International Journal of Parallel Programming, Vol. 33, Nos. 2/3, June 2005 (© 2005)
DOI: 10.1007/s10766-005-3569-3

Achieving Structural and Composable Modeling of Complex Systems

David I. August,[1,6] Sharad Malik,[2] Li-Shiuan Peh,[2] Vijay Pai,[3] Manish Vachharajani,[4] and Paul Willmann[5]

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 built easily from a core set of component libraries. This paper also describes the makeup and initial experience with a set of such libraries currently undergoing refinement. With LSE and these soon-to-be-released 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.

KEY WORDS: Structural simulation; liberty simulation environment.

[1]Department of Computer Science, Princeton University, 35 Olden St., Princeton, NJ 08544, USA. E-mail: august@princeton.edu
[2]Department of Electrical Engineering, Engineering Quadrangle, Princeton University, Princeton, NJ 08544, USA. E-mail: {sharad, peh}@princeton.edu
[3]Purdue University, School of Electrical and Computer Engineering, Electrical Engineering Building, 465 Northwestern Avenue, West Lafayette, IN 47907-2035, USA. E-mail: vpai@purdue.edu
[4]Department of Electrical and Computer Engineering, University of Colorado, 425 UCB, Boulder, CO 80309-0425, USA. E-mail: manishv@colorado.edu
[5]Department of Electrical and Computer Engineering, MS 366, Rice University, 6100 Main Street, Houston, TX 77005, USA. E-mail: willmann@rice.edu
[6]To whom correspondence should be addressed.

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 handwriting 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.[1–3] 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,[4–6] 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 domain-specific assumptions about the system to be modeled.[7–10] In other cases the tools are too generic and, thus, do not provide the necessary mapping constraints to ensure component interoperability and encourage reuse.[11–14] In almost 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.[15] Of the remaining systems, none provide the necessary mechanisms needed for enable reuse in practice, not just in principle.[16]

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. In addition, the system should leverage the advantages of high-level modeling. The specification should encourage interfaces and flexibility that enable component reuse at various levels of abstraction. One essential such interface is a domain independent control interface that enables reuse and interoperability of otherwise domain and design specific logic. This allows components and specifications different domains to interact easily, even in the absence of prior planning. Note, however, that these features that promote reuse should also be free of any assumptions that would limit the exploration of radical ideas.

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

LSE automatically synthesizes 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 encourages and enables the creation of highly reusable component libraries. It also 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.

Though LSE has already been released, the complete system must also provide several component libraries. These libraries will serve as a foundation for creating simulators that span multiple architectural levels. LSE's existing components have already been used to model a wide range of microprocessors and interconnection networks. We plan to expand these component libraries to support an even wider 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.

By providing a design-neutral, unrestricted open-source simulation framework to the community, we intend to improve the quality of best-known techniques. Through a common structural and composable model specification language, 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 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 module 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 Fig. 1.

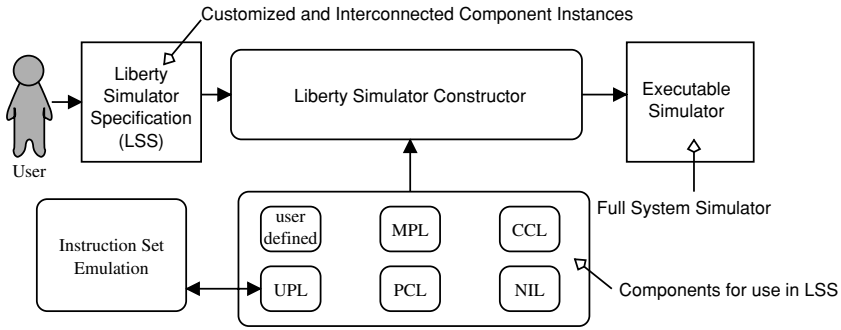

Fig. 1. An overview of LSE.

LSE makes no assumptions about the target system but provides a mechanism to abstract domain-specific control into reusable chunks. This means that components developed for one domain can easily be combined with components developed independently for another. Often this combination is simply a matter of connecting the corresponding port interfaces, with not special translation of control signals required. As a result, LSE can serve as a single platform for developing a limitless range of full system simulators. To further support rapid construction of detailed simulators, LSE contains features that allow for iterative refinement of designs, a variety of abstraction levels, and reuse of the components.

2.1. Reusable Components

LSE was developed with a specification language motivated by the shortfalls of existing systems. Many simulation systems, in particular those for processor design,[4, 6] are built upon sequential programming languages, such as C or C++. These systems attempt to leverage traditional software modularization and component composition techniques to allow for hardware component reuse. Unfortunately, when modeling concurrently executing structurally composed blocks in this way, individual component implementation and global system structure and communication become intertwined. This occurs because all computation must be serialized and encapsulated into function or method calls. This partitioning does not mimic the component encapsulation in hardware, but is instead constrained by the order in which signals are generated. Changing the global structure of the model by changing the interconnection of even a few components changes when signals are generated and often violates prior ordering constraints. Thus, much of the old simulators components cannot be reused, artificially limiting reuse. 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. Since this decomposition parallels that of hardware, LSE specifications are far easier and less error prone to specify. Furthermore, since the component decomposition parallels that of hardware, reuse is not artificially restricted.

To further facilitate easy reuse, LSE module templates encapsulate functionality with a flexible control interface that allows modularization of complex timing-control (i.e., the portion of the controller that determines when and what to stall). 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. These signals are similar to those used in asynchronous bus handshaking protocols, and serve a similar purpose, to guarantee that two components can interoperate even if they were not 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 desired control behavior, 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.[16]

In addition to control semantics, the specific functionality of hardware blocks will 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. These changes often become overwhelming, and, as a consequence, simulators are generally 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, while easily adapting the specific module instance 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 a powerful syntax through which users can specify the hierarchical module template's sub-module instantiations, interconnections, and customizations relative to template parameters and port connections. To lower the overhead of using these components, the LSS compiler will infer parameter values for these components based on their usage context.

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 route.[15, 17]

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 that can execute full programs.

2.3. Relation to Other System Specification Approaches

LSE is the only system of which we are aware that provides a domain independent modeling system with a component contract that allows for

reusable component libraries. The class of systems referred to as hardware description languages (HDLs), 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 without extensive work in build a controller or control adapter components. Worse still, the desired control logic may not be possible to implement if the correct control signals were not exposed.

SystemC[12] 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 modularizing control specifications. Thus, components in a SystemC library likely use an *ad-hoc* control interface based on assumptions about a global control paradigm. Thus, the same component interoperability problem exists for SystemC, making the standardization upon a single 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.[11] 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.[18] Techniques allow some MoCs to interact,[19,20] 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 many 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. (Note that a common MoC is not sufficient for reuse; the MoC must be carefully chosen to avoid

interfering with reusability[15]). Third, by carefully selecting the model of computation it is possible to analyze the LSS for optimization.[21] We intend to use this power to automatically synthesize multi-threaded simulators for execution on distributed memory multi-processor systems. Further work will also enable some of these threads to be implemented in FPGA hardware to further accelerate simulation speed.

Other domain specific approaches for simulator construction have been proposed. These approaches[7–10] gain most of their benefit from domain specific assumptions and thus are not suitable for general system-level simulation. Other approaches have been proposed[13, 14] 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.[16]

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 collection 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.

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.

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.

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 be used freely 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 2 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 Fig. 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 Fig. 2(b)), which is composed of a 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 multi-processor grid-in-a-box, with many GP modules from UPL, sophisticated network interface controllers from NIL, interconnected with high-speed electrical or optical

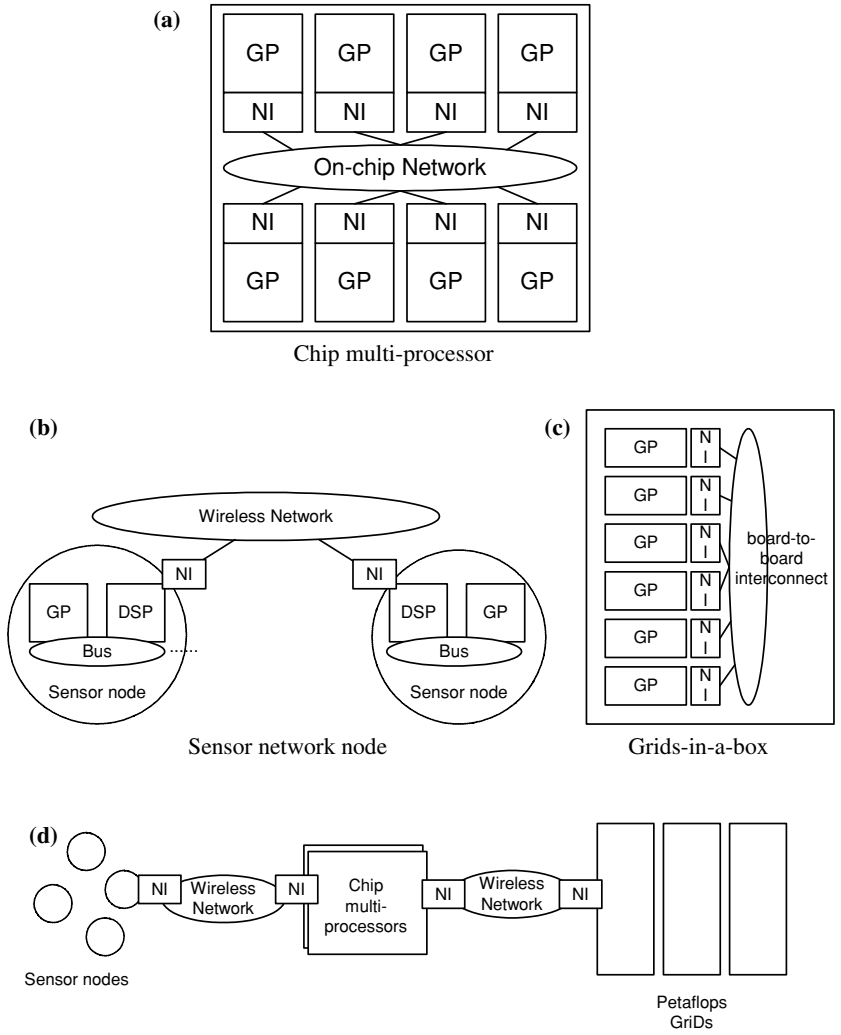

Fig. 2. A diverse range of systems that LSE aims to support through a suite of component libraries.

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

Fig. 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.

The PCL is the most mature of the libraries, with the UPL following closely behind. An extensive range of components has been built for the UPL and the library is being prepared for release. Using the UPL and PCL several processor models have been rapidly constructed, a subset of which is shown in Table I. Table II shows some statistics that quantify the size and reuse seen in each of the models. Each of these models was built

in 2 to 11 person-weeks and, as shown in the table, demonstrate considerable component reuse. When trivial hierarchical modules that exist solely for code clarity are neglected (shown in parentheses), even more reuse is seen.

The Itanium 2 model demonstrates the effectiveness of the structural approach and of the Uniprocessor Library. This Itanium 2 model was constructed and validated by a single student in only 11 weeks. Care was taken to validate the model against the real hardware, using a combination of custom benchmarks, SPEC benchmarks, and the hardware performance counters. Figure 3 shows the performance predictions of this model alongside the performance of the same applications measured on the actual hardware. As we can see, the model predicts the performance of the Itanium 2 hardware to within 3%, despite only 11 weeks of development effort.

Table I. Several Models Developed Using LSE

Model Name	Model Description
A	A Tomasulo Style machine for the DLX instruction set.
B	Same as A, but with a single issue window.
C	A model equivalent to the SimpleScalar simulator.[6]
D	An out-of-order processor core for IA-64.
E	Two of the cores from D sharing a cache hierarchy.
F	A validated Itanium 2 processor model.

Table II. Quantity of Component-based Reuse for Several Models

Model name	Instances	Hierarchical modules	Leaf modules	Instances per module	Instances from library	Modules from library
A	277	46(10)	18	4.33(8.61)	73%	13
B	281	46(11)	18	4.39(8.48)	73%	13
C	62	1	18	3.37	73%	10
D	192	4	25	6.62	86%	22
E	329	4	26	10.97	89%	22
F	183	18(3)	19	4.95(8.32)	82%	18
Total	1324	69(19)	39	12.26(2283)	80%	22

Model descriptions are in Table I.
Values in parenthesis discount trivial hierarchical modules used simply to wrap a collection of components.

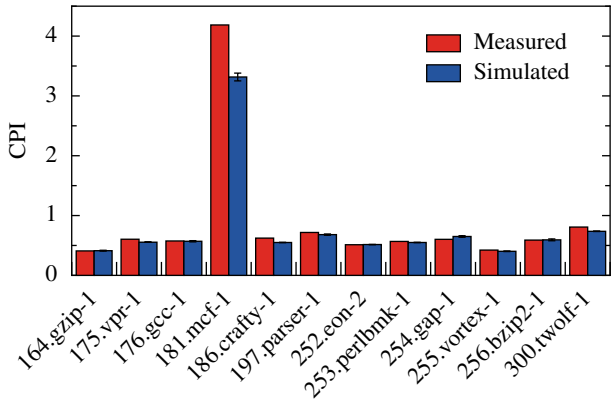

Fig. 3. Performance of actual Itanium 2 hardware versus model predictions.

3.3. Communication Component Library (CCL)

As system interconnect technology becomes increasingly important, a simulation infrastructure that supports diverse communication fabrics is critical. Orion,[17] 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. We will discuss each in turn.

Traffic workload modeling. We explored ways of modeling traffic workloads in on-chip networks, targeted for high-level power analysis of networks. Traffic flows are modeled as step-wise injection-rate functions that can range from cycle-level to segments that are thousands of cycles. By doing so, network contention can be viewed as overflow area in such functions, with buffering reflected as propagation of the overflow area, thus facilitating fast derivation of link utilization at each point in time, and estimation of network power across space and time.[22] Such high-level workload modeling, coupled with the power analysis framework, enables multi-granularity power simulation, so network power simulation can leverage the multi-granularity simulation feature of LSE.

Generalizable building blocks. 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.[17] Besides being used to model on-chip and chip-to-chip networks in multiprocessor systems, Orion's basic components have proven applicable to interconnected distributed caches,[23] cluster switches, heterogeneous multi-core SoCs,[24] and shared-memory processors.[25] We seek to further extend its generality to new network fabrics in multi-core architectures, as well as wireless sensor networks.
Component attribute models. In the original release of Orion,[17] only dynamic switching power is modeled, with counters in each module tracking activity, and factored with parameterized capacitance equations for each module to derive power consumption. Now, in addition to dynamic power, Orion characterizes leakage power[26] as well as the thermal impact of networks.[27]These attribute models are designed to be parameterizable, so they can be applied to architecturally parameterized network modules. We also craft these modules so they can be readily scaled to different process technologies, facilitating reuse of modules across technologies.

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, communication, and ensuring sufficient simulation speed for large systems. 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.[28]

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 LSE components along the lines of its current modules and to formalize the interactions between modules according to the LSE model of computation. Our prior experience with porting SimpleScalar to LSE should help guide our development efforts in these regards.

3.5. Network Interface Component Library (NIL)

Network interfaces connect processors to networks and are realized both in ASICs and, more recently, as processor based programmable network interfaces. The network interface controller (NIC) interacts with its corresponding host over local interconnect, such as a PCI or other on-chip bus, and with the physical network, such as copper Ethernet or an on-chip packet switching network. Programmable NICs are particularly valuable because they enable flexibility to implement various complex network tasks, such as TCP processing, thus reducing the computation load of the host processor. Unlike general purpose programmable environments, network interfaces have stringent space and power requirements, and their workloads exhibit little data reuse or instruction-level parallelism (ILP). Consequently, the primary techniques used in general purpose processors to improve performance—large caches, complex ILP structures, and faster clocks—are inapplicable. Multiprocessor architectures, however, are appealing for programmable network interfaces, since parallelism enables performance increases in this domain without increasing clock speeds.

In addition to the programmability and multiprocessor execution environment, the heterogeneous set of components used in NICs (e.g., ASIC DMA units and medium access controllers) and the asynchronous nature of communication between the host and network complicate simulation of NICs. Previous studies have attempted performance simulation for these devices using conventional processor simulators.[29] Previous studies have attempted performance simulation for these devices using conventional

processor simulators or have studied the behavior of network interface workloads on network processor architectures using proprietary NP simulators.[29,30] However, no prior work to date has provided a flexible, realistic simulation platform that accounts for the special hardware features or software tasks supported by network interfaces.

We have developed a network interface simulator (as part of the SPINACH infrastructure[31] built on LSE), which, as a subset of its functionality, can model the MIPS-based Tigon-2 programmable network interface chipset.[31] A block diagram of the model is shown in Fig. 4. This model is sufficiently detailed to execute the firmware of a real Tigon-2 NIC while sending and receiving frames.[32,33] In fact, the model very accurately predicts the throughput of the Tigon-2 for a trace of bidirectional UDP packets, as shown in Fig. 5. The NIL also enables modeling of more advanced designs, such as those featuring many more processors, different processor models, crossbar-connected banked scratchpad architectures, split-transaction memory buses, and coherent caches. LSE's flexible reconfigurability permits rapid exploration of the permutations of these architectural choices.

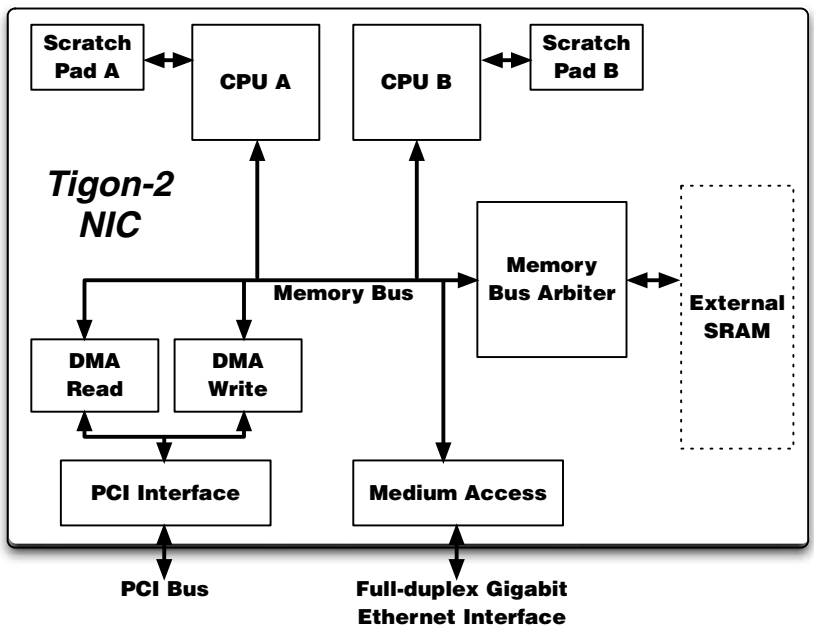

Fig. 4. A block diagram of a Tigon-2 NIC.

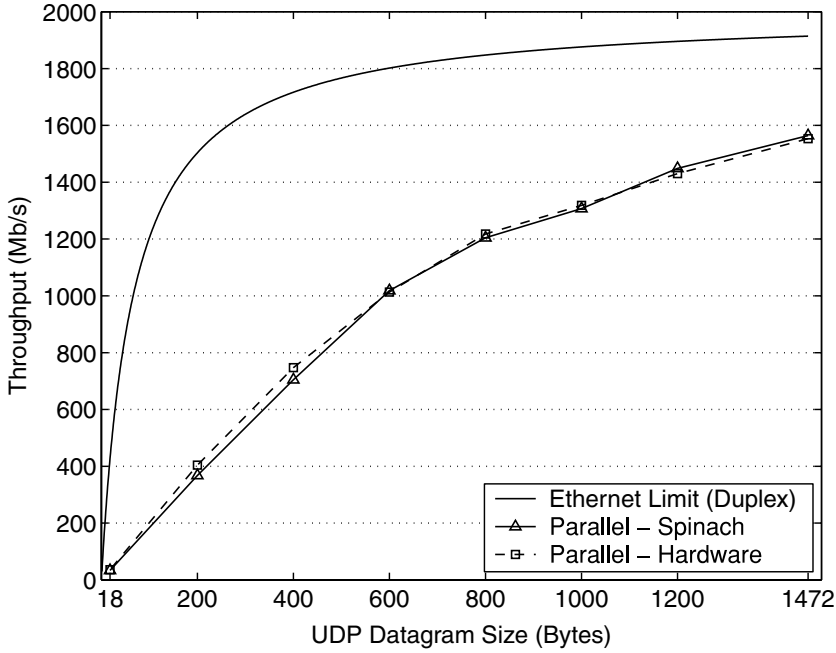

Fig. 5. The LSE model predicted transmit performance for a stream of UDP packets to respect to actual hardware.

Development of the Tigon-2 simulator consisted of two parallel tracks. One track focused on bringing up a uniprocessor sufficient to run the desired firmware, adding support for the various ASIC hardware units and memory mapped registers, and collecting the I/O traces of host and network traffic that will later drive the simulations. The second track contributed to the MPL by implementing a scalable parallel programmable network interface; in addition to the obvious multiprocessor issues, detailed memory system modeling is fundamental to the accuracy of network interface simulators, since NIC workloads are bandwidth-intensive. The components developed in this effort 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 interfaces using the NIL will continue to leverage modules of the UPL and the MPL.

4. CONCLUSIONS

Unlike traditional simulators, the 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, and our initial experience in modeling several systems, including the Itanium 2, the Tigon-2 NIC, and the Orion system, demonstrates the effectiveness of this approach. We have found that getting the most out of LSE requires the availability of a quality set of component libraries. Libraries currently in refinement are intended to be general enough for several target model domains, including uniprocessors, multiprocessors, networks, and programmable network interfaces.

By enabling varying levels of abstraction and a unified component connection frame-work, LSE provides excellent support for educational and for technology transfer needs. 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. We also thank the Liberty Research Group for their support and hard work in 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. 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).
2. H. W. Cain, K. M. Lepak, B. A. Schwartz, and M. H. Lipasti, Precise and Accurate Processor Simulation, *Proceedings of the Fifth Workshop on Computer Architecture Evaluation using Commercial Workloads* (February 2002).
3. J. Gibson, R. Kunz, D. Ofelt, M. Horowitz, J. Hennessy, and M. Heinrich, FLASH vs. (Simulated) FLASH: Closing the Simulation Loop, *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 49–58 (November 2000).

4. V. S. Pai, P. Ranganathan, and S. V. Adve, *RSIM Reference Manual, Version 1.0*, Electrical and Computer Engineering Department, Rice University (August 1997), technical Report 9705.
5. 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 (February 2002).
6. 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).
7. S. Önder and R. Gupta, Automatic Generation of Microarchitecture Simulators, *Proceedings of the IEEE International Conference on Computer Languages*, pp. 80–89 (May 1998).
8. S. Pees, A. Hoffmann, V. Živojnović, and H. Meyr, LISA – Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures, *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, pp. 933–938 (1999).
9. C. Siska, A Processor Description Language Supporting Retargetable Multi-Pipeline DSP Program Development Tools, *Proceedings of the 11th International Symposium on System Synthesis (ISSS)* (December 1998).
10. A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau, EXPRESSION: A Language for Architecture Exploration Through Compiler/Simulator Retargetability, *Proceedings of the European Conference on Design, Automation and Test (DATE)* (March 1999).
11. 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).
12. Open SystemC Initiative (OSCI), *Functional Specification for SystemC 2.0* (2001), URL http: //www. systemc.org,http://www.systemc.org.
13. 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).
14. P. Mishra, N. Dutt, and A. Nicolau, Functional Abstraction Driven Design Space Exploration of Heterogeneous Programmable Architectures, *Proceedings of the International Symposium on System Synthesis (ISSS)*, pp. 256–261 (October 2001), URL citeseer.nj.nec.com/506348.html.
15. M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August, Microarchitectural Exploration with Liberty, *Proceedings of the 35th International Symposium on Microarchitecture*, pp. 271–282 (November 2002).
16. M. Vachharajani, *Microarchitectural Modeling for Design-space Exploration*, Ph.D. thesis, Department of Electrical Engineering, Princeton University, Princeton, NJ USA (November 2004).
17. H.-S. Wang, X. Zhu, L.-S. Peh, and S. Malik, Orion: A Power-Performance Simulator of Interconnection Networks, *Proceedings of the 35th International Symposium on Microarchitecture* (November 2002).
18. P. Murthy, Modeling and Design of Reactive Systems (1997), presentation. URL: http://ptolemy.eecs.berkeley.edu/presentations/97/rasspfinal.pdf.
19. E. A. Lee and A. Sangiovanni-Vincentelli, Comparing Models of Computation, *Proceedings of ICCAD* (November 1996).
20. 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).

21. D. Penry and D. I. August, Optimizations for a Simulator Construction System Supporting Reusable Components, *Proceedings of the 40th Design Automation Conference* (June 2003).
22. N. Eisley and L.-S. Peh, High-Level Power Analysis of On-Chip Netowrks, *Proceedings of 7th International Conference on Compilers, Architectures and Synthesis for Embedded Systems* (September 2004).
23. B. M. Beckmann and D. Wood, Transmission Line Caches, *Proceedings of the International Symposium on Microarchitecture*, pp. 43–55 (December 2003).
24. T. T. Ye, L. Benini, and G. D. Micheli, Packetized On-Chip Interconnect Communication Analysis for MPSoC, *Proceedings of Design Automation and Test in Europe*, pp. 344–349 (March 2003).
25. H.-S. Wang, L.-S. Peh, and S. Malik, High-Level Power Analysis of On-Chip Netowrks, *In IEEE Micro, Vol. 24, No. 1, (Best of Hot Interconnects 10)* (February 2003).
26. X. Chen and L.-S. Peh, Leakage Power Modeling and Optimization of Interconnection Networks, *Proceedings of the International Symposium on Low Power and Energy Design*, pp. 90–95 (August 2003).
27. L. Shang, L.-S. Peh, A. Kumar, and N. K. Jha, Thermal Modeling, Characterization and Optimization of On-Chip Networks, *Proceedings of the 37th International Symposium on Microarchitecture* (December 2004).
28. 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, *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pp. 380–391 (June 1998).
29. P. Crowley, M. Fiuczynski, J.-L. Baer, and B. Bershad, Characterizing Processor Architectures for Programmable Network Interfaces, *Proceedings of the 14th International Conference on Supercomputing*, pp. 54–65 (May 2000).
30. K. Mackenzie, W. Shi, A. McDonald, and I. Ganev, An Intel IXP1200-based Network Interface, *Proceedings of the Second Workshop on Novel Uses of System Area Networks (SAN-2)* (2003).
31. P. Willmann, M. Brogioli, and V. S. Pai, Spinach: A Liberty-based Simulator for Programmable Network Interface Architectures, *Proceedings of the ACM SIGPLAN/SIGBED 2004 Conference on Languages, Compilers, and Tools for Embedded Systems*, pp. 20–29, ACM Press (July 2004).
32. Alteon Networks, *Tigon/PCI Ethernet Controller* (August 1997), revision 1.04.
33. Alteon WebSysterns, *Gigabit Ethernet/PCI Network Interface Card: Host/NIC Software Interface Definition* (July 1999), revision 12.4.13.