importance of simulation, current model creation methods are error prone and are unnecessarily time consuming. To address these problems, we have publicly released the Liberty Simulation Environment (LSE), Version 1.0, consisting of a simulator builder and automatic visualizer based on a shared hardware description language. LSE's design was motivated by a careful analysis of the strengths and weaknesses of existing systems. This has resulted in a system in which models are easier to understand, faster to develop, and have performance on par with other systems. LSE is capable of modeling *any* synchronous hardware system. To date, LSE has been used to simulate and convey ideas about a diverse set of complex systems including a chip multiprocessor out-of-order IA-64 machine and a multiprocessor system with detailed device models.

1 Introduction

High-level modeling is a crucial design evaluation tool used by both researchers and designers of hardware systems. Current analytical models, despite having many desirable properties, are only sufficient to provide accurate guidance for special cases. As a result, designers and researchers rely on high-level (e.g. microarchitecture level) software simulation models to evaluate designs [3, 5, 8].

Unfortunately, existing simulator model creation techniques are not well suited for modeling hardware. Models built with existing approaches either are difficult to understand, are hard to validate, are unnecessarily time-consuming to develop, or suffer from some set of these problems [16, 15, 17]. To address these problems, we developed the Liberty Simulation Environment (LSE) after performing a careful analysis of existing systems. This analysis allowed LSE to incorporate the strengths of existing systems while avoiding their shortcomings.

This paper is an overview of the Liberty Simulation Environment, Version 1.0. LSE is a high-level hardware modeling system designed to permit the *rapid*, *accurate* specification of models. Like some existing systems, LSE employs a concurrent-structural modeling methodology. This allows a natural specification of hardware in which the model structure parallels the hardware structure.

In addition to the benefits inherent in the concurrent-structural method of modeling, LSE has several other benefits not found in other concurrent-structural systems. First, LSE models can be constructed more rapidly because LSE utilizes a unique combination of specification language techniques that make reusing and building flexible components practical [15, 17]. This reuse makes development easier and improves collaboration as users can easily leverage another's model components in their own models. Second, LSE provides an abstraction that simplifies the time-consuming and tedious specification of timing control, which often cannot be modeled by reusing components. Third, LSE descriptions are statically analyzable. This enables, for example, simulator construction optimizations to improve simulator execution performance [10] and tools for automatic model visualization [2, 17].

The remainder of this paper is organized as follows. Section 2 motivates LSE by giving an overview of the strengths and weaknesses of existing approaches. Section 3 gives an overview of LSE. Section 4 relays some of our experiences with LSE to date, and Section 5 quantifies the benefits of LSE. Section 6 concludes and gives a glimpse LSE's future.

2 Motivation

Our analysis of existing modeling systems revealed that the lack of model clarity, the lack of high levels of easy reuse within and across models, and restricted domains of applicability prevent *rapid* and *accurate* model creation. This section briefly examines how existing modeling methodologies measure up in terms of model clarity, reuse, and generality. These methodologies can be classified into three groups: architecture description languages, sequential simulators, and concurrent-structural modeling tools.

2.1 Architectural Description Languages

Architecture description languages (ADLs) exist as a method to automate modeling of processors [6, 9, 11]. ADL models are easy to understand as they abstract away many confounding details of modern processors by making assumptions about the hardware target. Unfortunately, these assumptions dramatically reduce the generality of the tools, limiting the class of processors that can be modeled [16, 15].

2.2 Sequential Simulators

The most popular method for constructing simulation models is manually coding a simulator in sequential language such as C or C++[1] [3, 8]. This method is popular because it does not limit what can be modeled and because designers are very familiar with these languages for other reasons. However, familiarity with the language does not necessarily translate to model clarity.

Hardware is designed using a divide and conquer strategy in which designers divide the system's complex functionality into simpler communicating hardware components and design each individually. The final system is built by assembling the individually designed components. To ensure the components will interoperate when reassembled, designers agree on the communication interface for each component. This interface fully encapsulates the functionality of each component. Other parts of the system can change without affecting a particular component provided its communication interface is respected.

Unfortunately, the partitioning of behavior permitted by function calls and function interfaces in sequential languages does not match the partitioning of hardware into components and port interfaces. This means that traditional software components cannot directly correspond to arbitrary hardware components. This, in turn, means that models built by mapping a hardware design to a sequential program will be difficult to understand and validate because they do not resemble the hardware they model. Also, since this mapping process cannot be standardized, reuse of software components is limited. This is evidenced by the incompatibility of simulator components that are independently developed. Even code commonly thought of and provided as a single entity cannot be encapsulated into a reusable component in practice [16, 15, 17]. For example, caches are often provided as a software module but are typically modified to allow coupling to various other parts of the simulator to allow correct modeling of timing [4].

2.3 Concurrent-structural Systems

Concurrent-structural modeling [7] avoids many of the problems experienced by sequential simulator construction even while retaining the same level of generality. Concurrent-structural modeling does not require breaking hardware component encapsulation. Since this encapsulation is preserved, the models resemble hardware and, in general, are clear. Further, since hardware behavior is encapsulated within model components, these components could potentially be reused.

Unfortunately, existing implementations of the approach do not live up to this ideal. Some systems that appear to be concurrent and structural do not eliminate all the problems associated with sequential simulators [16]. The remaining systems preclude component-based reuse in practice. If reusable components are too difficult to build or reuse, users will not actually reuse components [12]. Existing systems force a trade-off between ease of using and ease of constructing reusable components. This trade-off makes components too cumbersome to build and use, precluding reuse in practice. A detailed analysis of these systems is beyond the scope of this paper but more information can be found elsewhere [15, 17].

3 An Overview of LSE

For full model generality and clarity, we chose a concurrent-structural modeling methodology for the Liberty Simulation Environment (LSE). Like other concurrent-structural systems, users build models by instantiating components and specifying their interconnections. However, unlike other high-level concurrent-structural systems, LSE provides *low-overhead* component-based reuse in practice. Additionally, LSE provides a mechanism to simplify timing-control specification further easing hardware modeling tasks.

This section gives an overview of the features found in LSE, Version 1.0 release. These are: component-based reuse, easy timing-control specification, a library of components, and a set of tools made possible by compile-time knowledge of the hardware model structure.

3.1 Component-based Reuse

Component-based reuse is appealing in hardware modeling because many hardware blocks within and across designs share high-level functionality (e.g. queues, arbiters, and switches). However, since most hardware blocks are custom built for each design, they contain variations in their behavior. Thus, to enable component reuse, LSE supports a variety of features to permit the creation and use of *flexible* components that can be customized to match these variations. Furthermore, to ensure that these features are practical, LSE provides several mechanisms to reduce the overhead of building and using these flexible components.

Each component in an LSE specification is instantiated from a template, called a module, that dictates the interface of the component and specifies its behavior. Each module has a set of parameters that can be set by the user. When constructing an instance, these parameters are used to determine an instance's ports (i.e its communication interface) and behavior.

There are two types of modules in LSE, leaf modules and hi-

[1] In the 30th International Symposium on Computer Architecture in 2003, at least 23 of 37 papers used this simulator construction methodology.

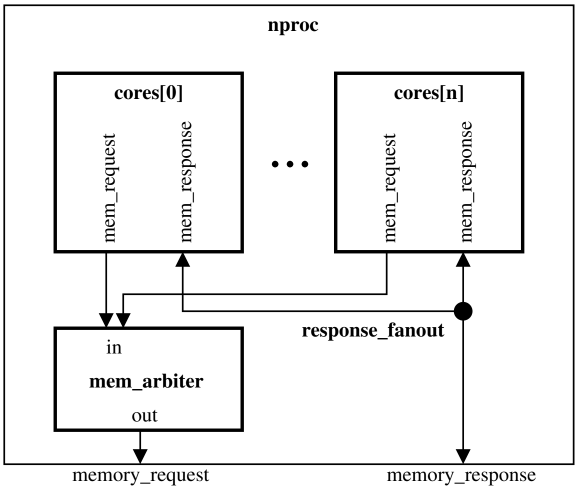

Figure 1: Block diagram of an n-processor module.

```
module nproc {
  parameter n:int;

  outport mem_request:request_t;
  inport  mem_response:response_t;

  var cores:instance ref[];
  cores=new instance[n]("cores", cpucore);

  instance response_fanout:tee;
  instance mem_arbiter:arbiter;

  /* Customization code for the arbiter */
  ...

  var i:int;

  mem_response -> response_fanout.in;
  mem_arbiter.out -> mem_request;
  for(i=0;i<n;i++) {
    cores[i].mem_request->mem_arbiter.in[i];
    response_fanout.out[i] -> cores[i].mem_response;
  }
};
```

Figure 2: LSE module declaration for an n-processor module.

erarchical modules. Leaf modules are basic templates whose instances' behavior is specified via sequential code that utilizes API calls to send and receive data via an instance's ports. Hierarchical modules, on the other hand, obtain their behavior by composing instances of other modules.

LSE supports flexible hierarchical modules by allowing the structure of hierarchical module instances to be specified algorithmically and to be controlled by the values of module parameters, rather than requiring it to be statically and explicitly specified by the user. For example, it is possible to create a module that models a collection of processor cores where the number of cores is controlled by a parameter, n. Note that in LSE, each of these cores would themselves be hierarchical modules.

A block diagram of an n-processor module is shown in Figure 1. Figure 2 shows the LSE code for this module, Figure 3 shows the LSE code to use this module to model 3 proces-

```
instance memory:mem_hierarchy;
instance procs:nproc;

procs.n=3;

procs.mem_request -> memory.request_in;
memory.response -> procs.mem_response;
```

Figure 3: Use of the module in Figure 2 with $n = 3$

sor cores connected to a common memory. Notice how the for-loop algorithmically specifies the structure of the `nproc` instances. The same features used in this example can be used to parametrically control the number of functional units in a processor model, the number of stages in a pipeline, and many other model structures. Note that, although the example shows a counted for-loop, any normal programmatic structure can be used in LSE specifications.

In order to allow scalable communication interfaces such as a register file module with a customizable number of read ports, each port in LSE is actually a variable length array of *port instances*. Rather than connecting two ports together to have two instances communicate, one connects two port instances together. To create a two-ported instance called `rf` of a module named `register_file`, one could type:

```
instance rf:register_file;

decode_stage.reg1 -> rf.read_reg[0];
decode_stage.reg2 -> rf.read_reg[1];
```

The model code in Figure 2 uses these port arrays for fanout with the `tee` module and for the input port of the `mem_arbiter` instance.

In the register file example above, the two connections made from the `decode_stage` module to the `rf` instance automatically sets the number of read ports to 2. The number of connections made to the `read_reg` port (called the width of the port) automatically appears as a parameter to the constructor of the `register_file` module. Thus, the user is not burdened with making two connections and then *redundantly* setting a parameter stating the number of register file read ports (connections). The author of the `register_file` module similarly does not have to worry about declaring and naming extra ports based on a `num_read_ports` parameter, the system provides this automatically. This reduces the overhead of using and building reusable components.

In addition to normal parameters and parameters that control structure, LSE allows modules to be customized via algorithmic parameters, called *userpoints*. Userpoints can contain fragments of code that module instances can call at run-time to implement their behavior. Thus, these algorithmic parameters allow users to extend and override existing modules, much like inheritance and method overriding in object-oriented programming languages allow customization of classes. This feature can be used to customize the arbitration logic in a generic arbiter module or control the cache replacement policy in a cache controller.

LSE also supports a number of other features designed to allow low-overhead use and creation of flexible modules. One such feature is polymorphism. Data types for ports and other entities in LSE can be declared to be polymorphic (i.e. have many types). For example, the `arbiter` module used earlier can arbitrate any LSE data type, though any given instance can only manage one particular type. To reduce the overhead of using polymorphism, the appropriate type for polymorphic entities is determined automatically when possible.

More information on polymorphism, automatic type inference, userpoints, flexible interfaces, and other LSE features for flexible components is available elsewhere [15, 17].

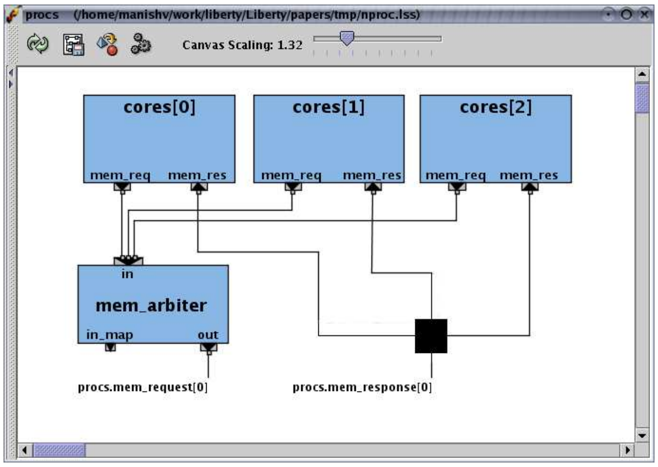

Figure 4: Visualization of `nproc` with $n = 3$.

3.2 Timing-Control

While component-based reuse dramatically improves modeling speed, the timing-control specification of a hardware model typically does not benefit from reuse because of its global nature. In existing concurrent-structural systems, timing control, the portion of the control logic responsible for computing when to stall and distributing stall signals, is often tedious and time-consuming to specify. To address this, LSE provides a timing-control abstraction mechanism that lifts some of the burden from the user making timing-control specification easier.

LSE's timing-control abstraction is enabled by partitioning timing control into two pieces: stall generation and stall distribution. Although timing-control specifications in general do not benefit from reuse, each subtask of timing control does benefit from some amount of reuse.

Reuse in stall generation logic is enabled by recognizing that stalls come in two varieties: semantic stalls and structural stalls. Semantic stalls are those that must occur due to the exact nature of the computation the hardware is performing (e.g. data-hazard stalls in a processor). Structural stalls are ones that usually occur due to local resource limitations (e.g. a pipeline stalls because there are no more slots free in a queue). It is very difficult to automate the detection of semantic stalls in a general way because they are so tightly coupled to the overall microarchitecture. However, reusable components can automatically generate stall signals for many structural stalls.

Stall distribution logic can also benefit from reuse by observing that stalls usually propagate in a very regular way: all hardware earlier in the datapath communicating with a component generating a stall usually stalls as well. For example, in the n-processor example shown in Figure 1 (which has a blocking memory sub-system), if the memory hierarchy stalls, the arbiter and the requesting processor core will usually stall.

LSE leverages these facts to alleviate the burden of specifying much of the timing-control logic. Instances of modules in the LSE module library automatically detect any local structural stalls that they can. Each connection in LSE not only carries data, but provides additional lines to carry stall signals. Module instances automatically propagate stall signals from their outputs to the appropriate inputs, eliminating the need for explicit stall distribution in most cases. Semantic stalls can be injected by custom logic and propagate using these same lines. Of course, users are free to override and customize this behavior as they see fit. More details can be found elsewhere [16, 17].

3.3 Component Libraries

Since LSE supports the creation of flexible reusable components, it is natural that it also has a collection of component libraries for use across many models. Currently, the LSE release comes with a set of core modules useful across many designs. Modules in this library include queues, routers, arbiters, control modules, and other utility modules.

There also exists a collection of components for modeling processor components such as caches, branch predictors, and issue windows. Other groups have contributed module libraries for modeling I/O devices and on chip interconnect networks. The ease of building models in LSE improves as additional flexible modules are released to the public. The National Science Foundation under the Next Generation Software program is supporting work by several groups to do exactly that.

3.4 Exploiting Model Structure

In addition to low-overhead specification, LSE has a number of additional capabilities enabled by analyzing the structure of LSE models. While some existing concurrent-structural tools rely on the dynamic execution of constructed simulators to achieve the flexibility described earlier, the structure of LSE models is known statically. This allows LSE to use the model structure statically to perform type inference (as described above), to infer port widths (also described above), to visualize the model structure, to allow the orthogonal specification of code that collects data during simulation, and to perform static scheduling and optimization of the model when building a simulator.

The LSE visualizer allows users to see a block diagram of their textual specifications, and can also display on the diagram data sent by the data collection code in the running simulator. Figure 4 shows a screenshot of the visualizer displaying the specification of Figure 3. More information about the visualizer as a tool for exploring models statically and as a tool for visualizing system activities during simulation can be found in other work [2, 13].

LSE's static scheduling and optimization engine allow LSE generated simulators to perform better than other concurrent-structural modeling systems that use user-customized modules for speed. Performance numbers and information about LSE's scheduling and optimization can also be found elsewhere [10]. Note that the optimizations currently performed are fairly limited and LSE's performance should be able to surpass that of existing concurrent-structural systems by even larger margins as LSE's analysis engine becomes more sophisticated.

4 Experience with LSE

This section recounts one of the many experiences that serve as anecdotal evidence of the benefits derived from the LSE features described in the prior section. The next section will recount experiments that quantify these benefits.

In June 2003, a number of visitors from a corporation visited the Liberty research group to evaluate LSE. At the end of our demonstration of a processor model, the visitors asked if this processor core could be used in a multiprocessor model since they were interested in modeling front-side bus designs that support multiple processors.

While this model had never been used in a multiprocessor specification before, we were confident that a simple multiprocessor specification should be relatively straight-forward to construct because of LSE's support for encapsulation and flexible components. In the presence of the visitors, we removed module instances for the memory hierarchy from the processor model and wrapped the processor core into its own hierarchical module. Then, using only modules from the core library, we were able to construct a simple bus arbiter and connect multiple core instances to it. After some minor updates so that the cores' load-store units (LSU) could handle losing arbitration of the memory bus, we had a working shared memory multiprocessor model. This entire process took only a few hours in the presence of the visitors. The resulting specification was similar to that shown in Figure 2.

LSE's features played a critical role in the simplicity of creating this model. First, LSE is a concurrent-structural system, and thus the processor core model could easily be instantiated multiple times and expected to behave correctly. Second, modifying the LSU to handle losing arbitration of the databus was simple because any stalls that needed to propagate due to the unavailability of memory occurred automatically. Third, LSE's component customization features made it easy to compose and customize modules from the component library to create a simple bus arbiter.

Users have been able to rapidly construct and modify a number of other complex models in addition to the one described here. In addition to the models described in the next section, these include clustered architecture models, a chip multiprocessor system with detailed device models, and a model of power consumption in on-chip interconnect networks [18].

5 Quantifying LSE's Benefits

This section quantifies the benefits of LSE by summarizing data that measures the the clarity of LSE models, the time taken to build various models, and the amount of reuse seen in LSE models.

5.1 Model Clarity

To measure the clarity of LSE models, we ran an experiment in which users were asked to identify various features of a processor architecture modeled in LSE. As a reference, subjects were asked the same questions about a model written in C. On average subjects were able to answer questions 33% more accurately with LSE [14, 17].

5.2 Modeling Speed

Within our own research group, LSE has been used to model several machines including an IA-64 processor core, a chip multiprocessor model that utilizes that core, two Tomasulo-style machines that execute the DLX instruction set, and a model that is cycle-equivalent to the popular SimpleScalar `sim_outorder.c` [3] sequential simulator. Each model was built by a single student, and took less than 5 weeks to develop. Some models took only a few days to build. For example, the chip multiprocessor version of the IA-64 model took only 1.5 developer-days to produce once the core model was complete. These development times are quite short. By comparison, SimpleScalar represents at least 2.5 developer-years of effort [1].

5.3 Quantity of Component-Based Reuse

In the models described above, there is a considerable amount of component reuse both within and across models. Within each model, modules are used 3-10 times on average. Furthermore, a significant percentage (73-89%) of instances come from modules in the LSE module library. Reuse across models is even more dramatic. Over all specifications taken together, each module is used an average of 20 times with 80% of instances coming from the module library. This significant reuse is a largely a result of LSE's features to reduce specification overhead. Models built before many of the LSE reuse features were available contain the largest number of non-trivial custom modules relative to the models' size [17].

6 Conclusion

The Liberty Simulation Environment (LSE), Version 1.0 is a high-level hardware modeling system designed to enable the *rapid* and *accurate* creation of hardware models. Experience with LSE demonstrates that it successfully meets its design goals. Models developed in LSE are easy to understand and can be automatically visualized to enhance model clarity. Additionally, LSE models are quick to develop and can be automatically compiled to efficient simulators. LSE is currently in use at several universities and is freely available for both commercial and non-commercial users. More information about LSE, future developments, and LSE itself can be found at the LSE web site [13].

Acknowledgments

We thank Kees Vissers, Timothy Kam, Frederica Darema, Ram Rangan, and the rest of the Liberty Research Group for their support throughout the development of LSE. This work has been supported by the National Science Foundation (NGS-0305617) and Intel Corporation. Opinions, findings, conclusions, and recommendations expressed throughout this work are not necessarily the views of the National Science Foundation or Intel Corporation.

References

[1] T. M. Austin. A User's and Hacker's Guide to the SimpleScalar Architectural Toolset (for toolset release 2.0), January 1997.

[2] J. Blome, M. Vachharajani, N. Vachharajani, and D. I. August. The Liberty simulation environment as a pedagogical tool. In *Proceedings of the 2003 Workshop on Computer Architecture Education (WCAE)*, June 2003.

[3] D. Burger and T. M. Austin. The SimpleScalar tool set version 2.0. Technical Report 97-1342, Department of Computer Science, University of Wisconsin-Madison, June 1997.

[4] R. Desikan, D. Burger, and S. W. Keckler. Measuring experimental error in microprocessor simulation. *Proceedings of the 28th International Symposium on Computer Architecture*, July 2001.

[5] J. Emer, P. Ahuja, E. Borch, A. Klauser, C.-K. Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, and T. Juan. Asim: A performance model framework. *IEEE Computer*, 0018-9162:68–76, February 2002.

[6] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *Proceedings of the European Conference on Design, Automation and Test (DATE)*, March 1999.

[7] Open SystemC Initiative (OSCI). *Functional Specification for SystemC 2.0*, 2001. http://www.systemc.org.

[8] V. Pai, P. Ranganathan, and S. Adve. RSIM: An execution-driven simulator for ilp-based shared-memory multiprocessors and uniprocessors. In *Workshop on Computer Architecture Education held in conjunction with HPCA97*, San Antonio, Texas, Feb. 1997.

[9] S. Pees, A. Hoffmann, V. Živojnović, and H. Meyr. LISA – machine description language for cycle-accurate models of programmable DSP architectures. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, pages 933–938, 1999.

[10] D. Penry and D. I. August. Optimizations for a simulator construction system supporting reusable components. In *Proceedings of the 40th Design Automation Conference*, June 2003.

[11] C. Siska. A processor description language supporting retargetable multi-pipeline dsp program development tools. In *Proceedings of the 11th International Symposium on System Synthesis (ISSS)*, Dec. 1998.

[12] S. Swamy, A. Molin, and B. Convot. OO-VHDL Object-Oriented Extensions to VHDL. *IEEE Computer*, October 1995.

[13] The Liberty Research Group. Web site: http://www.liberty-research.org/Software/LSE.

[14] M. Vachharajani and D. I. August. A study of the clarity of functionally and structurally composed high-level simulation models. Technical Report Liberty-04-01, Liberty Research Group, Princeton University, January 2004.

[15] M. Vachharajani, N. Vachharajani, and D. I. August. The Liberty Structural Specification Language: A high-level modeling language for component reuse. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, 2004.

[16] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August. Microarchitectural exploration with Liberty. In *Proceedings of the 35th International Symposium on Microarchitecture*, pages 271–282, November 2002.

[17] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, S. Malik, and D. I. August. The Liberty Simulation Environment: A deliberate approach to high-level system modeling. Technical Report Liberty-04-02, Liberty Research Group, Princeton University, January 2004.

[18] H.-S. Wang, X.-P. Zhu, L.-S. Peh, and S. Malik. Orion: A power-performance simulator for interconnection networks. In *Proceedings of 35th Annual International Symposium on Microarchitecture*, pages 294–305, November 2002.