Achieving Structural and Composable Modeling of Complex Systems

David I. August Sharad Malik Li-Shiuan Peh
Princeton University
Princeton, NJ
{august,sharad,peh}@princeton.edu

Vijay Pai
Rice University
Houston, TX
vijaypai@rice.edu

Abstract

This paper describes a recently-released, structural and composable modeling system called the Liberty Simulation Environment (LSE). LSE automatically constructs simulators from system descriptions that closely resemble the structure of hardware at the chosen level of abstraction. Component-based reuse features allow an extremely diverse range of complex models to be modeled easily from a core set of component libraries. This paper also describes a set of such libraries currently under development. With LSE and these component libraries, students will be able to learn about systems in a more intuitive fashion, researchers will be able to collaborate with each other more easily, and developers will be able to rapidly and meaningfully explore novel design candidates.

1. Motivation and Direction

There is an increasing need to rapidly and accurately model a diverse set of hardware systems. Ideally, in the creation of hardware systems, researchers and developers would build prototypes of each design candidate for evaluation. Prototype building can yield extremely accurate models, and the process of building the prototype itself can be informative. Of course, prototype building is impractical in almost all situations, but practical modeling methodologies that engineers employ should mimic the positive aspects of prototype building as much as possible.

The most prevalent modeling methodology employed today is hand-writing monolithic simulators in sequential programming languages such as C or C++. While writing a simulator in this way, the simulator writer must map systems, which are inherently structural and concurrent, to a sequential programming language with functional composition. Though much more cost effective than prototype construction, this mapping process is still quite laborious, often consuming many person-years of effort. The manual mapping process is also prone to error, and because the resulting simulator code does not resemble the design or operation of actual systems, errors introduced tend to go unnoticed [9, 6, 11]. Further, unlike prototype construction, little understanding of the system is gained during the mapping process.

The manual mapping problem has broader negative effects. These effects are most pronounced in the following three areas:

- **Collaboration.** There exist many correct ways to map a concurrent, structural system to a sequential language. Unfortunately, unless a common mapping scheme can be adopted, the resulting simulators cannot interoperate. Collaboration between and among members of academia and industry often stalls because of this tool incompatibility. Collaboration between domains is hardest hit for lack of common multi-domain solutions.
- **Novel Research.** Radical and disruptive research is often difficult to achieve with the current modeling methodology. Publicly available simulators provide a model only for systems similar to those preconceived by the tool's authors. High risk ideas requiring a new simulator are often discarded because of the potentially enormous cost of failure.
- **Rapid Reuse.** Monolithic simulator tools tend to be "one-off" items often rewritten from scratch for each project. Models of the same single component may be written many times to fit each simulation system used. Researchers and developers should not have to continue paying the price of these unnecessary recurring costs.

These negative effects have been identified and much work has been performed to address them. Some have proposed the creation of standard simulator tool sets upon which extensions could be built [20, 14, 5], but the diversity of needs *requires* that no monolithic simulator standard be adopted. Tools have been created to rapidly produce a simulator, but these tools typically speed the process by making

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

domain-specific assumptions about the system to be modeled [18, 21, 23, 13]. In other cases the tools are too generic and do not provide the necessary mapping constraints to ensure component interoperability [4, 19, 10, 16]. In all these cases, the root cause of these negative effects, the mapping problem, was not directly addressed and one or more of these problems remain [25].

The ideal modeling solution is a system that enables a methodology approximating prototype building. The specification of the model should resemble the hardware itself; it should be structural and concurrent, eliminating the need to map the candidate design to sequential code. Like prototyping, the specification process should force designers and researchers to think about the hardware, not to worry about simulator implementation issues. The specification should involve reusable components at various levels of abstraction. To facilitate collaboration, a domain independent component contract should be used to ensure that components and specifications from any domain can interact without prior planning. The system should also be free of any assumptions that would limit the exploration of radical ideas. We intend to deliver such a system to the modeling community.

Our goal is to research, develop, and disseminate a complete simulation environment that supports a structural and composable modeling methodology. This simulation system consists of the Liberty Simulation Environment (LSE) and an initial set of robust component libraries.

LSE automatically constructs simulators from system descriptions that closely resemble the structure of hardware and component libraries. This structural resemblance to the hardware provides confidence in the model and frees systems researchers to think about systems, not simulator coding concerns. LSE's strict but general component communication contract enables the creation of highly reusable component libraries and eases the task of rapidly exploring ever more exotic designs. LSE components and descriptions can be hierarchically composed of other components and can exist at any level of abstraction (statistical to gate-level). This choice of abstraction level combined with partial specification support allow models to be iteratively refined; descriptions generate fully functional simulators from the very start, allowing users to specify and validate precise models incrementally.

Several component libraries must be provided to serve as a foundation for creating simulators that span multiple architectural levels. LSE components have already been used to model a wide range of microprocessors and interconnection networks. By building on these component libraries, we intend to support a wide range of computational systems including systems-on-a-chip (SoCs), distributed clusters of workstations, tightly-coupled multiprocessors, and high-end supercomputers. While traditional simulators have focused on general-purpose programmable processors, LSE will allow the composition of complex heterogeneous system models that include both programmable components and dedicated application-specific hardware models for tasks such as wireless communication or high-speed network I/O. Such composability becomes essential as systems of interest evolve from commodity PCs to sensor network arrays and distributed low-power embedded systems.

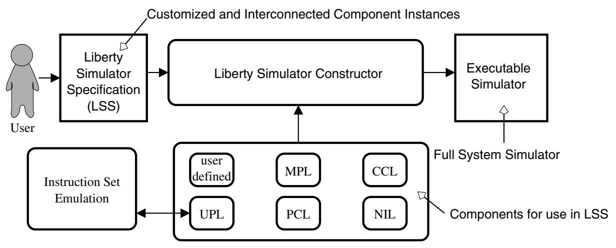

Customized and Interconnected Component Instances
User
Liberty Simulator Specification (LSS)
Liberty Simulator Constructor
Executable Simulator
Full System Simulator
Instruction Set Emulation
user defined
MPL
CCL
UPL
PCL
NIL
Components for use in LSS

Figure 1. An overview of LSE

By providing a design-neutral, unrestricted open-source simulation framework to the community, we intend to improve the quality of best-known techniques. Through structural and composable model specification, LSE will allow researchers to easily collaborate, exchange ideas, understand novel techniques, and evaluate the work of others in a variety of contexts, facilitating independent verification of research. The resemblance of LSE descriptions to real systems will allow it to be an effective educational tool when integrated with an interactive system visualizer. The development of several additional reusable core component libraries will provide a starting point for exploration by other researchers, and the "Liberation" of existing popular simulation systems, through encapsulation into LSE modules or through equivalent configuration, will allow a smooth transition for interested researchers.

2 The Liberty Simulation Environment

A user of the Liberty Simulation Environment (LSE) writes a Liberty Simulator Specification (LSS) to specify the desired system by defining interconnections between customized instances of reusable module templates. LSE reads the LSS, instantiates modules templates into module instances, and weaves the specification and module instances together to form an executable simulator. An overview of the Liberty simulator construction process is shown in Figure 1.

LSE makes no assumptions about the target system while ensuring that components interoperate. This guarantees that components developed for one domain can be combined with components developed independently for another. In this way, LSE can serve a single platform for developing a limitless range of full system simulators. LSE also contains features that allow for iterative refinement of designs, a variety of abstraction levels, and reuse of the components for reduced simulator construction time.

2.1 Reusable Components

LSE was developed with a specification language and component system based on observations about the shortfalls of existing systems. Many simulation systems, in particular those for processor design [20, 5], are built upon sequential programming languages, such as C or C++, and attempt to leverage traditional software componentization and composition techniques to allow for hardware component reuse. Unfortunately, this approach does not work well, since traditional software composition techniques are designed for components that execute sequentially in LIFO (program stack) order, typically waiting for all inputs before beginning computation. As a result, the timing, control, and functionality of components cannot be abstracted in a way that would allow them to be reused across a large variety of systems [25]. This lack of reuse makes designing a system model extremely cumbersome. Furthermore, since many of these systems do not have clean component interfaces with respect to hardware blocks, it becomes difficult to refine a coarse model to a more accurate one by replacing high-level models with more detailed ones.

In contrast, like real hardware, each LSE module instance executes concurrently with other LSE module instances. Modules specify their interface to other modules via ports. Each port represents an input or output channel for the module, and may have multiple connections so that users can easily scale the bandwidth a module instance has to the other blocks. Each module instance is abstracted solely by its communication interface, with no assumptions about sequentiality of the internal computation. Because of this decomposition, the LSE specification resembles the hardware that is being designed. As a result, it is far easier and less error prone to translate ideas about hardware to an LSS.

LSE module templates encapsulate functionality with a flexible control interface. Each connection in LSE actually corresponds to a connection of 3 signals. These 3 signals are used to negotiate whether or not data can be transmitted across a connection in a particular time-step. The signals are similar to those used in bus handshaking protocols, and serve a similar purpose, to guarantee that two components can interoperate even if they weren't explicitly designed to interoperate with each other. Within a set of simple rules, the user can manipulate how the "handshake" signals are used to specify any control behavior they desire, independent of module functionality. To prevent the user from having to specify full control semantics, module templates provide default control semantics. Using the default control semantics, working system models can be constructed by connecting the datapath and specifying minimal control. LSE allows the user to override the default control semantics so that any system behavior can be specified.

In addition to control semantics, details regarding the functionality of hardware blocks often vary from system to system. However, in monolithic simulators, these small variations often require extensive code modifications since functionality, timing, and control are intertwined in the specification. Since these changes become overwhelming, simulators are often written from scratch for each new project. To allow components to be reused, despite small changes or extensions to functionality, LSE also has a powerful component customization capabilities. Components have algorithmic parameters, parameters whose values describe functionality. Via these parameters, users can inherit the overall functionality of a module template, but adapt the specific behavior to the system being modeled.

To further improve reuse, LSE allows users to build new module templates based on the interconnection and customization of instances of existing module templates. To make the resulting *hierarchical* module template flexible, the LSS language has powerful syntax by which users can specify the hierarchical module template's parameters and port connections relative to the sub-module-instances' interconnections and customizations.

By providing hierarchical structural composition, customization of components, and a communication contract with default control semantics, LSE allows construction of module template that can be reused in many contexts. For example, a single module template can be instantiated to model a processor's instruction window, its reorder buffer, and the I/O buffers in a packet router [25, 26].

2.2 Levels of Abstraction and Iterative refinement

One of LSE's great strengths is the ability of LSE module instances to interoperate. Even module instances with different levels of abstraction can interoperate. As a result, it is possible to mix components with different levels of detail in the same LSS. For example, a model of an interconnect network may have connected to it a statistical packet generator used to simulate network traffic. However, it is possible to replace the statistical packet generator with a network interface controller for a microprocessor simply by replacing the packet generator. In this way, the same interconnect model can be used with an abstract statistical model, as well as a detailed microprocessor model.

Full system abstraction is also possible with LSE. Each module template can provide default semantics when some of its ports are left unconnected. This means that users can specify a partial system and rely on the default behavior to fill in omitted details, thus forming an abstract model of the entire system. We use this feature extensively while building processor microarchitectural models. The typical design process starts by first specifying simple fetch and issue logic. Then, once satisfied with this behavior, we add a pipeline specification, speculation control logic, predictors, and memory hierarchies in turn. At each stage in this refinement process, the specification is compilable into a working simulator.

2.3 Relation to Other System Specification Approaches

Other systems do not provide a domain independent modeling system with a component contract that allows for reusable component libraries. Hardware description languages, while capable of full system description, do not provide enough support for component customization to build reusable components. Thus, a user has no ability to affect the internal timing of pre-built components. Furthermore, HDLs still require the user to specify all control explicitly, making specification of new systems quite involved. Since the control interface is ad-hoc, pre-existing components may not be able to interact in the desired way if the correct control signals were not exposed.

SystemC [19] is superior to HDLs since it has better support for inheritance and a more advanced type-system allowing better abstraction. However, SystemC does not provide any mechanism for simplifying the specification of control, or providing default control semantics. As a result, specifying a model in SystemC can still be difficult. In addition, since the SystemC libraries do not provide a component communication contract, the control interface is, once again, ad-hoc. Thus, the same mapping problem exists for SystemC, making the standardization upon a standard component library unlikely. Note, however, that LSE could bring its benefits to SystemC to solve these problems by wrapping it. This is an option being explored.

While not exclusively a hardware modeling tool, the Ptolemy framework does allow users to model hardware by composing concurrently executing components [4]. However, unlike LSE, SystemC, and HDLs, Ptolemy allows each model to specify the model of computation (MoC) that governs the semantics of communication and execution. This flexibility, however, comes at a price. When using different models of computation, work must be done to ensure that the models can be composed. Sometimes this process is easy and automatic, at other times it may require solving difficult problems [17]. Techniques allow some MoCs to interact [15, 12], but they do not cover all MoCs leaving the possibility of incompatible components. Furthermore, for hardware modeling, this flexibility is unnecessary since most hardware can easily be specified using a single model of computation with little loss of clarity or specification ease. Also, MoC flexibility also comes at a simulation performance price too steep for applications of interest.

LSE fixes its MoC to a reactive model of computation. This has several advantages. First, power users only have to learn one set of computation and communication semantics easing the learning curve (though most users need not be concerned with these details). Second, since all components use the same model of computation, the MoC does not preclude reuse of components. It should be noted, however, that the standard MoC must be carefully chosen to avoid interfering with reusability [25]. Third, by carefully selecting the model of computation it is possible to analyze the LSS for optimization [22].

Other domain specific approaches for simulator construction have been proposed. These approaches [18, 21, 23, 13] gain most of their benefit from domain specific assumptions and thus are not suitable for general system-level simulation. Other approaches have been proposed [10, 16] that could extend to full system simulation. However, these approaches lack the necessary features to allow construction of interoperable components and highly reusable component libraries.

Perhaps the most important shortfall of many of these systems is availability. Many of these systems are proprietary or not publicly available. Through the support of the National Science Foundation's Next Generation Software program, we have been able to release LSE Version 1.0 without restriction ensuring that it will remain an open, standardized collaborative framework.

3 Component Libraries

The core of LSE is its libraries of components consisting of LSE modules. These pre-defined modules enable rapid modeling of complex systems through seamless composition. We classify the various components into the following libraries based on a functional partition:

Primitive Component Library (PCL) This consists of primitive building blocks that are likely to be used across a wide range of applications. Examples include arbiters and memory arrays.

Uni-processor Library (UPL) This consists of the micro-architectural elements of general purpose and application specific processors. Examples include instruction decoders and branch prediction units.

Communication Component Library (CCL) This consists of building blocks of communication fabrics. Examples include buses and routers.

Network Interface Library (NIL) This consists of components that serve as interfaces across network boundaries and in between networks and processors. As example is a format converter that sits between an Ethernet and a PCI bus.

Multi-processor Library (MPL) This consists of components used in multi-processor architectures. Examples include cache-coherence engines for implementing shared memory systems and DMA controllers for implementing message passing.

The above classification is a functional one. It is driven by the need for organization of what would otherwise be a single vast and difficult to manage library. This classification helps in both the library development stage as well as the library deployment stage. During library development, it is the natural partition of tasks among specialists in the various functional domains. During deployment, it serves as a catalog to help search for the appropriate match in the building of complex systems.

It should be noted that this classification does not create any usage boundaries. Components in one library can freely be used in other libraries. The primitives in PCL are likely to be used in all the other libraries. Similarly, MPL is likely to build on all the other libraries in constructing top-level models of complex multi-processor systems.

We now illustrate the use of these component libraries in assembling models for a diverse range of systems. Figure 3 sketches several systems that our proposed simulation framework will support. Each system can be composed in a plug-and-play fashion using the LSE modules defined in our suite of component libraries. By defining all these modules with LSE, modules can inter-operate seamlessly across component libraries. Thus, simulation of new complex systems can leverage the modules in these component libraries for substantial productivity gains. For instance, a chip multi-processor (see Figure 2(a)) will consist of general-purpose processor (GP) modules from UPL, interface modules (NI) from NIL, and network fabric modules provided by CCL, glued with multiprocessor modules from MPL.

Many of these libraries will share modules with similar semantics, with components carefully defined for reuse. For instance, in a sensor network node (see Figure 2(b)), which is composed of a general-purpose processor (GP) and a digital signal processor (DSP) from UPL, linked with a bus from CCL, and interfacing to a wireless radio component from CCL through a radio interface from NIL, the GP and DSP will share many common modules within UPL. The various libraries will also share many modules of PCL, such as the memory array primitive component which can double as bus queuing buffers for CCL as well as caches in UPL.

Our goal of reusing the components led to careful generalization of modules, so the same module can be parameterized and plugged into substantially different systems. Figure 2(c) demonstrates how similar modules used to simulate a chip multiprocessor can now be extended to simulate systems of a totally different scale - a petaflops multiprocessor grid-in-a-box, with many GP modules from UPL, sophisticated network interface controllers from NIL, interconnected with high-speed electrical or optical fabrics from CCL, and glued with MPL modules such as cache coherence controllers.

The hierarchical and iterative refinement features of LSE are especially critical when we consider complex systems-of-systems such as that in Figure 2(d). Here, we envision small sensor nodes peppered around an area, collecting and communicating data wirelessly back to coarser-grain nodes with chip multiprocessors that analyze and coordinate groups of sensors. Finally, analyzed data is aggregated back to a base camp where there are petaflops grids-in-a-box that performs computationally intensive tasks for coordinating and controlling the nodes in the field. With LSE, we can compose such a complex system hierarchically from the subsystems built with components of the various libraries. It also allows users to work at different levels of abstraction, so a network architect can iteratively define the wireless network component of CCL, perform detailed studies, while keeping the rest of the system at a high level of abstraction.

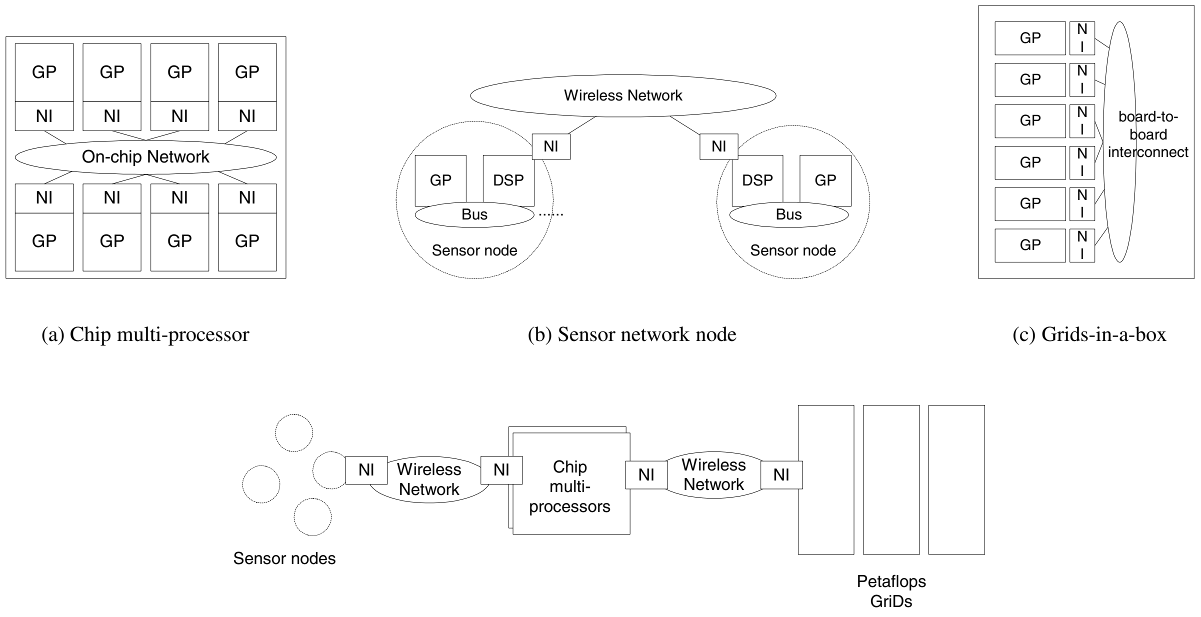
We will next detail the various component libraries, the challenges faced in each, and discuss the current status and future work.

3.1 Primitive Component Library (PCL)

As we built early versions of UPL and CCL, we found clear building blocks that are common across many libraries such as queues and arbiters. These primitives can be readily leveraged while building the functional component libraries, saving development time, maximizing reuse, and easing debugging. An arbiter is an example of a primitive that is readily used across various component libraries. For instance, the same arbiter module can be used in CCL to control access to network buffers and links, and in UPL to regulate access to synchronization locks. The PCL has been released with the support of the National Science Foundation along with LSE Version 1.0.

3.2 Uniprocessor Library (UPL)

The Uniprocessor Library (UPL) contains all the building blocks for standard microprocessor models. UPL includes basic buffering and queuing structures that can be customized to model the main processor pipeline including functional units, re-order buffers, instruction windows, and the corresponding interconnections. It also includes many modules that when composed hierarchically, can provide complex components such as realistic cache configurations.



(a) Chip multi-processor

(b) Sensor network node

(c) Grids-in-a-box

(d) A complex system of systems

Figure 2. A diverse range of systems that LSE aims to support through a suite of component libraries.

We have built an extensive range of components in UPL, and have successfully modeled IA-64 and Alpha processors. These models are being prepared for release along with the UPL.

3.3 Communication Component Library (CCL)

Systems are becoming increasingly interconnected – A simulation infrastructure that supports diverse communication fabrics is critically needed. Orion [26], a CCL, was proposed to address this need, targeting the communication components of a wide array of systems, ranging from on-chip networks in chip multi-processors, to electrical and optical chip-to-chip and board-to-board fabrics in petaflops grids-in-a-box, to wireless fabrics in sensor networks.

The challenges of Orion lie mainly in three areas: modeling of traffic workloads, development of component building blocks that are generalizable to different domains, and component attribute models that cover key design parameters in diverse applications.

An early version of Orion was developed, focusing on wired interconnection networks, supporting fabrics ranging from on-chip buses and networks for SoCs to chip-to-chip electrical backplanes for petaflop grids [26]. Now, in addition to dynamic power, Orion characterizes leakage power [7] as well as the thermal impact of networks. Besides being used to model networks in multiprocessor systems, both on-chip and chip-to-chip, the basic components of Orion have been found to be applicable to interconnected distributed caches [3], as well as heterogeneous multi-core SoCs [27], demonstrating its generality. We have also proposed and investigated various abstractions of different traffic patterns in mobile sensor networks. We plan to further extend Orion to a wider variety of communication fabrics, developing new attribute models for the design metrics that are significant in diverse application domains.

3.4 Multiprocessor Component Library (MPL)

Multiprocessor architectures form one of the most difficult and important simulation domains for high-performance systems engineering. Modules from PCL and CCL form the foundation of a multiprocessor system simulator. The additional complexities in multiprocessor system simulation stem from managing data replication, ordering, and communication. The MPL includes the modular components required for implementing a structural specification of a multiprocessor. These modules include DMA controllers (for simulating low-overhead message-passing systems), pluggable cache coherence controllers (including bus-based snooping for small scale multiprocessors and point-to-point coherence transactions for scalable systems), and pluggable memory ordering controllers to restrict the reordering allowed by the processor according to desired constraints. Many of these components are sufficiently flexible that they can be deployed in a variety of contexts and systems, ranging from chip multiprocessors to tightly-coupled systems to message-passing grids.

A hierarchical and component-based multiprocessor simulator based on LSE serves as an ideal platform for research into multiprocessor simulation methodologies. For example, investigation of speed-enhancing techniques is essential because of the need to support large-scale applications. Here, the reuse enabled by LSE eases the development of sampling versions of hierarchical modules, in addition to allowing the incorporation and study of acceleration techniques developed for the PCL. The support for multiple levels of abstraction in LSE also allows for simulation acceleration by integrating a detailed simulator of some portions with analytical representations of other system components. Such abstraction may increase the applicability of workload-driven analytical models proposed for multiprocessor performance evaluation [24].

We are currently porting the RSIM simulator released by Pai et al. to LSE as a base platform for developing component-based multiprocessor simulators. RSIM's modular open-source design has enabled external users to add substantial functionality; our approach here is instead to break the simulator into Liberty components along the lines of its current modules and to formalize the interactions between modules according to the Liberty model of computation. Our prior experience with porting SimpleScalar to Liberty should help guide our development efforts in these regards.

3.5 Network Interface Component Library (NIL)

Network interfaces bridge processors and fabrics, and multiple networks, and are realized both in ASICs and more recently, as programmable network interfaces. These devices translate between the formats understood on the external network and the local interconnect; the most common realization is a network interface card (NIC) that translates between Ethernet and PCI formats, implementing the arbitration policies of the PCI bus and the medium access policies of Ethernet. Network interfaces have stringent space and power requirements, combined with little data reuse. Consequently, the primary techniques used in general purpose processors to improve performance – large caches and faster clocks – are inapplicable. In contrast, multiprocessor microarchitectures are particularly appealing for these programmable network interfaces, since parallelism can allow for performance without increasing clock speed.

Additionally, these devices have a heterogeneous set of components, including DMA and MAC assist logic. The process of simulating these devices is further complicated by the asynchronous nature of the I/O interactions with both the network and the host. Some previous studies have attempted performance simulation for these devices using conventional processor simulators [8]. However, no work to date has provided a realistic simulation that accounts for the special hardware features or software tasks supported by these systems.

We are currently developing a network interface simulator, with an initial target of properly modeling the MIPS-based Tigon-2 programmable network interface chipset at a level of detail sufficient to simulate the firmware that supports its deployment as a Gigabit Ethernet interface [1, 2]. This simulator development consists of two parallel tracks. One track focuses on bringing up a uniprocessor sufficient to run the desired firmware, adding support for the various hardware assists and memory-mapped registers needed and collecting the I/O traces of host and network traffic that will later drive the simulation. The second track leverages the MPL to implement a scalable parallel programmable network interface and also works toward more aggressively parallelizing the target code. Such simulation efforts will both allow the architectural exploration needed to reach next-generation Ethernet speeds and facilitate the development of realistic models of other I/O devices. Clearly, development of the programmable network interface in NIL will leverage on modules of UPL and MPL.

4 Conclusions

Unlike traditional simulators, the Liberty Simulation Environment (LSE) automatically constructs simulators from system descriptions that closely resemble the structure of hardware. The well-defined component communication interfaces of LSE allow for the reuse and hierarchical configuration of components across systems, easing the exploration of a complex and diverse set of systems. The LSE system has been released. To get the most out of LSE, component libraries are to be released. These libraries target domains including microprocessors, multiprocessors, networks, and programmable network interfaces, using both domain-independent and domain-specific modules.

By enabling varying levels of abstraction and a unified component connection framework, LSE provides both ideal support for education and for technology transfer. Students using LSE will be able to focus on system design concepts and structural composition rather than the syntactic and functional composition of more common simulation schemes. Researchers may more easily release their modules built with LSE both for collaboration in academia and for technology transfer to industry, since the well-defined interfaces of LSE enable these modules to be composed into other LSE configurations with ease.

Acknowledgments

We thank Kees Vissers, Timothy Kam, and Frederica Darema for their comments at various points throughout the development of LSE. This work has been supported by the National Science Foundation (NGS-0305617). Opinions, findings, conclusions, and recommendations expressed throughout this work are not necessarily the views of the National Science Foundation.

References

[1] Alteon Networks. *Tigon/PCI Ethernet Controller*, Aug. 1997. Revision 1.04.

[2] Alteon WebSystems. *Gigabit Ethernet/PCI Network Interface Card: Host/NIC Software Interface Definition*, July 1999. Revision 12.4.13.

[3] B. M. Beckmann and D. Wood. Transmission line caches. In *Proceedings of the International Symposium on Microarchitecture*, pages 43–55, December 2003.

[4] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *International Journal in Computer Simulation*, 4:155–182, 1994.

[5] 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.

[6] H. W. Cain, K. M. Lepak, B. A. Schwartz, and M. H. Lipasti. Precise and accurate processor simulation. In *Proceedings of the Fifth Workshop on Computer Architecture Evaluation using Commercial Workloads*, February 2002.

[7] X. Chen and L.-S. Peh. Leakage power modeling and optimization of interconnection networks. In *Proceedings of the International Symposium on Low Power and Energy Design*, pages 90–95, August 2003.

[8] P. Crowley, M. Fiuczynski, J.-L. Baer, and B. Bershad. Characterizing Processor Architectures for Programmable Network Interfaces. In *Proceedings of the 14th International Conference on Supercomputing*, pages 54–65, May 2000.

[9] 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.

[10] 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.

[11] J. Gibson, R. Kunz, D. Ofelt, M. Horowitz, J. Hennessy, and M. Heinrich. FLASH vs. (simulated) FLASH: Closing the simulation loop. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 49–58, November 2000.

[12] A. Girault, B. Lee, and E. A. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 18(6):742–760, June 1999.

[13] 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.

[14] C. J. Hughes, V. S. Pai, P. Ranganathan, and S. V. Adve. Rsim: Simulating Shared-Memory Multiprocessors with ILP Processors. *IEEE Computer*, 35(2):40–49, Feb. 2002.

[15] E. A. Lee and A. Sangiovanni-Vincentelli. Comparing models of computation. In *Proceedings of ICCAD*, November 1996.

[16] P. Mishra, N. Dutt, and A. Nicolau. Functional abstraction driven design space exploration of heterogeneous programmable architectures. In *Proceedings of the International Symposium on System Synthesis (ISSS)*, pages 256–261, October 2001.

[17] P. Murthy. Modeling and design of reactive systems, 1997. Presentation. URL: http://ptolemy.eecs.berkeley.edu/presentations/97/rasspfinal.pdf.

[18] S. Önder and R. Gupta. Automatic generation of microarchitecture simulators. In *Proceedings of the IEEE International Conference on Computer Languages*, pages 80–89, May 1998.

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

[20] V. S. Pai, P. Ranganathan, and S. V. Adve. *RSIM Reference Manual, Version 1.0.* Electrical and Computer Engineering Department, Rice University, Aug. 1997. Technical Report 9705.

[21] 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.

[22] 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.

[23] 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.

[24] D. J. Sorin, V. S. Pai, S. V. Adve, M. K. Vernon, and D. A. Wood. Analytic Evaluation of Shared-Memory Systems with ILP processors. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 380–391, June 1998.

[25] 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.

[26] H.-S. Wang, X. Zhu, L.-S. Peh, and S. Malik. Orion: A Power-Performance Simulator of Interconnection Networks. In *Proceedings of the 35th International Symposium on Microarchitecture*, November 2002.

[27] T. T. Ye, L. Benini, and G. D. Micheli. Packetized on-chip interconnect communication analysis for mpsoc. In *Proceedings of Design Automation and Test in Europe*, pages 344–349, March 2003.