The Liberty Simulation Environment: A Deliberate Approach to High-Level System Modeling

MANISH VACHHARAJANI, NEIL VACHHARAJANI, DAVID A. PENRY, JASON A. BLOME, SHARAD MALIK, and DAVID I. AUGUST
Princeton University

In digital hardware system design, the quality of the product is directly related to the number of meaningful design alternatives properly considered. Unfortunately, existing modeling methodologies and tools have properties which make them less than ideal for rapid and accurate design-space exploration. This article identifies and evaluates the shortcomings of existing methods to motivate the Liberty Simulation Environment (LSE). LSE is a high-level modeling tool engineered to address these limitations, allowing for the rapid construction of accurate high-level simulation models. LSE simplifies model specification with low-overhead component-based reuse techniques and an abstraction for timing control. As part of a detailed description of LSE, this article presents these features, their impact on model specification effort, their implementation, and optimizations created to mitigate their otherwise deleterious impact on simulator execution performance.

Categories and Subject Descriptors: I.6.2 [**Simulation and Modeling**]: Simulation Languages; I.6.5 [**Simulation and Modeling**]: Model Development—*Modeling methodologies*; C.4 [**Performance of Systems**]—*Modeling techniques*; I.6.7 [**Simulation and Modeling**]: Simulation Support Systems—*Environments*

General Terms: Design, Experimentation, Human Factors, Languages

This work was supported by National Science Foundation grants CCR-0082630, CCR-0133712, and NGS-0305617; a grant from the DARPA/MARCO Gigascale Silicon Research Center; and donations from Intel. Opinions, findings, conclusions, and recommendations expressed throughout this work are not necessarily the views of the National Science Foundation, DARPA/MARCO, or Intel Corporation.

Authors' current addresses: M. Vachharajani, Department of Electrical and Computer Engineering, University of Colorado, 425 UCB, Boulder, CO 80309; email: manishv@colorado.edu; N. Vachharajani Computer Science Department, Princeton University, Princeton, NJ 08540; email: nvachhar@princeton.edu; D. Penry, Electrical and Computer Engineering, Brigham Young University, 459 Clyde Building, Provo, UT 84602; email: dpenry@ee.byu.edu; J. A. Blome, Advanced Computer Architecture Lab, Department of Electrical Engineering and Computer Science, University of Michigan, 2260 Hayward, Ann Arbor, MI 48109-2121; email: jblome@umich.edu; S. Malik, Electrical Engineering, Department, Princeton University, B224 Engineering Quad, Princeton, NJ 08540; email: sharad@princeton.edu; D. August, Computer Science Department, Princeton University, Princeton, NJ 08540; email: august@princeton.edu.

Additional Key Words and Phrases: Liberty Simulation Environment (LSE), structural modeling, simulator construction, component reuse

1. INTRODUCTION

In digital hardware system design, the quality of the product is directly related to the number of meaningful design alternatives properly considered. Since prototyping a candidate design is prohibitively expensive, designers rely instead on models to evaluate design alternatives. While analytical models have many desirable properties, current analytical modeling methods are only sufficient to provide accurate guidance for special cases. As a result, designers generally construct high-level (e.g., microarchitecture level) software simulation models for feedback.

In the computer architecture community, manually coding a simulator using a language such as C or C++ is the most common method of producing a simulation model.[1] Unfortunately, this methodology does not provide an *efficient* path to an *accurate* simulation model. The methodology requires the designer to meticulously map the microarchitecture, which is inherently structural and concurrent, to a sequential programming language with functional composition. At best, this manual mapping is labor-intensive and results in simulator code that does not conveniently convey architectural ideas. At worst, the simulator code is also difficult to understand and contains potentially serious errors that go unnoticed.

A common approach aimed at mitigating the problems related to the construction of these simulators is to reuse an existing, carefully constructed, and validated simulator for the exploration of similar designs. The belief is that modifying a validated simulator will be easier and result in an accurate derivative. However, as this article will show, simulator modification suffers from the same problems as simulator construction; it is time-consuming and error-prone. Worse, the quality of the original is likely to lead one to a false sense of confidence in the derivative, resulting in only cursory validation, and permitting potentially serious errors to remain unnoticed.

The concurrent-structural approach is a different approach that eliminates the mapping problem by simply eliminating the manual mapping. This approach involves a language which allows designers to directly express the composition of the hardware in terms of components and static connections. Without the need to manually map, the modeling process is much less labor intensive. Since the model is a description of the hardware design, it conveys architectural ideas, is easy for designers to understand, and exposes model/design mismatches.

Unlike in the manually coded simulator approach, reuse can be quite effective in the concurrent-structural approach. In concurrent-structural models, reuse at the component level is an attractive way to reduce model construction time [Swamy et al. 1995; Charest and Aboulhamid 2002] *and* improve

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

accuracy. A component can be built and validated once and then used repeatedly, reducing modeling effort and potential sources of errors. The utility of such reuse is demonstrated by common hardware components such as queues and arbitration elements which can be used, unmodified, in vastly different hardware designs.

This article will show, however, that existing concurrent-structural modeling languages and tools force a tradeoff between the ease of *building* reusable components and the ease of *using* such components. In current systems, this tradeoff puts a high overhead on reuse, reducing reuse in practice, and thus negating its benefits. Further, one aspect of hardware design, timing control, does not benefit from reuse in concurrent-structural systems since timing control is nonlocal in nature, making it difficult to partition into one or more reusable model components. Consequently, in existing systems, users are forced to manually specify control for each design.

To address problems with existing systems and methodologies, we present the design and implementation of the Liberty Simulation Environment (LSE). To avoid the mapping problem, LSE is based around a concurrent-structural model specification language. Unlike existing concurrent-structural systems, LSE supports the low-overhead use *and* construction of reusable components through several programming language techniques. LSE is also the first system to provide an abstraction that simplifies the specification of timing control. Finally, LSE descriptions are statically analyzable enabling, for example, simulator construction optimizations to improve simulator execution performance and tools for automatic model visualization.

The remainder of this article is organized as follows. The first few sections carefully analyze existing systems to identify the root cause of their shortcomings. Section 2 explores in detail how the manual mapping of microarchitectures to sequential programs is slow and error-prone, and why reuse cannot allow the cost of simulator development to be amortized. Section 3 analyzes systems that do not suffer from the mapping problem to determine why they still do not create an environment which encourages reuse. This analysis is then used to motivate the design of the Liberty Simulation Environment. Section 4 describes the Liberty Simulation Environment. Section 5 identifies the features that reduce component reuse overhead, and Section 6 describes LSE mechanisms to permit rapid specification of timing control. Section 7 discusses our experience with LSE and quantifies the reuse observed in practice. Finally, Section 8 concludes by summarizing the contributions of this article.

2. THE SEQUENTIAL MAPPING PROBLEM

To manage the design of complex hardware, designers divide the system's functionality into separate communicating hardware components and design each individually. Since each component is smaller than the whole, designing the component is significantly easier than designing the entire system. If components are too complex, they too can be divided into subcomponents to further ease the design process. The final system is built by assembling the individually designed components. To ensure the components will interoperate, designers,

when dividing the system, agree on the communication interface of each component. This interface, which defines what input each component requires and what output each component will produce, encapsulates the functionality of each component; other parts of the system can change without affecting a particular component provided its communication interface is respected. We call this type of encapsulation and communication *structural composition*.

Leveraging encapsulation to allow this divide-and-conquer design strategy is also very common in software design. Sequential programming languages such as C or C++ use functions to encapsulate functionality. Each function has a communication interface (its arguments and return value) and this interface encapsulates the function's behavior. Software systems are built by assembling functions which communicate by calling one another and passing arguments and receiving return values. We call this type of encapsulation and composition *functional composition*.

The presence of encapsulation combined with designer familiarity and tool availability make sequential programming languages seem like a natural tool with which to model hardware systems. However, as will be seen in this section, the encapsulation permitted by functional composition in sequential languages is not the same as the encapsulation provided by structural composition. This mismatch forces designers to *map* their structurally composed hardware designs to functionally composed sequential programming languages. This section demonstrates that this mapping is time-consuming, error-prone, and yields simulators that are difficult to understand and hard to modify. Thus we can conclude that, despite the popularity of this methodology, manually coding hardware models in sequential languages is ill-suited for design space exploration.

The discussion of the *mapping problem* proceeds as follows. Section 2.1 describes why simulators built using sequential languages (*sequential simulators*) are hard to build, difficult to understand, and, thus, prone to error. Section 2.2 presents empirical data supporting this claim. Section 2.3 explains why building a new simulator by modifying an existing one is difficult, illustrating that the cost of building and validating simulators cannot be amortized across many designs during exploration. Section 2.4 presents empirical data supporting this claim.

2.1 Simulator Construction and the Mapping Problem

When dividing a complex hardware design into simpler components, designers choose a partitioning that allows them to most easily understand the design. Often this partitioning forms the vocabulary that designers use to think about and discuss the design. Consequently, the easiest simulator to build and understand would share this same partitioning. Unfortunately, differences between the styles of encapsulation used in hardware and sequential programming languages prevent this. As will be described in this section, when mapping from hardware components to software functions, the encapsulation provided by the hardware components must be broken forcing the designer to reason about many components simultaneously. This reasoning, and therefore the mapping, is laborious and extremely time-consuming. Further, since the encapsulation

of code in the simulator is not representative of the hardware, understanding how a simulator written in a sequential language models hardware is also difficult. Ultimately, modeling hardware in a sequential language hides pieces of a component's interface, intertwines computation and communication, and requires manual orchestration of concurrency.

A fundamental attribute of the encapsulation provided by hardware is the explicit specification of interfaces and the clear separation between functionality and communication. A hardware component will typically define its communication interface as a collection of ports through which it receives input and sends output. The component will define its behavior by specifying how it translates data arriving at its input ports to data it will send to its output ports. Independent of this specification of interface and behavior, the communication of the system is determined by the connectivity of its components' ports. A particular component may receive input from one or more other components and, similarly, may send its output to one or more recipients.

The encapsulation provided by functions in sequential programming languages seems similar. The arguments to the function seem to mirror a component's input ports, and the return value seems to mirror the output ports. The body of the function specifies its behavior as a translation from inputs to outputs. Unfortunately, calling a function from within the body of another function implicitly augments the communication interface of the caller and intertwines functionality with communication. The arguments sent to the callee and the return value received from it are additional *implicit* outputs and inputs of the caller. Further, the recipient of data sent and the sender of data received from this augmented interface is determined by the function being called. Unlike structural composition, a function must receive all of its arguments from a single caller and send all of its outputs back to that same caller. Therefore, the arbitrary and independently specified communication patterns provided by structural composition are absent when using functional composition.

An alternative style of modeling hardware in a sequential programming language uses global variables, as opposed to function arguments and return values, to communicate information through the system. Unfortunately, in such systems, the problems discussed above still exist. The communication interface of a particular function is still implicitly specified through its behavior specification. Each global variable accessed defines a piece of the function's communication interface. Further, the behavior and communication of a function are still intertwined since two functions communicate if one writes to a global variable that the other reads. Furthermore, when using global variables, even the specification of communication is implicit, unlike in the previous style. The target of communication is never explicitly specified but implied by analyzing how data flows through global variables. Two functions may appear to communicate because they access the same global variable, but a third function may overwrite the global variable after the first has written it but before the second has consumed the data. Careful examination is required to truly understand the communication present in such systems.

The implicit communication when using global variables reveals another shortcoming of the encapsulation provided by functional composition.

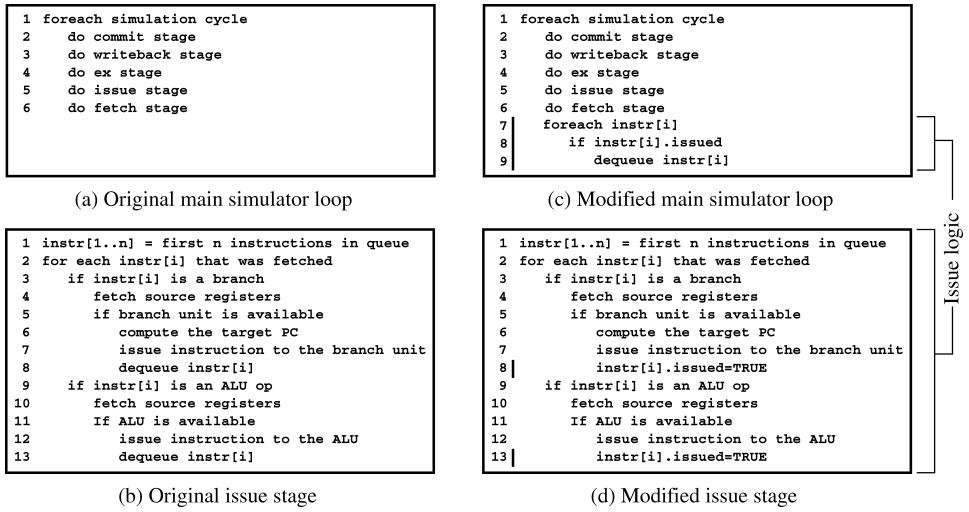


Fig. 1. Sequential simulator code.

Components in a hardware system execute *concurrently* with one another. If a component has sufficient input to perform a computation, it will proceed without waiting for additional input. With sequential programming languages and functional composition, however, the interactions between components must be manually orchestrated by sequencing function invocation. Sequencing these invocations may not be straightforward. For example, when using global variables to communicate, interchanging the order in which two functions are called can cause data to be delayed by a cycle or can even change the communication pattern. Great care must be taken to ensure the proper sequence is specified. Worse still, if functions have not been appropriately partitioned, there may be no correct order of invocation. For example, if component A generates output that feeds component B, and an output of component B feeds component A, then no order of invocation between A and B will work. The functions would need to be partitioned so that the new functions could be scheduled.

The problems discussed above are all manifestations of the mapping problem. To see how this problem can occur in practice, consider the following example. Figure 1(a) shows a typical main simulation loop for a sequential simulator that models a five-stage superscalar processor pipeline. The hardware is modeled using a function per pipeline stage. The functions communicate through global variables, which effectively model the pipeline registers between the stages. Since later pipeline stages wish to use data produced from previous cycles, they must run before the global variables get overwritten by earlier stages. Therefore, the main simulator loop begins computation at the back of the pipe and moves toward the front so that later pipeline stages use state from previous cycles, before earlier stages overwrite the data. This invocation order also allows back-pressure to flow through the pipe. If a stage later in the pipeline stalls, it can set a global variable to inform earlier stages of the stall.

We now focus on the issue stage of the pipeline, whose code is shown in Figure 1(b). From the pseudocode we see that, when an instruction is sent to its

functional unit, it is simultaneously removed from the instruction window (lines 7–8 and 12–13 in Figure 1(b)). When the fetch stage (the stage that places instructions into the instruction window) is executed, the newly created space will be available for new instructions.

Now, suppose that the designers would like to model a different behavior in which freed slots in the instruction window are not available until the cycle after the instruction was issued. Such a behavior may be desirable if, for example, the dequeue signals would arrive too late in the cycle with the original behavior. The hardware differences between the original and the new behavior simply amount to removing the dequeuing logic from the computation of a control signal indicating the number of slots available. Figures 1(c) and 1(d) show the necessary changes to the simulator main loop and issue logic, respectively, to model the new behavior.

Notice that the sequential simulator code that models two very similar architectures contains significant differences. These differences are indicated by the bars to the right of the line numbers in Figure 1. Specifically, the change to the microarchitecture required partitioning of code for the issue logic and the addition of new simulator state to allow the pieces of the issue logic to communicate. The code to dequeue instructions from the instruction window had to be separated from the code that dispatched instructions to the functional units since these two events occur at different times in the modified hardware design. The majority of the issue logic remains in the issue stage function, but some of the logic is now intermingled with the code that schedules the execution of the pipeline stages (lines 7–9 in Figure 1(c)). The modified code also needs an additional global variable to maintain the issued status for each issue window slot so that the piece of the issue logic that dequeues instructions knows which instructions were issued.

Just as changing instruction window timing required partitioning a single logical entity in the hardware into multiple functions in the simulator model, other microarchitecture features may also force undesirable partitioning. While this small example may not seem overwhelming, this kind of partitioning is very common throughout the code for sequential simulators. Because of this, sequential simulator authors need to carefully plan how hardware component functionality needs to be partitioned, decide what global state will be used for communication, and carefully orchestrate the invocation of functions to ensure that all global state is updated in the correct sequence.

Notice that this mapping process can be extremely complicated and therefore difficult to perform correctly. Since mistakes can easily be made, the resulting simulator needs to be carefully checked to ensure correctness. Correctness is usually determined by testing each component as a unit and then testing the composed whole. In the best case, the entire model will be built by reusing prevalidated components. As we have seen, however, the mapping process breaks structural encapsulation, thus making component testing and component-based reuse impossible. Even components commonly thought of as testable units in a sequential simulator, such as a cache component, are not independently testable since, to allow correct modeling of timing, they are often tightly coupled to the whole simulator [Desikan et al. 2001].

Sequential simulators are also difficult to manually validate as a whole. Designers understand the hardware in terms of hardware components and their communication. A strict separation of computation, communication, and operation sequencing is critical to the understanding of a hardware design. As seen above, however, the way in which a sequential simulator is built breaks component encapsulation and intermingles communication and computation. Furthermore, recall that with sequential simulators communication between code that models hardware blocks is often implicit. This makes the resulting simulator difficult to understand and thus difficult to manually validate. This in turn makes an *accurate* simulator even more difficult and time-consuming to build.

2.2 Simulator Construction and the Mapping Problem: Model Clarity

The previous discussion describes why accurate sequential simulators are difficult to build. Correctness is especially hard to ensure because the simulator is very hard to understand. The experiment presented in this section supports the claim that sequential simulators are difficult understand and thus difficult to validate.

To quantify the clarity of sequential simulators, a group of subjects was asked to examine sequential simulator code modeling a microprocessor and identify properties of the machine modeled. As a reference point, the subjects were asked *exactly* the same questions for a model of a similar machine built in the Liberty Simulation Environment (LSE). As described in detail later, LSE is a hardware modeling framework in which models are built by connecting concurrently executing software blocks much in the same way hardware blocks are connected. Thus, LSE models closely resemble the hardware block diagrams of the machines being modeled. We call these structurally composed models *structural models* to distinguish them from functionally composed *sequential simulators*.

Subjects received different versions of the machine models to ensure that the effects observed were not a due to a particularly hard-to-understand hardware policy. The sequential simulators used were modified and unmodified versions of `sim-outorder.c` from version 3.0 of the popular SimpleScalar tool [Burger and Austin 1997]. `sim-outorder.c` models a superscalar machine that executes the Alpha instruction set. The LSE models were variations of a superscalar processor model that executed the DLX instruction set. A refined version of this model was released along with the Liberty Simulation Environment in the package `tomasulodlx` [The Liberty Research Group 2003].

To ensure that the experiment measured the quality of the model, not the knowledge of the subjects, all the subjects were either Ph.D. students studying computer architecture or Ph.D. holders whose primary work involved computer architecture. To ascertain the background of subjects, each was given a questionnaire to determine their familiarity with computer architecture, programming languages, and existing simulation environments, particularly, SimpleScalar. A summary of the answers to this questionnaire is in Table I.

Note that finding qualified subjects unaffiliated with the Liberty Research Group for this experiment was challenging given the requirements. Subjects had to be very familiar with computer architecture and also had to be familiar

Table I. Subject Responses to the Background Questionnaire

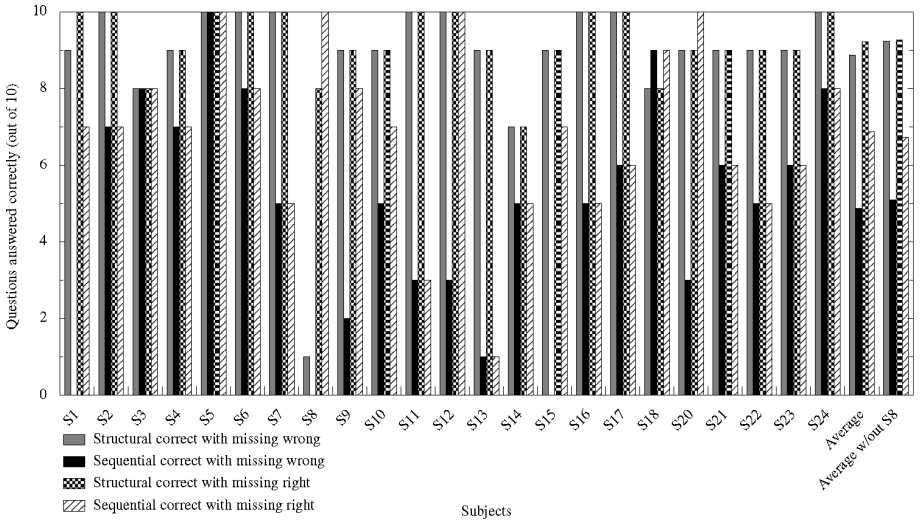
Subject	Years in Architecture	Wrote a C Simulator	Wrote RTL for a CPU Core	Years Experience w/ C/C++	Days Experience w/ LSE	Used Simple-Scalar
S1	3	Yes	No	5	3	Yes
S2	10	Yes	Yes	15	2	No
S3	3	No	Yes	5	2	No
S4	3	No	No	6	3	Yes
S5	3	No	No	7	3	No
S6	3	Yes	Yes	6	3	Yes
S7	3	No	Yes	8	3	Yes
S8	3	No	No	5	4	No
S9	2	No	No	14	5	No
S10	6	Yes	No	10	5	Yes
S11	7	No	No	7	2	No
S12	3	Yes	Yes	2	2	Yes
S13	7	No	Yes	10	2	No
S14	2	No	No	10	2	No
S15	1	No	No	10	2	No
S16	2	Yes	No	8	2	Yes
S17	6	No	No	6	2	No
S18	3	No	No	4	2	Yes
S20	3	No	No	5	2	Yes
S21	2	Yes	No	6	2	Yes
S22	2	Yes	No	6	2	Yes
S23	1	No	Yes	10	21	No
S24	4	No	Yes	6	7	No

with LSE. Though growing in popularity, LSE was still a new system, making the second constraint the most difficult to satisfy. Only 24 of all LSE users could be recruited as subjects for this experiment. As will be seen, however, even with 24 subjects the results were quite dramatic.

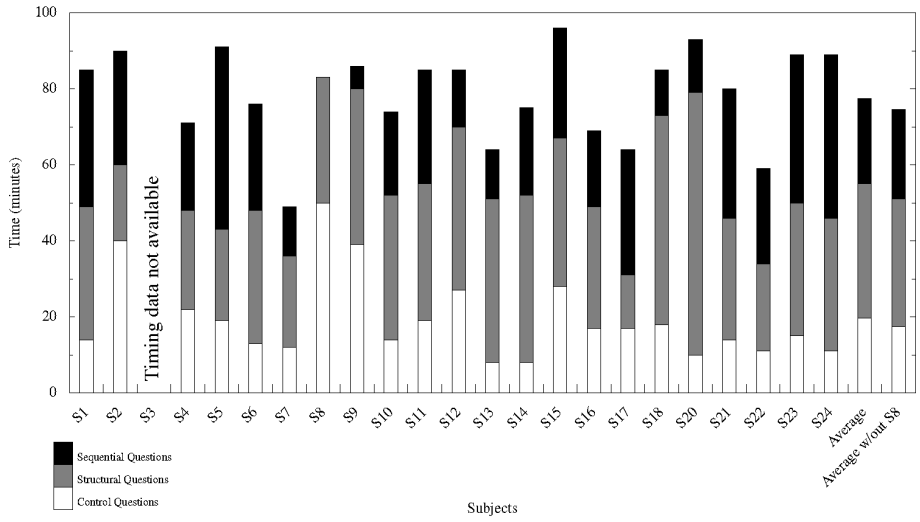
Subjects were given 90 min to answer two control questions, two multipart questions for the structural model, and two identical multipart questions for the sequential simulator (for a total of 10 question parts, excluding the control questions). These questions, with placeholders for the line numbers, can be found in the literature [Vachharajani 2004]. The control questions were used to determine if the subjects had basic familiarity with LSE since all but one subject had less than 1 week of experience with the tool. The answers to the control questions indicated that all subjects, except subject S19, understood LSE enough for the purposes of this experiment. Accordingly, the data presented here excludes subject S19. It is clear from the questionnaire results in Table I that all remaining subjects were familiar with the C programming language, the language with which the sequential model was constructed.[2]

Figure 2(a) summarizes the results. Since the subjects had limited time, not all subjects could answer all questions. The first and third bars correspond to responses regarding the structural model. The first bar assumes that all questions left unanswered were answered incorrectly. The third bar shows the

[2]More details regarding the questions and machine models are available in the references [Vachharajani 2004].



(a) Number of correctly answered questions, by subject



(b) Total time taken for each group of questions, by subject

Fig. 2. Results of the simulator clarity experiment.

same data assuming that all unanswered questions were answered correctly. The second and fourth bars show the same information for questions regarding the sequential simulator. Just as before, the second bar assumes that unanswered questions were answered incorrectly, and the fourth bar assumes that unanswered questions were answered correctly. "Missing" bars represent subjects that answered no questions correctly for a particular model. From the

graph, we can see that in each case subjects were able identify the machine properties at least as accurately with the structural model as they were with the sequential simulator, and usually much more accurately. On average, subjects answered 8.87 questions correctly with the structural model versus 4.87 for the sequential model. Even if unanswered questions are assumed correct for *only* the sequential simulator, almost all subjects still answered more questions correctly with the structural model. The only subjects who did better with the sequential simulator, under these circumstances, were subjects S8 and S20. Subject S20 simply failed to respond to 7 out of 10 of the questions for the sequential simulator. S8 spent an unusually long time on the control questions and thus did not have time to answer any questions regarding the sequential simulator. In fact, S8 only completed the first two parts of the first noncontrol question for the structural model and no questions for the sequential simulator. Clearly subject S8 is an outlier.

If subject S8 is excluded from the data, we see that, on average, 9.23 questions were answered correctly for the structural model versus 5.09 for the sequential simulator. Even when unanswered questions are assumed correct for the sequential simulator and incorrect for the structural model, the structural model still had 9.23 correctly answered questions versus 6.73 for the sequential simulator. The only subject who actually answered more questions correctly for the sequential simulator was subject S18. The best explanation for this is that S18 was extremely familiar with `sim-outorder.c`. The model that the subject examined was the stock `sim-outorder.c` model and, the subject took very little time to answer the sequential questions. An interesting experiment would have been to test S18 on a modified version of the `sim-outorder.c` models, but a retest was not possible.

Figure 2(b) summarizes the time taken to answer each class of questions: control questions, questions for the structural model, and questions for the sequential model. With S8 excluded, the average time taken per question for the structural model was greater by about 10 min when compared to the time for questions about the sequential simulator. Some of this gap is due to the fact that some subjects did not complete all the sequential simulator questions, and the time taken for unanswered questions did not count toward the average. Despite this, the increased number of correct responses for the structural model was not likely due to the extra time spent on those questions; many of the total times are well below the 90-min time limit. One conclusion to be drawn from the timing data is that the sequential simulator is extremely misleading. Subjects voluntarily spent less time answering the sequential simulator questions despite plenty of extra time. Presumably, this was due to their confidence in their rapid analysis. Yet the subjects frequently mischaracterized properties of the machine modeled. Timing data for subject S3 was unavailable because the subject failed to record the data during the examination.

Note that the experiment was skewed in favor of the sequential simulator. First, subjects were asked the LSE question immediately before being asked the same question for the sequential simulator. Thus subjects could use the hardware-like model to understand what to look for in the sequential simulator. Second, many of the subjects were familiar with the stock `sim-outorder.c`

simulator. Third, all subjects had many years of experience using the C language (the language used for `sim-outorder.c`) and less than a week of experience with LSE (the tool used to build the structural model). Fourth, no subject had ever seen a full LSE processor model before the experiment. Finally, the testing environment did not permit subjects to use the LSE visualization tool [Blome et al. 2003] to graphically view block diagrams of the structural specification. Experience indicates that this tool significantly simplifies model understanding; subject responses to the questions indicated that much of their time answering questions about the structural model was spent drawing block diagrams. Despite all this, the results clearly indicate that sequential simulators are significantly more difficult to understand.

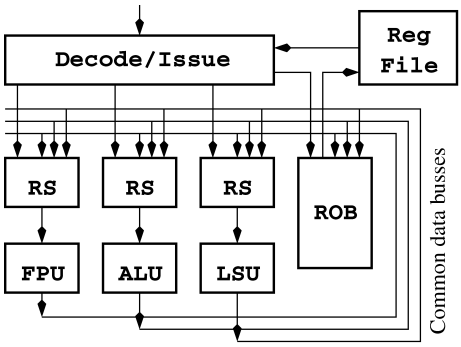
2.3 Reuse and the Mapping Problem

A tempting approach to allow rapid construction of simulation models in the face of the previously described difficulties is to amortize the cost of model construction via whole-model reuse. In this approach, a sequential simulator is built once, validated versus a "golden" reference, such as real hardware, and then modified to model a new design. Since the new model is a modification of a validated model, the belief is that the likelihood of error is reduced and modeling efficiency is increased. However, this is not the case. Sequential simulators are not only difficult to build and understand, but also difficult to modify correctly.

To see why these simulators are difficult to modify correctly, consider a simulator, written in the same style as the one in Figure 1, which models a machine that uses Tomasulo's dynamic scheduling algorithm. Figure 3(a) presents a block diagram of such a machine, and Figure 3(b) shows the code for the writeback stage of the pipeline. The code iterates over all instructions that have completed execution and updates the dependency information of instructions pending execution. While the code seems to model the hardware reasonably well, closer examination reveals that this machine has unrestricted writeback bandwidth; any instruction that has completed execution will be written back.

Limiting the writeback bandwidth seems simple. One need only modify the loop termination condition to cause the loop to exit if the writeback bandwidth has been exceeded. Figure 3(d) shows this modified code. Careful inspection, however, reveals that this one line code modification has had unexpected results. The functional units which will be able to successfully write back results to the writeback buses is determined by the order that the while loop (line 1 in Figure 3(d)) processes the requests. Thus, as shown in Figure 3(c), the sequential semantics of the programming language used during modeling has *implicitly* introduced a bus arbiter that is not *explicitly* modeled by the code.

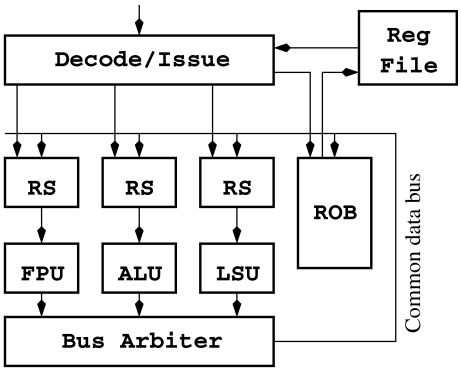
Worse still, the arbiter's exact functionality depends on *simulator state* that does not correspond to any *microarchitectural state*. The order in which the simulator iterates over the instructions in the while loop determines which instructions are written back. Prior to this modification, the iteration order was irrelevant. Now the iteration order determines the behavior of the microarchitecture



(a) Original machine

```
1 while( instructions are pending )
2    instr=next pending instruction
3    if instr is a mispredicted branch
4       set flag so reorder-buffer kills
            dependent ops as they leave pipe
5    else
6       foreach reservation station r
7          foreach entry e in r
8             if e.need_value(instr.dest) then
9                e.set_src(instr.dest)
```

(b) Pseudocode for original machine



(c) Limited writeback machine

```
1 while( instructions are pending AND
       instructions written back this cycle < n )
2    instr=next pending instruction
3    if instr is a mispredicted branch
4       set flag so reorder-buffer kills
            dependent ops as they leave pipe
5    else
6       foreach reservation station r
7          foreach entry e in r
8             if e.need_value(instr.dest) then
9                e.set_src(instr.dest)
```

(d) Pseudocode for limited writeback machine

Fig. 3. The structure and pseudocode for two Tomasulo-style machines.

and this order is not necessarily determined in any one place in the simulator's code. For example, the iteration order can be affected by the order in which functional units are processed (which is often arbitrary) or the specific implementation of the data structure used to store the instructions waiting to write back. Thus, this small change breaks the encapsulation of the writeback stage allowing seemingly irrelevant implementation details to affect simulation results.

2.4 Reuse and the Mapping Problem: Simulator Modification Time

To show that correctly modifying a sequential simulator is unnecessarily difficult and time-consuming, an experiment that gauges how rapidly an existing simulator could be *correctly* modified was conducted. Specifically, a subject was asked to perform various modifications to a sequential simulator and to an equivalent model whose specification more closely resembles hardware. The requested modifications are shown in Table II. The sequential simulator modified by the subject was the SimpleScalar 3.0 `sim-outorder.c` simulator. The model that more closely resembles hardware was once again built with the Liberty Simulation Environment. The base LSE model and the base `sim-outorder` simulator[3] that were modified in the experiment had exactly matching pipeline traces. At the time this experiment was conducted, the subject had little

[3]The base `sim-outorder.c` model had a few patches applied to fix bugs in the stock model.

Table II. Descriptions of the Modeled Microarchitectural Variants

Configuration Name	Configuration Description
mispred_imm	Force all branches to resolve immediately in the writeback stage
mispred_old	Force all branches to resolve in order in the writeback stage
mispred_com	Force all branches to resolve in order in the commit stage
delaydec	Place one cycle of delay after decode
splitda	Split the decode stage into decode and register rename
splitruu	Split the issue window from the reorder buffer (split RUU into 2 modules)

Table III. Time Spent and Code Changed for Modifications from the Baseline Configuration

	LSE Model			Sequential Model		
Configuration	diff/wc	Modules Affected	Time	diff/wc	Functions Affected	Time
mispred_imm	146	8	1.5hrs	400	16	5 hrs
mispred_old	165	9	45 min	413	16	1.5 hrs
mispred_com	177	10	15 min	629	17	15 min
delaydec	16	1	15 min	94	6	2 hrs
splitda	124	5	40 min	N/A	N/A	>5 hrs
splitruu	13	1	36 min	50	7	3 hrs

experience with both LSE and SimpleScalar. To ensure correctness of the modified models, each pair of modified models' output was carefully checked by hand and against each other. The subject had to resolve any discrepancies between pipeline traces generated by the two models.

In order to evaluate how difficult it was to correctly modify the sequential simulator, three metrics were used to compare the sequential model to the LSE model. The first metric, the diff/wc metric, measured how much a specification deviated from the base specification by counting the number of lines in a `diff` between the original and modified configuration. The second metric captured the locality of the changes necessary to move from an initial architectural model to a modified one. Since the specifications of the two simulators being compared were different, a hand count of the number of components affected in the LSE model was compared to a hand count of the number of C functions modified for SimpleScalar. The final metric used was a timing of how long each particular modification took. Any time required to resolve discrepancies between the output of two models was charged to the model that was deemed to be incorrect.

The results of the experiment are summarized in Table III. Note that the `splitda` modification could not be completed in the sequential model in under 5 h and was abandoned. Across the board, it took less time and fewer modifications to build the LSE model when compared to the hand-coded sequential C simulator. Furthermore, the changes were more local in the LSE specification than they were in SimpleScalar. These results clearly indicate that sequential models are unnecessarily time-consuming to modify given that the LSE models can be used to automatically generate simulators. Note that in each case where there was a discrepancy, inspection revealed that the sequential model was the one that contained the error. This lends support to the claim that sequential simulator construction and modification are error-prone.

2.5 Prognosis

The examples and data presented demonstrate that manually mapping a concurrent, structural microarchitecture to a *correct* sequential program can be quite difficult. Others have noted that there is a problem with the accuracy of simulators, but disagree on the source of the problems [Desikan et al. 2001; Cain et al. 2002; Gibson et al. 2000]. We contend that the mapping problem is the fundamental cause of inaccuracies in and long development times of sequential simulators.

While validation may seem like an attractive solution to the problem of sequential simulator accuracy, validating a sequential simulator will further lengthen simulator development times. Furthermore, for a novel design, it is difficult to determine if a simulator is correct since no "golden" reference exists and the simulator is difficult to understand.

Unfortunately, no obvious technique to rapidly hand-craft correct sequential simulators has been proposed to date. A number of authors have proposed architecture description languages (ADLs) as an alternative means of modeling [Halambi et al. 1999; Pees et al. 1999; Siska 1998]. Unfortunately, these languages can only be used for processor modeling, and even within that domain the tools are generally inadequate. The systems either suffer from the mapping problem or limit the class of processors that can be modeled [Vachharajani et al. 2002].

Fortunately, the intuition and the results from the presented experiments indicate that models designed using a general-purpose methodology which allows structurally composed concurrently executing components can prove effective. However, some structural systems [Emer et al. 2002] still force functional composition for some intercomponent computation, while others [Önder and Gupta 1998; Mishra et al. 2001] suffer from limited models of concurrency [Vachharajani et al. 2002]. In other systems, building a component, validating it, and reusing it across many designs is a possibility. This improves both the speed of model construction and reduces sources of error. In the next few sections, existing modeling tools that use concurrency and structural composition will be surveyed and analyzed in order to determine how effectively their features support the creation and use of reusable components.

3. EXISTING TRUE CONCURRENT-STRUCTURAL APPROACHES

From the previous section, it is clear that a simulation system that avoids the mapping problem must be fully concurrent and support structural composition. This way the modeling methodology itself does not *prevent* the construction and use of reusable components. In practice, however, it is insufficient to simply not prevent component-based reuse; high-level concurrent-structural modeling systems need to possess certain capabilities which *enable* component-based reuse [Swamy et al. 1995]. These capabilities include the following:

—*Parameters*. The ability to customize component properties with user-specified values:
  —*Structural parameters*: the ability to customize hierarchical structure with parameters. This allows existing components to be reused hierarchically

Table IV. Capabilities of Existing Methods and Systems

	Static Structural		Concurrent-Structural OOP	
Capability	Theory	Practice	Theory	Practice
Parameters	Yes	Yes	Yes	Yes
—Structural			Yes	Yes
—Algorithmic	Yes	Yes	Yes	Yes
Polymorphism	Yes	Yes	Yes	Yes
—Parametric	Yes		Yes	Yes
—Overloading	Yes	Yes		
Static analysis	Yes	Yes		
Instrumentation	Yes		Yes	

to create a *flexible* component. Example: parameters controlling the mix of functional units and presence of bypass connections in a structurally specified reusable CPU core:

—*Algorithmic parameters*: the ability to inherit and augment the behavior of an existing component with an algorithm. Example: parameter specifying arbitration logic inside a bus arbiter component:

—*Polymorphism*. the ability to support reuse across varying data types:

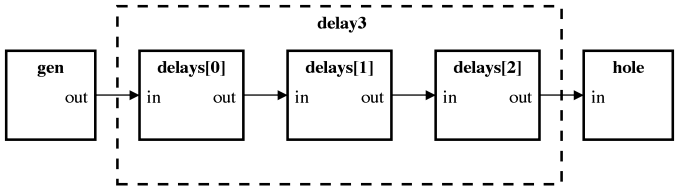
—*Parametric polymorphism*: the ability to create and use component models in a data type independent fashion. Examples: queues, memories, and crossbar switches that can store or process any data type.

—*Component overloading*: the ability to select different component implementations to match different data types. Note that function overloading, in which *argument* types select a function's implementation, differs from component overloading, where *port and connection* types select a component's implementation. Example: automatic selection between floating point ALU implementation and integer ALU implementation based on connection data types.

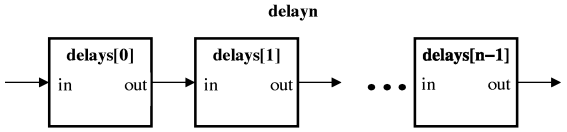
—*Static analysis*. The ability to analyze the resulting concurrent-structural model for user convenience, verification, and simulator optimization. Example: type inference to automatically resolve polymorphic port types.

—*Instrumentation*. The ability to probe a model for dynamic behavior without modifying the internals of any component. This allows reuse across different model objectives. Examples: data collection, debugging, and visualization.

The following two subsections relate the above abilities to two existing modeling methodologies: static structural modeling and modeling with a concurrent-structural library in an object-oriented programming (OOP) language. These systems are *true* concurrent-structural systems and thus *do not* suffer from the mapping problem. The analysis in this section will identify which of the above capabilities are supported in each of these methods and also highlight potential pitfalls present in these methods. The insight gained will guide the design of the Liberty Simulation Environment and its modeling language, the Liberty Structural Specification (LSS) language. Table IV can be used as a reference during the discussion.



(a) Block diagram of a three-stage delay chain specification



(b) Block diagram of a flexible n-stage delay component

Fig. 4. Block diagrams of chained delay components.

3.1 Static Structural Modeling

Static structural modeling systems are concurrent-structural modeling systems that statically describe a model's overall structure. Models in these systems often resemble netlists of interconnected components, and typically these tools have drag-and-drop graphical user interfaces to construct models. Examples of such tools are Ptolemy II with the Vergil interface [Janneck et al. 2001] and HASE [Coe et al. 1998].

These systems support many of the features described above. Components typically export parameters so that they can be customized. Depending on the underlying language used to implement the components, a mechanism may exist to support algorithmic parameters via inheritance. Some systems support polymorphism [Janneck et al. 2001] and type inference to resolve the polymorphic types [Xiong 2002]. Models could even be instrumented using aspect-oriented programming (AOP) [Kiczales et al. 1997] to weave instrumentation code into the structure of the described model.

Unfortunately, the fact that these specifications are static implies a fundamental limitation of static structural modeling systems. Consider the structure shown in Figure 4(a). In static structural systems, one would explicitly instantiate the three blocks within the dotted box in the figure. However, this chain of blocks could not be wrapped into a flexible hierarchical component, as shown in Figure 4(b), where the length of the chain is a parameter since static structural systems provide no mechanism to iteratively connect the output of one block to the input of the next a parametric number of times. As a result, to permit flexibility, this simple hierarchical design would have to be discarded in favor of a more complex implementation of a primitive component implemented using a sequential programming language. Implementing the primitive component for this simple example may not be difficult, but more

complex examples, such as parametrically controlling the mix of functional units in a processor model, would require implementing a monolithic primitive processor component. This is tantamount to writing the whole simulator in a sequential programming language. Note that some static structural modeling systems may provide idioms for common patterns, such as chained connections. However, the fundamental lack of general mechanisms to parametrically and programmatically control model structure still remains. This deficiency ultimately restricts the flexibility of components built hierarchically and forces users to build large primitive components.

3.2 Modeling with Concurrent-Structural Libraries in OOP

A promising concurrent-structural modeling approach, such as the one taken by SystemC [Open SystemC Initiative (OSCI) 2001], which allows flexible primitive *and* hierarchical components, is to augment an existing OOP language with concurrency and a class library to support structural entities such as ports and connections. Objects take the place of components, and simulator structure is created at run-time by code that instantiates and connects these objects.

The basic features of object-oriented languages provide many of the capabilities described above. Object behavior can be customized via instantiation parameters passed to class constructors. Algorithmic parameters are supported via class inheritance. If the particular OOP language and the added structural entities support parametric polymorphism, then type-neutral components can be modeled as well.

Since component instantiation and connection occur at run-time, the OOP language's basic control flow primitives (i.e., loops, if statements, etc.) can be used to *algorithmically* build the structure of the system. This code can be encapsulated into an object and the internal structure can be easily controlled by structural parameters thus producing *flexible* hierarchical components. For example, the n-cycle delay component (Figure 4(b)) seen in the last section could be built by composing n single-cycle delay components as shown in the pseudocode in Figure 5.

Unfortunately, run-time composition of structure provides component flexibility by precluding static analysis of model structure. This makes using these flexible components cumbersome. For example, any parametric polymorphism must be resolved via explicit type instantiation by the user, since the constraints used in type inference are obtained from the model's structure, which is unavailable at compile time. Ideally connecting the output of a floating point register file to an overloaded ALU should automatically select the ALU implementation that handles floating point data. However, this component overloading is not possible since the user must codify the particular ALU implementation in the instantiation statement rather than the compiler automatically determining this based on connectivity. Additionally, all component parameters, particularly those that control structure, must be explicitly specified by the user since the compiler is unable to automatically infer these values by analyzing the structure of the machine statically. Finally, implementing instrumentation that is orthogonal to machine specification is at best

```
1  class delayn {
2    public InPort in;
3    public OutPort out;
4
5    Delay[] delays;
6    delayn(int n) {
7      int i;
8
9      in=new InPort();
10     out=new OutPort();
11
12     delays=new Delay[n];
13     for(i=0;i<n;i++) {
14       delays[i]=new Delay();
15     }
16
17     in.connect(delays[0].in);
18     for(i=0;i<n-1;i++) {
19       delays[i+1]=new Delay();
20       delays[i].out.connect(delays[i+1].in);
21     }
22     delays[n-1].out.connect(out);
23   }
24 };
```

Fig. 5. Concurrent-structural OOP pseudocode for an n-stage delay chain.

cumbersome. Powerful techniques such as aspect-oriented programming cannot be used since the desired join points (locations where instrumentation code should be inserted) are often parts of the model structure that is not known until run-time.

In addition to burdening the user, the lack of static analysis prevents certain key optimizations that can increase simulator performance. Existing optimizations can provide as much as a 40% increase in simulator performance [Penry and August 2003], eliminating performance loss due to reuse, and future optimizations will enable parallel execution of simulators for further speed enhancement. These optimizations rely on static analysis of machine structure and thus are unavailable when run-time composition of structure is used. In practice, simulator performance penalties combined with reuse burdens encourage users to build design-specific components that are fast and easy to use but enjoy little to no reuse due to their inflexibility.

3.3 Mixed Approaches

A few approaches share some features of static structural models and some features of concurrent-structural OOP-based modeling. For example, VHDL allows limited algorithmic specification of structure via its `generate` statements and supports static analysis but does not support any of the other needed capabilities such as polymorphism and algorithmic parameters. The Balboa [Doucet et al. 2002] modeling environment supports algorithmic specification and component overloading by running type inference at run-time. However, Balboa and its type inference algorithm do not support parametric polymorphism [Doucet et al. 2003]. As we will see in the next section, the Liberty Simulation Environment gains, in practice, the full benefits of both static structural modeling and concurrent-structural OOP modeling.

3.4 Control Abstraction

Up to this point, this section focused on identifying the reuse-enabling features present in existing systems. However, facilitating rapid accurate modeling requires that a system also provide mechanisms for simplifying parts of a design that do not benefit from reuse. As was mentioned earlier, the global nature of control in a hardware design makes the components that manage control unlikely candidates for reuse. Existing systems, in addition to lacking some features to enable low-overhead reuse, also lack abstractions to simplify the task of describing control. Since the majority of components in these systems are not reused, having abstractions to reduce control specification overhead seems unnecessary; implementing components from scratch is the norm. Conversely, since components built in the Liberty Simulation Environment are reusable, the need for control abstraction becomes more apparent. Section 6 will discuss the abstraction used in LSE.

4. THE LIBERTY SIMULATION ENVIRONMENT

LSE is a fully concurrent-structural modeling framework designed to maximize reusability of components while minimizing specification overhead. A user models a machine in LSE by writing a machine description in the Liberty Structural Specification language. This description specifies the instantiation of components, the customization of the flexible reusable components, and the interconnection of component ports. LSE includes a simulator generator that transforms this concurrent-structural machine description into an executable simulator and additional tool chains for other purposes, such as model visualization.

To avoid the mapping problem, LSE only allows components to communicate structurally, but this structure, along with component customizations, can be specified algorithmically via imperative programming constructs. Using these constructs, a model's structure can be built using code similar to the concurrent-structural OOP code shown in Figure 5. However, unlike modeling in concurrent-structural OOP, the LSS code only describes the model's structure and *not* its run-time behavior. Thus, as shown in Figure 6, the LSS description can be executed at *compile time* to generate the system's static structure. This allows model structure to be used for compile-time static analysis. Currently, LSE analyzes this structure to reduce specification overhead [Vachharajani et al. 2004a, 2004b] and to optimize the built simulator performance [Penry and August 2003].

Each component in a model built using LSE is instantiated from a component template, called a *module*, that is analogous to a class in a concurrent-structural OOP system. The body of an LSS module specifies a component's parameterization interface, communication interface, and constructor. There are two types of modules in LSS. The first, *leaf modules*, are simple modules defined without composing behavior from other modules. The other style, *hierarchical modules*, are more complex modules obtaining their behavior through the composition and customization of existing modules. The next two sections will describe leaf and hierarchical modules and their parameterization.

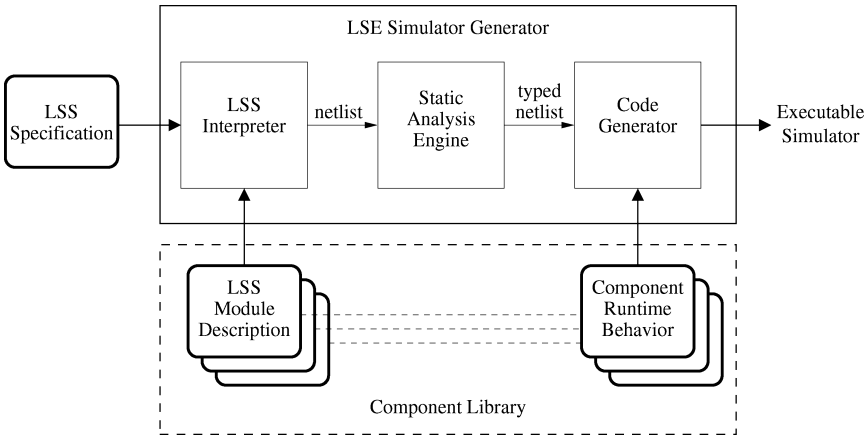


Fig. 6. Overview of the simulator generation process in LSE.

```
1  module delay {
2    parameter initial_state = 0:int;
3
4    inport in:int;
5    outport out:int;
6
7    tar_file="corelib/delay.tar";
8
9    // BSL specific parameters here
10 };
```

(a) LSS module declaration for a leaf delay element

```
1  instance d1:delay;
2  instance d2:delay;
3  ...
4  d1.initial_state = 1;
5  d1.out -> d2.in;
6  ...
```

(b) Sample use of the `delay` module

Fig. 7. Delay element declaration and use.

4.1 Leaf Modules

Leaf modules are simple modules whose behavior is externally specified. The module declaration is responsible for declaring the parameterization and communication interface of the module and for specifying where the module's behavior can be found. Figure 7(a) shows the declaration of a leaf module named `delay`. Line 2 in the figure declares a module parameter named `initial_state` with type `int` and assigns the parameter a default value of `0`. Lines 4 and 5 illustrate defining the communication interface of the module. These two lines define an input port named `in` and an output port named `out`, respectively, both with type `int`. Line 7 specifies where the code defining the run-time behavior of instances of this module can be found.

The code which defines a leaf module's behavior is not written in LSS, but a separate behavior specification language (BSL).[4] A leaf module's behavior code specifies how values arriving on input ports are combined with internal state to produce values on the instance's output ports. The module behavior code uses

[4]Currently, LSE uses a stylized version of C as the BSL, but the LSS language and the techniques presented in this article are not dependent on the specific BSL used.

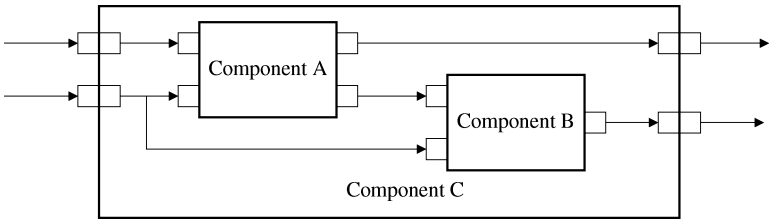


Fig. 8. Hierarchical component composition.

the user specified values of module parameters to customize the behavior of a particular instance.

Figure 7(b) shows an example of instantiating and parameterizing the `delay` module. Lines 1 and 2 each instantiate the `delay` module to create module instances named `d1` and `d2`, respectively. Line 4 gives the `initial_state` parameter on instance `d1` the value `1`. Line 5 connects the output of `d1` to the input of `d2`. Notice that the `initial_state` parameter on instance `d2` is not set. When such assignments are omitted, the parameter takes on its default value as defined in the module body (line 2 of Figure 7(a)).

Notice from the example that parameters in LSS are referenced nominally and can be specified *after* the instantiation statement (e.g., `initial_state` is referenced on line 4 of Figure 7(b)) rather than in an a positional argument list as part of the instantiation statement. These choices were made because flexible modules typically have many parameters. Nominal parameter references clarify models since parameter names describe the parameter's purpose better than position in an argument list. Similarly, flexible placement of parameter assignment allows groups of related parameter assignments for different module instances to be co-located rather than scattered based on where modules are instantiated. Both features make using flexible components (i.e., those with many parameters) easier, encouraging their construction and use.

4.2 Hierarchical Modules

In addition to leaf modules, LSS supports the creation of complex modules by composing the behavior of existing modules into new *hierarchical* modules. Hierarchical modules, just like leaf modules, define a parameterization and communication interface by declaring ports and parameters. However, unlike leaf modules, the behavior of the module is specified by instantiating modules and connecting these sub-instances to the new module's input and output ports (see Figure 8). These module subinstances execute *concurrently* and define the hierarchical module's behavior.

5. LOW-OVERHEAD REUSE IN LSE

This section gives an overview of LSE's features that allow for easy reuse of flexible components. When discussing LSE features, the text highlights challenges in the features' implementation. The details about technology developed to address these challenges is discussed in the literature [Vachharajani et al. 2004a, 2004b].

```
1  module delayn {
2    parameter n:int;
3
4    inport in:int;
5    outport out:int;
6
7    var delays:instance ref[];
8    delays=new instance[n](delay,"delays");
9
10   var i:int;
11
12   in -> delays[0].in;
13   for(i=1;i<n;i++) {
14     delays[i-1].out -> delays[i].in;
15   }
16   delays[n-1].out -> out;
17 };
```

(a) The LSS module declaration

```
1  instance gen:source;
2  instance hole:sink;
3  instance delay3:delayn;
4
5  delay3.n=3;
6
7  gen.out -> delay3.in;
8  delay3.out -> hole.in;
```

(b) Use of delayn, $n = 3$

Fig. 9. n-stage delay chain declaration and use.

5.1 Structural Parameters

Recall from Section 3 that, to fully enable reuse, a modeling system needs to support parameters that control the structure of hierarchical modules. LSS allows the use of imperative control flow constructs to guide the subcomponent instantiation, parameterization, and connection. *Any* parameter can be used to control these constructs; therefore all LSS parameters can be used as structural parameters.

To see how a parameter can be used to control structure, consider the LSS code shown in Figure 9(a). This code defines a module that models an arbitrary depth delay pipeline (Figure 4(b)) built using single-cycle delay modules. The module `delayn` declares a single parameter `n` (line 2) which controls the number of stages in the pipeline. Anywhere after this declaration, the body of the module can read this parameter to guide how subinstances will be created, connected, or parameterized.

Lines 7 and 8 create an array of instances of the `delay` module that will be named `delays` in the BSL. Notice that the length of the array (the value enclosed in brackets on Line 8 of the figure) is controlled by the parameter `n`.

Lines 12 through 16 connect the `delay` instances in a chain as shown in Figure 4(b). Notice how the general purpose C-like for-loop causes the length of the connection chain to vary with the parameter `n`.

Figure 9(b) shows how the `delayn` module can be used to create a three-stage delay pipeline. The module is instantiated on line 3, its `n` parameter is set on line 5, and finally the instance is connected on lines 7 and 8. The block diagram of the resulting system is the same as in Figure 4(a).

5.2 Extending Component Behavior

Section 3 also stated that a system that supports reuse must support algorithmic parameters to allow an existing component's behavior to be extended or augmented. In LSE, these algorithmic parameters are called *userpoints*. Userpoints accept string values whose content is BSL code that forms the body of

a function invoked by a module's behavioral specification to accomplish some computation or state-update task. The function signature, the arguments it receives and the return type it must produce, is defined by the data type of the userpoint. Just like other parameters, userpoint parameters can have default values, thus allowing the module to define default behavior which can be overridden by the user.

In concurrent-structural OOP systems, inheritance takes the place of algorithmic parameters. Just like algorithmic parameters, inheritance allows a component's behavior to be modified or extended. However, a single userpoint parameter assignment on a module instance is the concurrent-structural OOP equivalent of inheriting a class, overriding a virtual member function, and then instantiating the inherited class. Thus userpoints dramatically reduce the overhead of one-off inheritance (i.e., inheriting a module and instantiating it once). Since one-off inheritance is common in structural modeling, using userpoints rather than inheritance reduces specification overhead. More formal styles of inheritance can be achieved via userpoint assignment and hierarchical module construction.

To allow userpoints to maintain state across multiple invocations, LSS also allows the state of a module instance to be extended. State is added by declaring *run-time variables* (i.e., variables available during simulation rather than during model compilation). To allow this state to be initialized and *synchronously* updated, LSE provides on every instance the predefined userpoints `init` and `end_of_timestep`, which are invoked at the beginning of simulation and the end of each clock cycle, respectively.[5]

5.3 Flexible Interface Definition

To maximize the flexibility of components, LSS extends parametric control of structure to include parametric control of interfaces as well. A common use of this facility is parametric control of interface size, such as the number of read ports on a register file. However, as will be seen, this customization can control any portion of the module's interface.

5.3.1 *Flexible Interface Size.* To facilitate scalable interfaces such as a register file with a customizable number of read ports, each port in LSS 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. For each port in a module, the port's width (the number of connections made to the port) is available as a parameter for use in a module's body. These width parameters are automatically set by counting the number of connections actually made to a particular port. This automatic inference of port width greatly simplifies specifications. Without this inference, users would have to manually keep the width parameters consistent with the connections. This process would be prone to error, and fixing the errors would be tedious, time-consuming, and unnecessary.

[5]These userpoints are altered when modeling multiple clock domains in LSE.

```
1  module delayn {
2    parameter n:int;
3
4    inport in:'a;
5    outport out:'a;
6
7    if(in.width != out.width)
8      punt("in.width must match out.width");
9
10   var delays:instance ref[];
11   delays=new instance[n](delay,"delays");
12   var i:int;
13
14   /* The LSS_connect_bus(x,y,z)
15    * built-in does:
16    *
17    *   for(i=0; i<z; i++) { x[i]->y[i]; }
18    */
19   LSS_connect_bus(in,delays[0].in,in.width);
20   in -> delays[0].in;
21   for(i=1;i<n;i++) {
22     LSS_connect_bus(delays[i-1],
23                     delays[i].in,in.width);
24   }
25   LSS_connect_bus(delays[n-1],out,in.width);
26 };
```

(a) Modified `delayn` module that supports multiple port connections

```
1  instance gen:source;
2  instance hole:sink;
3  instance delay3:delayn;
4
5  delay3.n=3;
6
7  LSS_connect_bus(gen.out,
8                  delay3.in, 5);
9  LSS_connect_bus(delay3.out,
10                 hole.in, 5);
```

(b) Use of the modified `delayn` module

Fig. 10. Modified `delayn` module and a sample use.

Figure 10 illustrates how one would use these scalable interfaces to build a hierarchical module, and it also demonstrates how a port's width parameter is automatically set. Recall the `delayn` module presented in Figure 9. While the `delay` module (which was used to build the `delayn` module) supports multiple connections to its `in` and `out` ports, the `delayn` module internally connects only one port instance to the chain of `delay` modules. If a connection were made to more than one port instance of either port on the `delayn` module, it would be ignored since it is internally unconnected.

Figure 10(a) shows the `delayn` module extended to support connections to multiple port instances, and Figure 10(b) shows a sample use of the module. Notice that many connections are now made from the the `in` port to the head of the delay chain (line 19 of Figure 10(a)), between delay elements in the chain (lines 22–23), and finally from the tail of the chain to the `out` port (line 25). Further, notice that the number of connections made is controlled by the parameter `in.width`, yet this parameter has no explicit default value or user assignment. Instead, its value is inferred by the system based on the number of connections made. In this example, since five connections are made to the `in` port (lines 7–8 of Figure 10(b)), the parameter would have the value 5.

5.3.2 *Parameterized Interface Definition.* The inference of the width parameter described above is an example of an LSS feature called *use-based specialization*. This feature allows a module's context (its parametericity and connectivity) to alter its behavior and its interface. In the above example, only the widths of ports were varied, but use-based specialization can be used to

```
1  module ... {
2    inport in:'a;
3    outport out:'a'
4
5    if(out.width < in.width) {
6      parameter arbitration_policy:
7             userpoint( /* args */ =>
8                        /* ret */);
9      instance arb:arbiter;
10     arb.policy = arbitration_policy;
11     ...
12   } else {  ...  }
13   ...
14 };
```

Fig. 11. Use-based specialization exporting additional parameters.

alter *any* piece of the a module's interface. For example, by detecting whether a `branch_target` port is connected, a branch prediction module can infer whether or not it should also implement BTB (branch target buffer) functionality. If BTB functionality is necessary, the component can export additional parameters and ports to further customize this behavior.

To see how use-based specialization can affect a hierarchical component's interface and structure, consider the code in Figure 11. Here the module infers whether an internal arbiter is necessary by comparing the width of its input port to that of its output port. If the input port is wider than the output port, an arbiter is instantiated, and a userpoint parameter is exported so that the arbitration policy can be parametrically specified.

Notice that the module's interface can change *after* it has been instantiated and used. In the example, the module's connectivity, which is determined after instantiation, controls whether the module will have a certain parameter. Without use-based specialization, the module's interface would be fixed at instantiation, and the `arbitration_policy` parameter would always exist. If the parameter has no default value, then the user would be forced to set it, even when no arbitration is necessary. Alternatively, the parameter could be assigned a default value. However, since there are many possible default arbitration policies, having the module quietly make this important design decision when widths are changed is undesirable. While this is less severe than the problem illustrated in Figure 3 since the arbiter is explicitly created and its behavior determined only by microarchitectural state, a design decision is nonetheless being made without user knowledge. Use-based specialization makes deciding whether the parameter ought to have a default value unnecessary by providing the best of both worlds; the user must provide the policy when it is necessary and is not forced to provide it when it is not.

While use-based specialization reduces the overhead of using flexible components by automatically tailoring components to their environment, it introduces complications into the execution of LSS. Since a module's parameterization and connectivity can affect its interface, the module's interface is not known until after its parameters have been set and its ports connected. However, this parameterization and connection relies on the interface being known. To resolve this apparent circularity, LSS uses novel evaluation semantics that are described in the literature [Vachharajani et al. 2004a].

5.4 Polymorphic Interfaces

In addition to flexible interface definitions, LSE also supports modules with polymorphic interfaces. LSS supports two types of polymorphism: parametric polymorphism and component overloading.

5.4.1 *Parametric Polymorphism.* Parametric polymorphism allows for the creation of data type independent components. This feature is particularly useful for reusable communication primitives like routers, arbiters, and filters, and for reusable state elements like buffers, queues, and memories.

As an example, recall the `delayn` module shown in Figure 9(a). As shown, the `delayn` module can only handle the `int` data type since the ports are created with type `int` (lines 4 and 5 in Figure 9(a)). However, the behavior of the module, creating a delay pipeline that is n stages deep, is independent of data type. Consequently, the `delayn` module is an ideal candidate for using parametric polymorphism. To make the module parametrically polymorphic, rather than making the `in` and `out` ports have the data type `int`, one would declare the ports' types using type variables as shown below:

```
4  inport in:'a;
5  outport out:'a;
```

The type `'a` is a type variable (all type variables in LSS begin with a `'`) which can be instantiated with any LSS type. This flexibility makes the modified `delayn` module data-type-independent.[6] Since the `in` and `out` ports use the same type variable, both ports must have the same concrete type. This guarantees that the type of data entering the delay pipeline is consistent with the type of data that comes out. While this example demonstrates parametric polymorphism on a hierarchical component, it can also be used on leaf components. In such cases, the BSL code for the leaf component is specialized based on the concrete type given to all type variables.

5.4.2 *Component Overloading.* Component overloading is useful when defining a component that supports more than one data type on a particular port, but needs to be customized based on which type is actually used. Component overloading is supported with *disjunctive types*. A disjunctive type, denoted as `type1 | type2` in LSS, specifies that the entity with this type may statically have type `type1` or `type2`, but not both simultaneously.[7] Depending on which type is actually selected, a different module implementation will be selected.

As an example of component overloading, consider trying to build an ALU component that could be used as either an integer ALU or a floating-point ALU depending on how it is instantiated. The interface for such an overloaded ALU component is shown in Figure 12. Each port of the ALU is defined using the

[6]This modification to `delayn` assumes that the `delay` module also uses parametric polymorphism. The `delay` module defined in the LSE core module library does, in fact, support this.

[7]Note that the disjunctive type is *different* from union types in other programming languages. Union types dynamically store data of *any* of the enumerated types rather than data of a single, statically selected type.

```
1  module ALU {
2    inport in1:(int | float)
3    inport in2:(int | float)
4
5    outport result: (int | float)
6
7    constrain ''in1 == ''in2;
8    constrain ''in1 == ''result;
9    ...
10 };
```

Fig. 12. An overloaded ALU module interface.

disjunctive type `int | float` (lines 2, 3, and 5 of Figure 12). Lines 7 and 8 in this example force all the ports of the component to have the same type (″`in1`, ″`in2`, and ″`result` are type variables corresponding to the types of their respective ports). Depending on the type selected, the appropriate ALU behavior (integer or floating point) will be used when this ALU is instantiated.

Since modules may define multiple ports with disjunctive types, and not all ports with disjunctive types will be constrained to be equal, a naïve implementation of component overloading would require implementing the full cross-product of allowable overloaded configurations for a particular overloaded module. Creating all these implementations for a module with many overloaded ports may be extremely cumbersome. However, since in LSE the types are resolved statically, rather than implementing multiple *entire* behaviors for a given component, the BSL can specify type-dependent code fragments. The code generator can customize this code using the statically resolved type information, and then combine it with a module's type-independent code.

5.4.3 *Type Inference.* In order to reduce the designer's overhead in using polymorphic components, polymorphism in LSS is resolved via type inference based on the structure of the model. For example, if the `in` port of the polymorphic `delayn` module constructed above were connected to a module which outputs values of type `int`, the type variable ′`a` would be resolved to have type `int`. Due to the presence of disjunctive types, however, implementing this inference is not straightforward. Algorithms found in the literature are not appropriate to solve the LSS type inference problem. The problem is also NP-complete which suggests that it may be prohibitively expensive to implement. Fortunately, LSS uses a heuristic inference algorithm that keeps compile times reasonable. Details regarding the LSS type inference problem, a proof of its NP-completeness, and details of the heuristic type-inference algorithm can be found in the literature [Vachharajani et al. 2004a].

5.5 Instrumentation

To separate model specification from model instrumentation, as was possible in static structural modeling, LSE supports an aspect-oriented data collection scheme. Each module can declare that its instances emit certain *events* at run-time. These events behave like join points in aspect-oriented programming (AOP) [Kiczales et al. 1997]. Each time a certain state is reached or a certain value computed, the instance will emit the corresponding event. User-defined *collectors* fill these join points and collect information for statistics

```
1  module Pipeline {
2    /* Other pipeline stages */
3    ...
4    /* Writeback stage */
5    instance bus_arbiter:arbiter;
6    instance cdb_fanout:tee;
7    instance ALU_fanout:tee;
8
9    bus_arbiter.comparison_func =
10   <<< if(instr_priority(data1) >=
11          instr_priority(data2))
12         return 0;
13       return 1;
14   >>>;
15
16   /* ALU to LSU bypass */
17   ALU.out -> ALU_fanout.in;
18   ALU_fanout.out[0] -> LSU.store_operand;
19
20   ALU_fanout.out[1] -> bus_arbiter.in[0];
21   FPU.out -> bus_arbiter.in[1];
22   LSU.out -> bus_arbiter.in[2];
23
24   bus_arbiter.out -> cdb_fanout.in;
25   /* Fan out to the reorder buffer */
26   cdb_fanout.out[0] -> rob.instr_wb;
27   /* Fan out to reservation stations */
28   for(i=0;i<n;i++) {
29     cdb_fanout.out[i+1] -> res_station[i].instr_wb;
30   }
31   ...
32 }
```

Fig. 13. An LSE specification of the writeback stage of the machine shown in Figure 3, with an additional connection from the ALU to the LSU.

calculation and reporting. BSL code may be specified for the collector that processes the data sent with the event to accumulate statistics, to allow model debugging, and to drive visualization.

In addition to defined events, LSS automatically adds an event for each port. This event is emitted each time a value is sent to the port. Since important hardware events are often synchronized with communication, many useful statistics can be gathered using just these port firing events.

5.6 Putting It All Together

Figure 13 shows a sample LSS description of the writeback stage of the machine shown in Figure 3(c) with an additional connection directly from the ALU to the load-store unit (LSU). The primary module in the writeback stage is the common data bus arbiter, instantiated on line 5 in the figure. For reasons discussed in Section 6, connections in LSE are always point-to-point. Thus, two `tee` modules need to be instantiated (lines 6 and 7) to handle the common data bus net and the ALU output net because these nets must fan out. Lines 17–30 connect the various ports on the modules to the appropriate ports on other modules. Notice how the variable sizing of the port interface is used to allow the `arbiter` module instance to accept an arbitrary number of inputs and the `tee` instances to fan out an arbitrary number of signals. Internal widths for the arbiter's ports are inferred based on these connections by LSE.

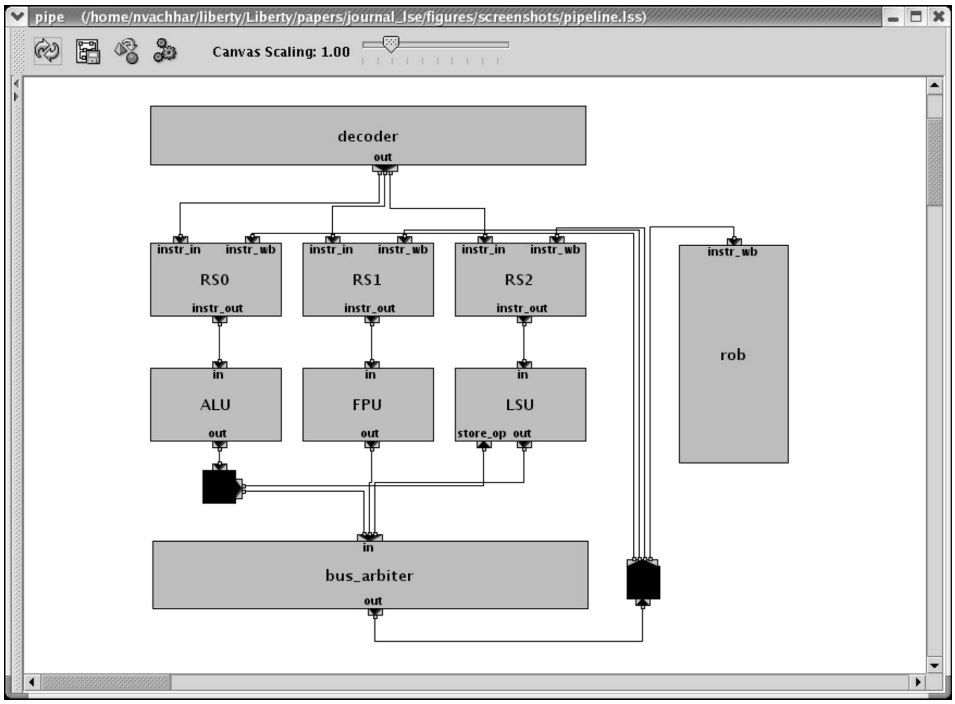


Fig. 14. Screenshot of the LSE Visualizer showing the model from Figure 13.

LSE's userpoint parameters are used in the above example to set the arbitration strategy that the `bus_arbiter` instance will use to arbitrate the common data bus. The standard arbiter module in the library does pairwise arbitration, and thus we need only provide a function to arbitrate between a pair of inputs (lines 9–14). Note that the standard `arbiter` and `tee` module will work with any type, since they are polymorphic. The types are resolved automatically based on the types of the ALUs and other connected components.

Clearly, this LSE description directly corresponds to the structure of the hardware in terms of hardware blocks and interconnections. Since the LSS language is evaluated at compile time, the structure of the described machine can be visualized in a graphical tool, as shown in Figure 14, and can be analyzed so that the generated simulators may be optimized [Penry and August 2003]. Note that this description is constructed using only components from the LSE standard module library, highlighting the utility and reusability of the standard components.

Figure 15 illustrates module extension mechanisms with modifications to this model. Suppose that a round-robin arbitration policy was needed in the example instead of the shown priority arbitration scheme. A round-robin arbitration policy requires that the arbiter alternate which input has the highest priority. To store which input currently has highest priority, the `bus_arbiter` instance needs to be augmented with additional state.

On line 1 of Figure 15, an instance of a general arbitration module is created. To customize this instance with a round-robin arbitration policy, the instance

```
1  instance bus_arbiter:arbiter;
2  var index:runtime_var ref;
3
4  index = new runtime_var("index",int);
5
6  bus_arbiter.init = <<< ${index} = 0; >>>;
7
8  bus_arbiter.end_of_timestep = <<<
9    ${index} = (${index}+1)%LSE_port_width(in);
10 >>>;
11
12 bus_arbiter.comparison_func = <<<
13   int dist1,dist2;
14   int WIDTH = LSE_port_width(in);
15   dist1 =(port1 + WIDTH - ${index}) % WIDTH;
16   dist2 =(port2 + WIDTH - ${index}) % WIDTH;
17   return (dist2 < dist1);
18 >>>;
```

Fig. 15. Customization of an `arbiter` module for round-robin arbitration.

```
1  collector out.resolved on bus_arbiter {
2    ...
3    record = <<<
4      ...
5      printf(LSE_time_print_args(SIM_time_now));
6      printf(": Message:\n")
7      print_cdb_data(*datap)
8      printf("Won arbitration this cycle");
9    >>>
10 }
```

Fig. 16. Collector to monitor bus arbiter output.

will need to have state indicating which input has the highest priority. The state is added to the instance by creating a run-time variable. Line 2 declares an LSS variable to hold a reference to this run-time variable and line 4 instantiates a new integer run-time variable whose BSL name is `index`.

Line 6 sets the value of the `init` userpoint so that the `index` run-time variable will be initialized to 0. The `${index}` notation allows a reference to the run-time variable to be embedded into the code quoted with the <<<...>>> characters. Similarly, lines 8–10 set the value of the `end_of_timestep` userpoint so that `index` will be incremented at the end of each clock cycle, wrapping around to 0 when it reaches the maximum number of inputs (which is the width of the arbiter's `in` port). Finally, the code that implements the arbitration policy is assigned to the `comparison_func` userpoint on lines 12–18. The function computes the distance of requested ports from each `index`, and selects the input which is closest.

The instrumentation features of LSS can be used to emit the data transmitted by the arbiter in each cycle to check if the round-robin arbitration code is correct. A sample data collector for the output of the bus arbiter is shown in Figure 16. The first line states that this collector should be activated any time a signal on the `out` port of the `bus_arbiter` instance is resolved. Lines 3–9 specify a fragment of code that executes each time the event occurs. Here, the code simply prints out the current cycle number (line 5), followed by the actual data (lines 6–8). The code to print the common data bus (CDB) data is in the function `print_cdb_data`. This is arbitrary code provided by the author of the model.

More information on the details of collectors, the syntax and semantics of the LSS language, and the BSL can be found in the LSE release documentation [The Liberty Research Group 2003].

6. CONTROL ABSTRACTION IN LSE

LSE provides a set of features to enable easy construction and use of flexible reusable components in concurrent-structural descriptions. However, even with ideal component-based reuse, several daunting challenges still remain for designers building and modifying hardware models. In particular, component-based reuse does little to assist in building the components that implement timing and stalling control logic in complex systems. The timing controller's correct operation is based on a global understanding of the datapath and the way that different events in the system are correlated. For example, the pipeline stall logic in a microprocessor is aware of structural hazards, and it stalls various parts of the pipeline when there are insufficient resources for computation to proceed. If any part of the datapath is changed, the controller must be altered to take those changes into consideration. In general, the precise details of why, when, and what to stall are heavily dependent on the particular design, and even minor variations can radically affect control. The controller is connected to many parts of the system, and these connections and interfaces must be managed explicitly. This tight intertwining of control and datapath makes creating a single, easily customized, and flexible stock component impractical if not impossible.

Although the controller cannot be broken up into components, control can still be separated into two parts. First, there is the portion of control that determines when to stall. Second, there is the portion of the controller that distributes the stall signal to all system components affected by a certain stall condition.

The generation of stalls can be further subdivided into structural stalls and semantic stalls. A buffer running out of space is an example of a structural stall. In this example, it is likely that all operations that require sending data to the buffer in the present cycle will need to stall. Semantic stalls, on the other hand, require understanding the overall function of the system. For example, to generate stalls due to data hazards in a processor pipeline, it is necessary to understand the detailed semantics of the instruction set architecture.

Since LSE is designed as a general-purpose tool, LSE does not provide a mechanism to specify a semantic description of the system. Therefore, little can be done to avoid specification of semantic stalls, though additional tools can be layered on top of LSE to provide this functionality. On the other hand, the portions of the controller that generate structural stalls can be handled by integrating this functionality into the components themselves. For example, buffers are aware of when they are full, and therefore can generate stall signals autonomously. Since the components themselves are reusable, the portion of the control logic that generates these structural stall signals does not need to be specified by the user of these components.

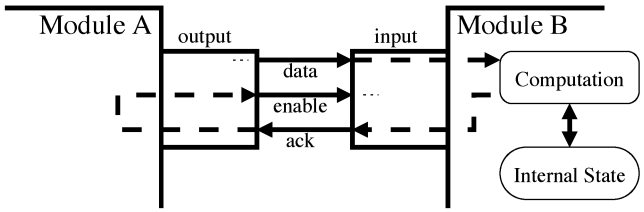


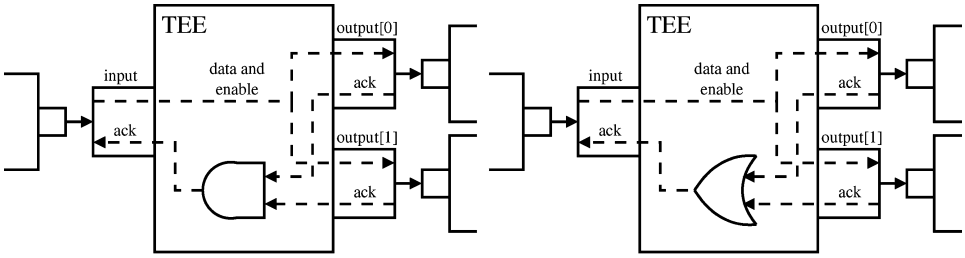
Fig. 17. Connection with standard control flow semantics.

To distribute stall information, regardless of whether it is structural or semantic, LSE exploits the connectivity of the components in the system. Components that generate stalls can communicate this information to their upstream neighbors. Those that do not need or do not generate stall information pass the downstream information to their upstream neighbors. Asynchronous hardware is a class of design that exploits this exact principle. Since there is no global clock on which a global controller can synchronize in asynchronous designs, stall information and stall signals are generated locally and then transmitted back to previous stages of a pipeline. By employing a similar strategy, the portion of global control that distributes stall information can be simplified.

Analogous to asynchronous hardware, each connection in an LSE description actually specifies three subconnections as shown in Figure 17: a `DATA` and an `ENABLE` signal in the forward direction and an `ACK` signal in the reverse direction. As its name implies, the `DATA` subconnection carries the data from the sending module (Module A in the figure) to the receiving module (Module B). The `ENABLE` signal, when asserted, indicates that the transmitted data should be used to perform state update. Finally, the `ACK` signal indicates that the receiving component is able to accept and process the sent data.

In a typical communication, like the one shown in Figure 17, the sending module (Module A) will send data on its output port. The receiving module (Module B) will determine whether or not it can accept the data and generate the corresponding `ACK` message. If the receiving module indicates that the data has been accepted, then the sending module will indicate that the receiver should use the data for state update by raising the `ENABLE` signal. Otherwise the sending module will indicate that the data should not be used by lowering the `ENABLE` signal. In this way, modules that cannot process a request can send a negative acknowledgment, creating a stall. Other components in the system will propagate this stall back along the datapath until the module that generated the request determines how to proceed, usually by sending a disable signal and retrying the request in the next cycle.

The three-way handshake described above can also be used to coordinate stalls between more than just a linear chain of modules. The `tee` module, for example, uses the handshake mechanism to coordinate the `ACK` signal of all the downstream recipients. The `tee` module forwards its incoming `DATA` and `ENABLE` signals to all its output ports, as shown in Figure 18(a). By default, it takes the incoming `ACK` signal from all of its outputs, computes the logical AND of these values, and passes that result out through the `ACK` wire on the input port. With these semantics, a module connected to the input of a `tee`



(a) Default semantics (b) Modified semantics

Fig. 18. The `tee` module's control semantics options.

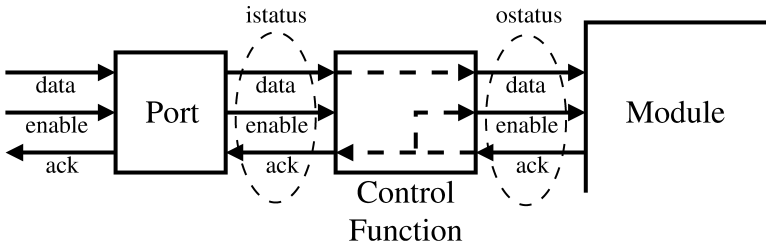
will only get an affirmative `ACK` signal if *all* modules downstream of the `tee` can accept the data. By modifying a parameter, the `tee` module's behavior can be changed so that it computes the logical OR of the incoming `ACK` signals to generate the outgoing `ACK` signal. This behavior is shown in Figure 18(b). With these new semantics, the sending module will see an affirmative `ACK` if *any* module downstream can accept the data. Thus, using the three-way handshake, we are able to implement useful control semantics for fan-out without difficulty. Since control can be handled in a variety of ways during fan-out, all LSE connections are point-to-point with explicit fan-out modules.

To automatically propagate stalls and automatically generate structural stalls, modules had to make assumptions about when stalls should be generated and how they should be propagated. While these defaults are normally correct, this is not always the case. LSE provides two mechanisms to allow these defaults to be modified and to allow semantic stalls to be injected.

The most obvious mechanism is to insert a new module which modifies the control signals to effect a stall. A custom module can be written from scratch; however, the LSE standard library provides several modules that can be used to help inject semantic stalls. These library modules accept input from components designed to detect semantic stall conditions, and they manipulate the control signals so that the stalls can propagate normally. While this mechanism is the most general, many common situations requiring only slightly modified control can be handled by *control points*, the other mechanism provided by LSE.

Each port in LSE defines a *control point* that can optionally be filled with a *control function* that modifies the behavior of the signals on the corresponding ports. As illustrated in Figure 19, one may think of the control function as a filter situated between a module instance and the instance's port.

To understand how control functions can be used in practice, recall the example from Figure 13. Assume the designers wish to use the ALU-to-LSU connection to forward store operands to the LSU before the ALU wins arbitration for the common data bus. By default, if the ALU fails to obtain the common data bus, the arbiter will send a negative `ACK` to the `ALU_fanout` tee causing the LSU's `store_operand` port to receive a low `ENABLE` signal. This low `ENABLE` will inform the LSU to ignore any data sent to its `store_operand` port.




```
1  LSU.store_operand =
2          <<< return LSE_signal_extract_data(istatus) |
3                     LSE_signal_ack2enable(ostatus) |
4                     LSE_signal_extract_ack(ostatus);
5          >>>;
```

Fig. 19. Control function overrides standard control.

We will use a control function to change the behavior so that the LSU receives a high ENABLE signal on the store_operand port any time the LSU asserts the corresponding ACK signal.

The code shown in Figure 19 fills the control point on the LSU's store_operand port. The control function receives the status of all signals on the input side of the control function (to the left in the figure) in the istatus variable, and the status of all signals on the output side in the ostatus variable. The function's return value will be used to set the DATA, ACK, and ENABLE status on all the outgoing wires (DATA and ENABLE on the right and ACK on left side of the control function in the figure). These statuses indicate whether or not data is present and whether or not the ENABLE and ACK signals are asserted. The status for each of the three signals is stored using several bits in a status word. This particular control function passes the incoming DATA and ACK signals straight through without modification by using the LSE_signal_extract_data and the LSE_signal_extract_ack API calls, thus preserving the signals' original behavior. It moves the incoming ACK signal to the outgoing ENABLE wire by using the LSE_signal_ack2enable API call, thus attaining the desired behavior. From this example, we see that the control function is able to alter machine control semantics without requiring modules to be rewritten or new modules to be inserted.

7. EXPERIENCE WITH LSE

Up to this point, this article has described the key features of LSE that enable component-based reuse and rapid simulator construction. This section describes our experience with LSE's modeling speed and presents some data that quantifies the amount of component-based reuse we have observed.

7.1 Modeling Speed

To date, LSE has been used to model a variety of microarchitectures in a number of research groups. Within our own group, it has been used to model several machines including a IA64 processor core validated to within 5.4% of hardware [Penry et al. 2005], a PowerPC processor core, chip

Table V. Quantity of Component-Based Reuse

Model Name	Instances	Hierarchical Modules	Leaf Modules	Instance 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
Total	1141	51 (16)	39	12.68 (19.82)	80%	22

Note: 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 [Burger and Austin 1997]; D—an out-of-order processor core for IA64; E—two of the cores from D sharing a cache hierarchy.

multiprocessor models utilizing those cores, two Tomasulo-style machines that execute the DLX instruction set, and a model that is cycle-equivalent to the popular SimpleScalar [Burger and Austin 1997] `sim-outorder.c` sequential simulator.

Each model was built by a single student and took under 5 weeks to develop. Some models took far less time to build. For example, once one version of the Tomasulo-style machine was built, the second model could be constructed in under a day. The chip multiprocessor version of the IA64 model also took only a day or two 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 [Austin 1997]. For each of the models described, LSE's control abstraction and reuse features were critical in achieving these development times, as will be seen in later sections.

7.2 Quantity of Component-Based Reuse

Table V quantifies the reuse in each of the models discussed above. There is a good amount of reuse within each specification with each module being used 3–10 times on average. Furthermore, a significant percentage (73–89%) of instances come from modules in the LSE module library. Notice also that the numbers for specification A and B are quite conservative. To improve code clarity, 36 and 35 modules, respectively, exist solely to wrap collections of other components and took less than 5 min to write. The reuse statistics ignoring these modules, shown in parenthesis in the table, show greatly increased reuse per module.

Reuse across specifications is even more dramatic. Over all specifications, each module is used 12.68 times with 80% of instances coming from the module library. Neglecting the trivial wrapping modules, each module is used about 20 times, indicating considerable reuse.

The significant reuse described in the table is largely a result of LSE's features to reduce specification overhead. The SimpleScalar model, which was built before many of the LSS features were available, contains the largest number of nontrivial custom modules relative to the total size of the specification. All the other models, which were built after the LSS features described in this article were added, have far fewer nontrivial custom modules relative to their

size. This indicates that these LSS features are important for realizing reuse in practice.

8. CONCLUSION

This article presented the design and implementation of the Liberty Simulation Environment, a publicly available tool engineered to enable *rapid* construction of *accurate* models. LSE is a concurrent-structural modeling environment, which allows it to avoid the mapping problem. LSE makes reuse practical by employing several language techniques to reduce overhead. LSE simplifies specification of timing control through a novel control abstraction mechanism. Finally, LSE has an optimizer, enabled by careful selection of execution semantics, that yields a simulator as fast as other concurrent-structural approaches.

LSE's design was motivated through a careful analysis of existing modeling systems. Sequential simulators suffer from the mapping problem which makes the simulators difficult to build, validate, and modify; the mapping problem also precludes the reuse of validated components. Existing concurrent structural systems, which avoid the mapping problem, preclude component reuse in practice. Even with component reuse, specification of timing control is time-consuming and no other systems attempt to address this.

Results and experience show that LSE makes rapid modeling and component-based reuse practical. Furthermore, LSE's optimization techniques allow users to enjoy this benefit without suffering performance penalties when compared to other concurrent-structural modeling systems. These properties make LSE an excellent tool for high-level design space exploration.

ACKNOWLEDGMENTS

We thank Azmat Hussain, Vijay Pai, John Sias, Kees Vissers, Paul Willmann, Ram Rangan, and the entire Liberty Research Group for their suggestions and support during the development of LSE and this article. We also thank the participatants in the user study presented in this article. Finally, we thank the editors and reviewers for their insightful comments.

REFERENCES

AUSTIN, T. 1997. A user's and hacker's guide to the SimpleScalar Architectural Toolset (for toolset release 2.0). Go online to `http://www.cs.virginia.edu/~skadron/cs 654/slides/hack_guide.pdf`.

BLOME, J., VACHHARAJANI, M., VACHHARAJANI, N., AND AUGUST, D. I. 2003. The Liberty Simulation Environment as a pedagogical tool. In *Proceedings of the 2003 Workshop on Computer Architecture Education* (WCAE).

BURGER, D. AND AUSTIN, T. M. 1997. The SimpleScalar tool set version 2.0. Tech. Rep. 97-1342, Department of Computer Science, University of Wisconsin-Madison, Madison, WI.

CAIN, H. W., LEPAK, K. M., SCHWARTZ, B. A., AND LIPASTI, M. H. 2002. Precise and accurate processor simulation. In *Proceedings of the Fifth Workshop on Computer Architecture Evaluation Using Commercial Workloads*.

CHAREST, L. AND ABOULHAMID, E. M. 2002. A VHDL/SystemC comparison in handling design reuse. In *Proceedings of 2002 International Workshop on System-on-Chip for Real-Time Applications*.

COE, P., HOWELL, F., IBBETT, R., AND WILLIAMS, L. 1998. A hierarchical computer architecture design and simulation environment. *ACM. Trans. Model. Comput. Sim. 8*, 4 (Oct.), 431–446.

DESIKAN, R., BURGER, D., AND KECKLER, S. W. 2001. Measuring experimental error in microprocessor simulation. In *Proceedings of the 28th International Symposium on Computer Architecture* (ISCA). 266–277.

DOUCET, F., OTSUKA, M., SHUKLA, S., AND GUPTA, R. 2002. An environment for dynamic component composition for efficient co-design. In *Proceedings of the Conference on Design, Automation and Test in Europe* (DATE).

DOUCET, F., SHUKLA, S., AND GUPTA, R. 2003. Typing abstractions and management in a component framework. In *Proceedings of Asia and South Pacific Design Automation Conference* (ASP-DAC). 115–122.

EMER, J., AHUJA, P., BORCH, E., KLAUSER, A., LUK, C.-K., MANNE, S., MUKHERJEE, S. S., PATIL, H., WALLACE, S., BINKERT, N., ESPASA, R., AND JUAN, T. 2002. Asim: A performance model framework. *IEEE Comput. 0018-9162*, 68–76.

GIBSON, J., KUNZ, R., OFELT, D., HOROWITZ, M., HENNESSY, J., AND HEINRICH, M. 2000. 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). 49–58.

HALAMBI, A., GRUN, P., GANESH, V., KHARE, A., DUTT, N., AND NICOLAU, A. 1999. EXPRESSION. A language for architecture exploration through compiler/simulator retargetability. In *Proceedings of the European Conference on Design, Automation and Test* (DATE).

JANNECK, J. W., LEE, E. A., LIU, J., LIU, X., NEUENDORFFER, S., SACHS, S., AND XIONG, Y. 2001. Disciplining heterogeneity—the Ptolemy approach. In *ACM SIGPLAN 2001 Workshop on Languages, Compilers, and Tools for Embedded Systems* (LCTES).

KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J.-M., AND IRWIN, J. 1997. Aspect-oriented programming. In *Proceedings of the 11th European Conference for Object-Oriented Programming* (ECOOP). 220–242.

MISHRA, P., DUTT, N., AND NICOLAU, A. 2001. Functional abstraction driven design space exploration of heterogeneous programmable architectures. In *Proceedings of the International Symposium on System Synthesis* (1SSS). 256–261.

ÖNDER, S. AND GUPTA, R. 1998. Automatic generation of microarchitecture simulators. In *Proceedings of the IEEE International Conference on Computer Languages*. 80–89.

OPEN SYSTEMC INITIATIVE (OSCI). 2001. *Functional Specification for SystemC 2.0*. Available online at `http://www.systemc.org`.

PEES, S., HOFFMANN, A., ŽIVOJNOVIĆ, V., AND MEYR, H. 1999. LISA—machine description language for cycle-accurate models of programmable DSP architectures. In *Proceedings of the ACM/IEEE Design Automation Conference* (DAC). 933–938.

PENRY, D. AND AUGUST, D. I. 2003. Optimizations for a simulator construction system supporting reusable components. In *Proceedings of the 40th Design Automation Conference* (DAC).

PENRY, D. A., VACHHARAJANI, M., AND AUGUST, D. I. 2005. Rapid development of a flexible validated processor model. In *Proceedings of the 2005 Workshop on Modeling, Benchmarking, and Simulation* (MOBS).

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

SWAMY, S., MOLIN, A., AND COVNOT, B. 1995. OO-VHDL: Object-oriented extensions to VHDL. *IEEE Comput. 28*, 10 (Oct.), 18–26.

THE LIBERTY RESEARCH GROUP. 2003. Web site: `http://www.liberty-research.org/Software/LSE`.

VACHHARAJANI, M. 2004. Microarchitectural modeling for design-space exploration. Ph.D. dissertation, Department of Electrical Engineering, Princeton University, Princeton, NJ.

VACHHARAJANI, M., VACHHARAJANI, N., AND AUGUST, D. I. 2004a. 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* (PLDI). 195–206.

VACHHARAJANI, M., VACHHARAJANI, N., MALIK, S., AND AUGUST, D. I. 2004b. Facilitating reuse in hardware models with enhanced type inference. In *Proceedings of the 2004 Conference on Hardware/Software Codesign and System Synthesis* (CODES+ISSS).

VACHHARAJANI, M., VACHHARAJANI, N., PENRY, D. A., BLOME, J. A., AND AUGUST, D. I. 2002. Microarchitectural exploration with Liberty. In *Proceedings of the 35th International Symposium on Microarchitecture* (MICRO). 271–282.

XIONG, Y. 2002. An extensible type system for component based design. Ph.D. dissertation. Electrical Engineering and Computer Sciences, University of California, Berkeley, CA.

Received January 2004; revised October 2005; accepted October 2005