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

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 sequential language such as C or C++ is the most common method of producing a software simulation model[1]. Unfortunately, this methodology does not provide an *efficient* path to an *accurate*

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

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 20YY ACM 0000-0000/20YY/0000-0001 $5.00

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 with construction of these simulators is to reuse an existing, carefully constructed, and validated simulator for 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, the model conveys architectural ideas, is easy for designers to understand, and exposes model/design mismatches.

Unlike in the manually-coded simulator approach, reuse is 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 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 trade-off between the ease of *building* reusable components and the ease of *using* such components. In current systems, this trade-off 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 non-local 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 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.

---

simulator construction methodology.

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. Sections 8, 9, and 10 describe novel approaches needed to implement the techniques discussed earlier in the article. Finally, Section 11 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 sub-components 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 alternate 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

```
foreach simulation cycle
   do commit stage
   do writeback stage
   do ex stage
   do issue stage
   do fetch stage
```

(a) original main simulator loop

```
foreach simulation cycle
   do commit stage
   do writeback stage
   do ex stage
   do issue stage
   do fetch stage
   foreach instr[i]
      if instr[i].issued
         dequeue instr[i]
```

(c) modified main simulator loop

Issue Logic

```
instr[1..n] = first n instructions in queue
for each instr[i] that was fetched
   if instr[i] is a branch
      fetch source registers
      if branch unit is available
         compute the target PC
         issue instruction to the branch unit
         dequeue instr[i]
   if instr[i] is an ALU op
      fetch source registers
      If ALU is available
         issue instruction to the ALU
         dequeue instr[i]
```

(b) original issue stage

```
instr[1..n] = first n instructions in queue
for each instr[i] that was fetched
   if instr[i] is a branch
      fetch source registers
      if branch unit is available
         compute the target PC
         issue instruction to the branch unit
         instr[i].issued=TRUE
   if instr[i] is an ALU op
      fetch source registers
      If ALU is available
         issue instruction to the ALU
         instr[i].issued=TRUE
```

(d) modified issue stage

Fig. 1. Sequential simulator code.

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. 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 typical 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 pseudo-code 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 logical entity in the hardware into different 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 pre-validated 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 gives intuition as to 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 difficult to understand hardware policy. The sequential simulators used were a collection of 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].

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 our requirements. Subjects had to be very familiar with

Table I. Subject responses to the 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 SimpleScalar
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

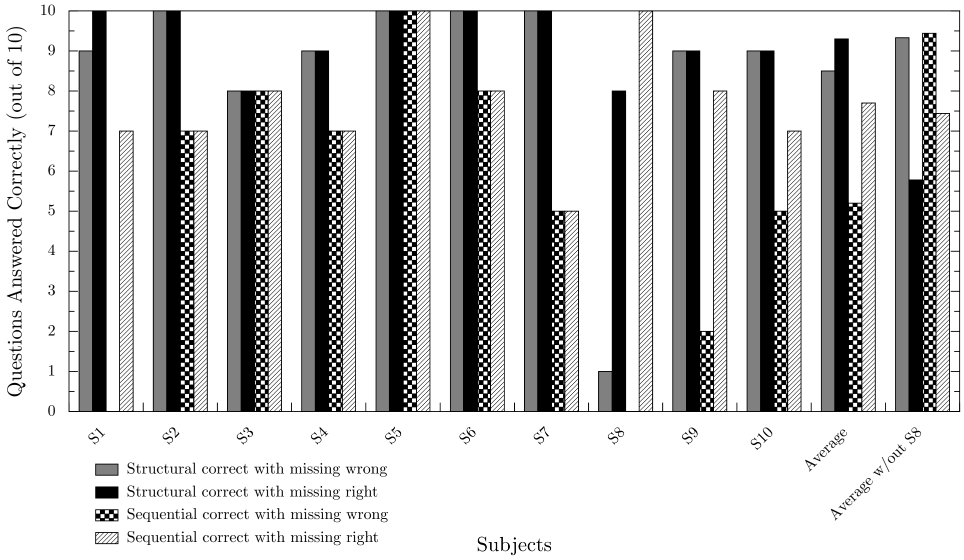
computer architecture and also had to be familiar with LSE. The second constraint proved to be the most difficult given that LSE is a relatively new system. Thus, only ten subjects were available for this experiment. As will be seen, however, even with ten subjects the results are quite dramatic.

Each subject was given 90 minutes to answer 2 multi-part questions for the sequential SimpleScalar simulator and the same questions for the structural LSE model along with 4 control questions. The control questions were asked first. Following these questions, each multi-part question for the sequential simulator was asked immediately after the same question for the structural model. The questions (with placeholders for line numbers and filenames) are shown in Appendix A[2].

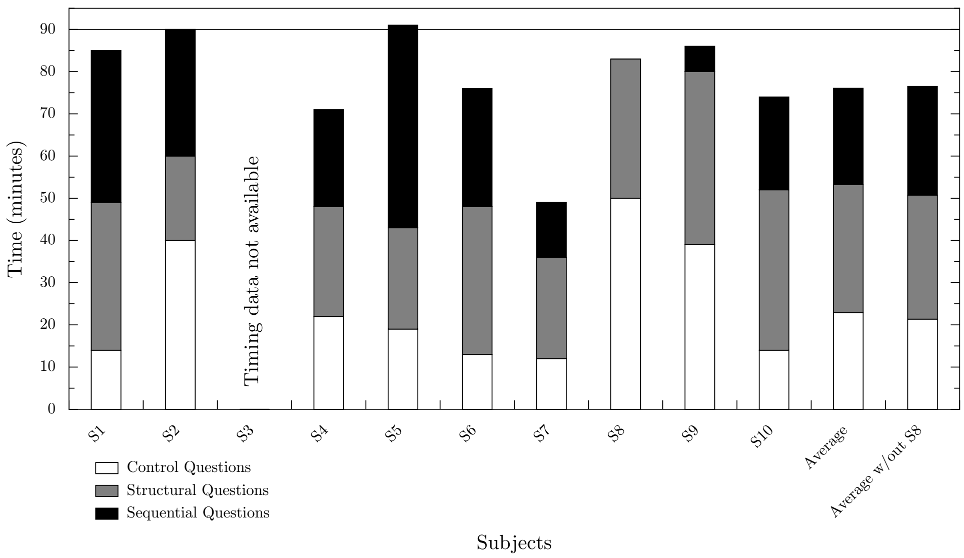
The control questions were used to determine if the subjects had basic familiarity with LSE since each had less than one week of experience with the tool. The answers to the control questions indicated that all subjects understood LSE enough for the purposes of this experiment.

Figure 2(a) summarizes the results. Since the subjects had limited time, not all subjects could answer all questions. The first pair of bars corresponds to responses regarding the structural model. The first bar in the pair assumes that all questions left unanswered were answered incorrectly. The second bar in the pair of bars shows the same data assuming that all unanswered questions were answered correctly. The second pair of bars shows the same information for questions regarding the sequential simulator. 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, usually much more accurately. On average, subjects answered 8.5 questions correctly with the structural model versus 5.2 for the sequential model. Even if unanswered questions are assumed correct for *only* the sequential simulator, almost all subjects answered more questions correctly with the structural model. The only subject who does better with the sequential simulator, under these circumstances, is subject S8, who 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 non-control question for the structural model (questions 1(a) and 1(b) as numbered in Appendix A) and no questions

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

for the sequential simulator. Clearly subject S8 is an outlier.

If subject S8 is excluded from the data, we see that, on average, 9.33 questions were answered correctly for the structural model versus 5.78 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 has 9.33 correctly answered questions versus 7.44 for the sequential simulator. These averages are statistically significant[3].

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 differs by only 30 seconds when compared to the time for questions about the sequential simulator. Therefore, the increased number of correct responses for the structural model is not due to substantially more time spent on those questions. Note that the total times are typically below 90 minutes since the time taken for any breaks was not counted. Subject S7 completed the questions extremely quickly, hence the short bar. Subject S3 failed to provide complete timing information.

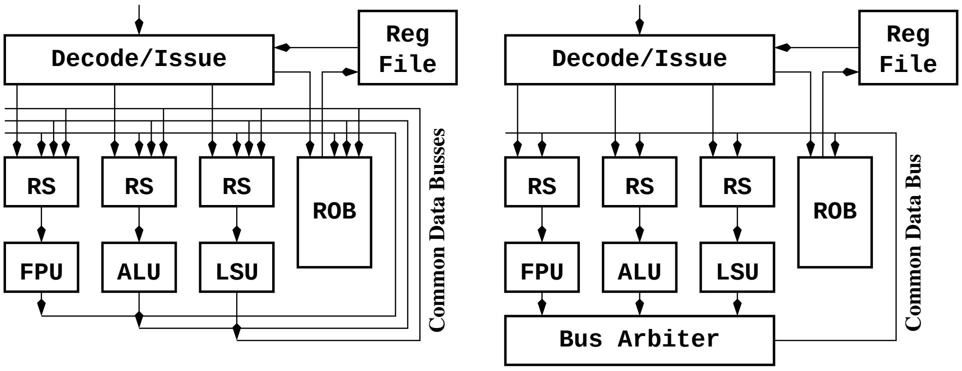
Note that the experiment is skewed in favor of the sequential simulator. First, since subjects were asked the LSE question immediately before being asked the same question for the sequential simulator, subjects could use the hardware-like model to understand what to look for in the sequential simulator. 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 to graphically view block diagrams of the structural specification. In our experience, 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 presents a block diagram of such a machine and 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,

[3]The probability of a Type I error (the p-value in statistics) is 0.004



(a) Original machine

(c) Limited write−back 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) Pseudo−code for original 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) Pseudo−code for limited write−back machine

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

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 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.

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

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 (LSE). The base LSE model and the base `sim-outorder` simulator [4] that were modified in the experiment had exactly matching pipeline traces. At the time this experiment was conducted the subject had little 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 are 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.

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

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 five hours 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. Some structural systems [Emer et al. 2002], however, still force functional composition for some inter-component 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

Table IV. Capabilities of existing methods and systems.

Capability	Static Structural Theory	Static Structural Practice	Concurrent-Structural OOP Theory	Concurrent-Structural OOP 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	

enable component-based reuse [Swamy et al. 1995]. These capabilities include:

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 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.

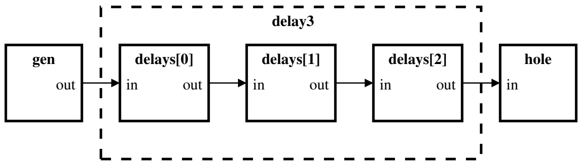
Parametric Polymorphism. The ability to create and use component models in a data type independent fashion. Examples: queues, memories, and crossbar switches that can store or process any data type.

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.

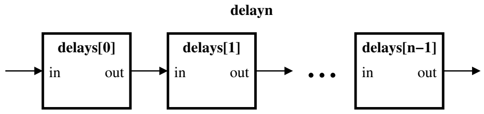
Static Analysis The ability to analyze the resulting concurrent-structural model for user convenience, verification, and simulator optimization. Example: type inference to automatically resolve polymorphic port types.

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



(a) Block diagram of a 3-stage delay chain specification.



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

Fig. 4. Block diagrams of chained delay components.

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.

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

```
class delayn {
  public InPort in;
  public OutPort out;

  Delay[] delays;
  delayn(int n) {
    int i;

    in=new InPort();
    out=new OutPort();

    delays=new Delay[n];
    for(i=0;i<n;i++) {
      delays[i]=new Delay();
    }

    in.connect(delays[0].in);
    for(i=0;i<n-1;i++) {
      delays[i+1]=new Delay();
      delays[i].out.connect(delays[i+1].in);
    }
    delays[n-1].out.connect(out);
  }
};
```

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

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 pseudo-code 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 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, this lack of static analysis prevents certain key optimizations that can increase simulator performance. These optimizations can provide as much as a 40% increase in simulator performance (see Section 10), eliminating performance loss due to reuse. In practice, simulator performance penalties combined with these 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 runtime. 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

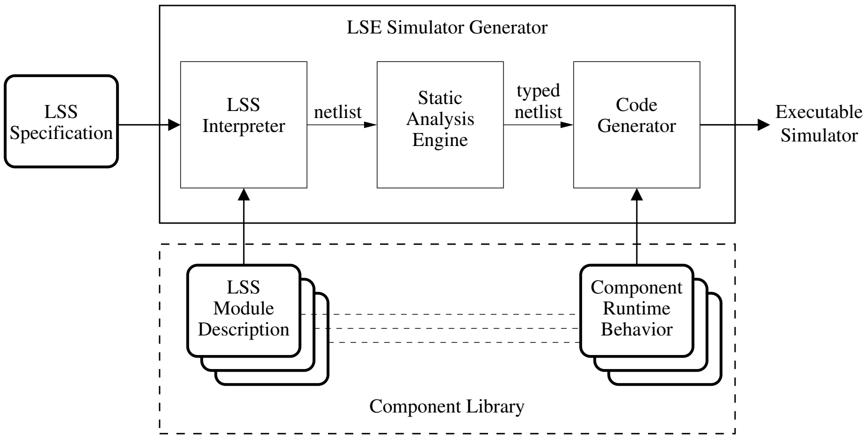


Fig. 6. Overview of the simulator generation process in LSE.

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

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 (LSS) 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 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 (described in Sections 8 and 9) and to optimize the built simulator performance (described in Section 10).

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.

```
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)[5]. 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 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.

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

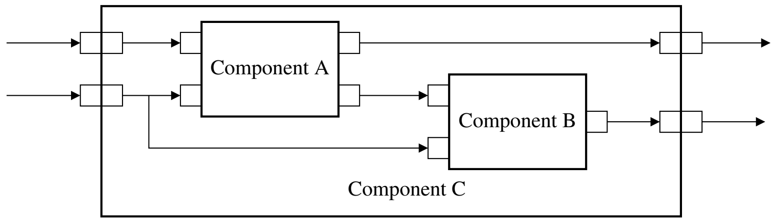


Fig. 8. Hierarchical component composition.

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 sub-instances 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. Later sections will discuss the details of technology developed to address these challenges.

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 sub-component 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 3-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

```
module delayn {
  parameter n:int;

  inport in:int;
  outport out:int;

  var delays:instance ref[];
  delays=new instance[n](delay,"delays");

  var i:int;

  in -> delays[0].in;
  for(i=1;i<n;i++) {
    delays[i-1].out -> delays[i].in;
  }
  delays[n-1].out -> out;
};
```

(a) The LSS module declaration.

```
instance gen:source;
instance hole:sink;
instance delay3:delayn;

delay3.n=3;

gen.out -> delay3.in;
delay3.out -> hole.in;
```

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

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

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[6].

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

```
module delayn {
  parameter n:int;

  inport in:'a;
  outport out:'a;

  if(in.width != out.width)
    punt("in.width must match out.width");

  var delays:instance ref[];
  delays=new instance[n](delay,"delays");
  var i:int;

  /* The LSS_connect_bus(x,y,z)
   * built-in does:
   *
   *   for(i=0; i<z; i++) { x[i]->y[i]; }
   */
  LSS_connect_bus(in,delays[0].in,in.width);
  in -> delays[0].in;
  for(i=1;i<n;i++) {
    LSS_connect_bus(delays[i-1],
                    delays[i].in,in.width);
 }
 LSS_connect_bus(delays[n-1],out,in.width);
};
```

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

```
instance gen:source;
instance hole:sink;
instance delay3:delayn;

delay3.n=3;

LSS_connect_bus(gen.out,
                delay3.in, 5);
LSS_connect_bus(delay3.out,
                hole.in, 5);
```

(b) Use of the modified `delayn` module.

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

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.

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.

```
module ... {
  inport in:'a;
  outport out:'a'

  if(out.width < in.width) {
    parameter arbitration_policy:
           userpoint( /* args */ =>
                      /* ret */);
    instance arb:arbiter;
    arb.policy = arbitration_policy;
    ...
  } else {  ...  }
  ...
};
```

Fig. 11. Use-based specialization exporting additional parameters

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 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 will be presented in Section 8.

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[7]. 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 simultane-

[7]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.

```
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.

ously[8]. 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 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 Section 9.

[8]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 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.

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 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 many important hardware events are 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

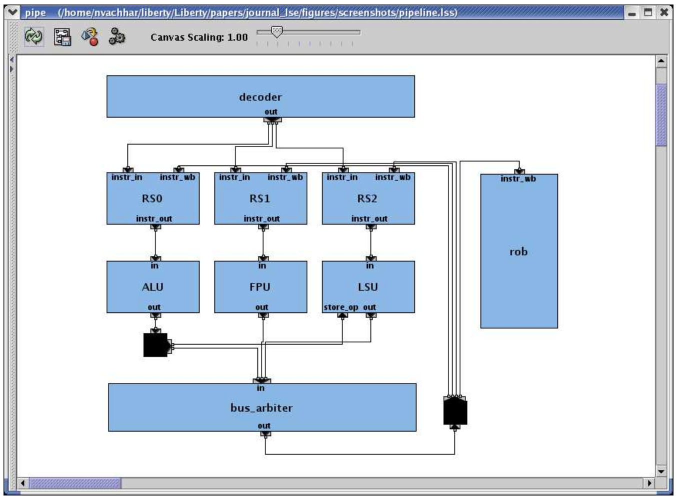


Fig. 14. Screenshot of the LSE Visualizer showing the model from Figure 13.

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.

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 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

```
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

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].

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.

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.

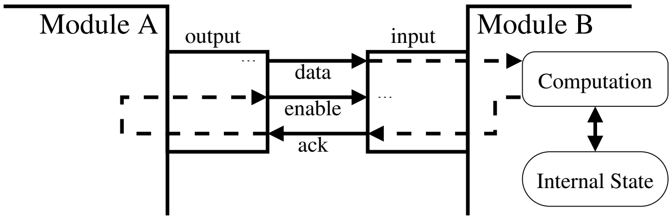


Fig. 17. Connection with standard control flow semantics.

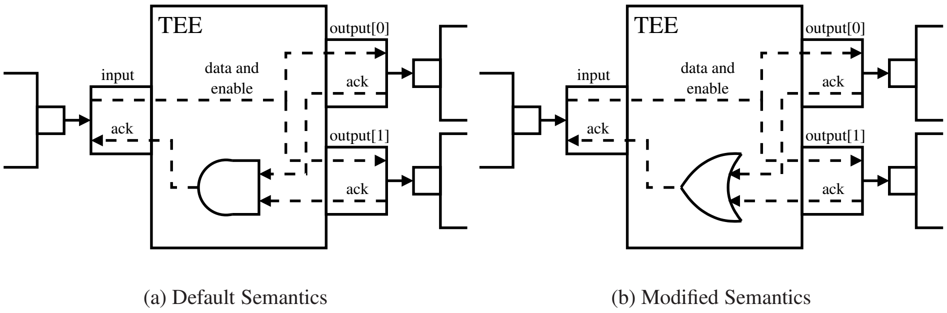
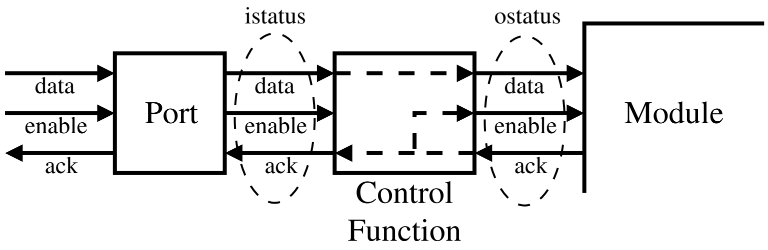


Fig. 18. The `tee` module's control semantics options.

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 3-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, com-




```
LSU.store_operand =
        <<< return LSE_signal_extract_data(istatus) |
                   LSE_signal_ack2enable(ostatus) |
                   LSE_signal_extract_ack(ostatus);
        >>>;
```

Fig. 19. Control function overrides standard control.

putes 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` 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 3-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. 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. Before moving on to the novel techniques needed to address the implementation challenges highlighted in earlier sections, a discussion of the effectiveness of LSE is in order. 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. Later sections in this article will evaluate the effectiveness of particular LSE features in isolation.

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 an IA64 processor core, a chip multiprocessor model that utilizes that core, two Tomasulo-style machines that execute the DLX instruction set, and a model that is cycle-equivalent to the popular SimpleScalar [Burger and Austin 1997] sim-outorder.c sequential simulator.

Each model was built by a single student, and each model 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 IA-64 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.

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

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.

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 minutes 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 a 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 non-trivial 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 non-trivial custom modules relative to their size. This indicates that these LSS features are important for realizing reuse in practice.

The next few sections will describe the novel techniques needed to implement LSE and its features for low-overhead reuse. In these sections, the features discussed will be evaluated on the above models to determine each feature's contribution to the reduction of specification overhead.

8. IMPLEMENTATION OF USE-BASED SPECIALIZATION

As was mentioned in Section 5.3, explicitly specifying values of all parameters can be cumbersome and often unnecessary. For example, the widths of ports can be inferred by analyzing the connectivity pattern of a hardware model. We call this process of components customizing themselves according to inferred parameter values *use-based specialization*. Use-based specialization can dramatically reduce specification time. However, implementing a modeling language compiler that infers component parameters from structure *and* customizes components using these inferred parameters requires novel language evaluation semantics. This section describes the implementation challenges raised by use-based specialization and presents the implementation of use-based specialization in LSE.

8.1 Evaluation Semantics

To support use-based specialization, a module instance's communication and parameterization interface must be programmatically determined rather than statically specified. This allows pieces of the interface to be present conditionally, as was demonstrated in Figure 11. Unfortunately, the code that defines a module instance's interface (the module body in LSS) depends on the values of parameters and connectivity of ports it defines. This circularity precludes the use of standard evaluation semantics in a system supporting use-based specialization.

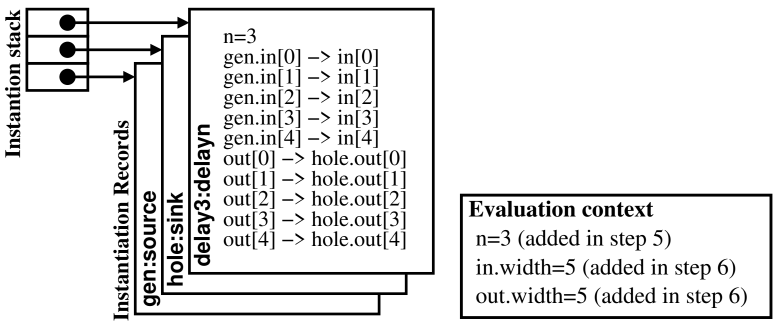
To resolve this circularity, the evaluation semantics of LSS defer the execution of an instance's module body until after parameterization and connectivity have been determined, rather than invoking it upon encountering the instance declaration. Evaluation of LSS statements occurs in sequence. Whenever an instantiation statement is reached, the identifier for the instance being created and the corresponding module are pushed onto an instantiation stack. Whenever a subsequent assignment to a subfield of a newly defined instance (e.g. `delay3.n = 3` in Figure 9(b)) is encountered, the assignment is recorded as a *potential parameter assignment*. Similarly, whenever any connection is made to a subfield of a newly created instance, the connection is recorded as a *potential port connection*.

At the time these recordings are made, the interfaces of all newly created instances are unknown since their module bodies have not yet been run. Therefore, the LSS interpreter does not know if the referenced subfields exist on the newly created instances. Further, even if the referenced subfields will exist, it is not known whether they were used correctly. For example, a port might have been incorrectly referenced as a parameter and vice versa. Therefore, when the module bodies for these newly created instances are executed, the validity of all potential parameter assignments and potential port connections must be checked.

After each block of statements finishes evaluation, the instance at the top of the instantiation stack for that block is popped off and the statements from its module body executed. Whenever a parameter declaration is encountered, the previously recorded potential parameter assignments are consulted to see if the parameter has a user specified value. If so, the type of the value is checked against the parameter's type and if the types match, the parameter is assigned that value. If no assignments were recorded, the parameter will get its value from default parameter assignments inside the module body, if they exist. Similarly, when a port is declared, the recorded list of connections is consulted to see if any attempts to connect to this port have been made. If so, the port is connected, and its implicit width attribute is set. After evaluation of the module instance completes, the potential parameter assignment and potential connection records are checked to make sure no nonexistent subfields were referenced and to ensure that no port was referenced like a parameter and vice-versa. Additionally, all the parameters are checked to ensure that they have some value.

The following example will illustrate the execution of the code shown in Figure 10(b).

(1) Line 1. The interpreter records that an instance of the `source` module named `gen` was created by pushing it onto the instantiation stack.



(a) Instantiation stack.

(b) Context for delay3.

Fig. 20. LSS interpreter state.

(2) Lines 2-3. The same is done for the `hole` and `delay3` module instances.
(3) Line 5. The interpreter records the assignment to potential parameter `n`.
(4) Lines 7-10. The interpreter records the connections made.

At this point, the top-level code has finished and so the module instance at the top of the instantiation stack is popped and its constructor evaluated. Figure 20(a) shows the instantiation stack in the LSS interpreter at this point in the execution. In this case, the next set of code to run is that for the `delayn` module shown in Figure 10(a).

(5) Line 2. The interpreter checks the potential parameter assignment record to see if `n` has been assigned a value, and if so, to verify that the type of the assignment was an integer. If so, the interpreter adds it to the evaluation context based on the value in the record. The evaluation context is shown in Figure 20(b).
(6) Lines 4-5. The interpreter records the fact that connections to the `in` and `out` ports are valid, computes the port widths, and adds the *portname*.width parameters to the context.
(7) Lines 7-26. The interpreter executes the code recording the instantiations, assignments, and connections as before.

Once this code is finished, the interpreter checks to make sure that no connections were made to nonexistent ports, and no illegal parameter assignments were made. In this example, the code is correct, so evaluation continues. The next instance on the stack, in this case one of the `delay` module instances pushed onto the stack during the evaluation of the instance `delay3`, is popped off the stack and evaluated. Evaluation continues until the instance stack is empty.

This behavior can be specified more formally by describing the small-step semantics for the LSS language. The complete semantics is large and closely resembles small-step semantics for common imperative programming languages. Thus, in this section, only the small step semantics that relate to the implementation of use-based specialization will be described. This section uses the notation and terminology that is used by Harper [2002], unless otherwise specified.

The small step ($\rightarrow$) semantics of LSS will be expressed as the evolution of the state of the LSS program. The states of the program are represented by a 7-tuple, (M, I_s, L, A, B, e, S). M is the netlist of the design as it is known at the current point in program evaluation. I_s is the stack of instances that need to be processed. L is the evaluation context and maps symbols to values. A is the recorded list of potential parameter assignments and port connections obtained from the parent instance (the instance in the hierarchy above the one currently being processed). B is a context that records potential parameter assignments and port connections for children of the current instance. e is the current expression being evaluated and S is the current list of statements being evaluated.

The program starts in the initial state $(\cdot, \cdot, L_0, \cdot, \cdot, \cdot, S_0)$, where L_0 defines built-in functions and S_0 is the statement list at the top-level of the LSS specification.

The interesting rules for LSS evaluation are those that are used when a new instance is created and when the body of a module has finished evaluation. The rule for the instance creation statement is shown below. Note that the portion of the rules related to actually augmenting M are not shown but the extension is straightforward.

$$\frac{c \text{ current instance name} \quad n \notin \mathsf{dom}(L) \quad m \in \mathsf{dom}(L) \quad S' = \mathsf{body}(m) \quad i = (c.n, S')}{(M, I_s, L, A, B, \cdot, \texttt{instance}\, n : m;, S) \rightarrow (M', i \triangleright I_s, \{n \mapsto (c.n, S')\} \cup L, A, B, \cdot, S)}$$

This rule pushes the module body for instance i onto the stack of instances, I_s, that must be evaluated and continues evaluating the statements in the current statement list. Notice that this differs from standard evaluation which would have immediately begun processing S'.

When the current instance is finished (i.e. no statements are left in the current statement list), the following rule begins evaluating the next instances module body.

$$\frac{A = \emptyset \quad c \text{ current instance name} \quad i = (c.n, S') \quad A^* = \mathsf{extract}(c.n, B) \quad A' = \mathsf{strip}(A^*)}{(M, i \triangleright I_s, L, A, B, \cdot, \cdot) \rightarrow (M, I_s, L_0, A', B \setminus A^*, \cdot, S')}$$

The function $\mathsf{extract}(c.n, B)$ extracts from B all parameter assignments and connections for the instance named $c.n$. The function $\mathsf{strip}(A^*)$ strips the $c.$ prefix from all the symbol names, $c.n$, in the context A^*. The state for the next instance to be processed is established by extracting the recorded potential assignments for the about-to-be-processed instance and making this set of assignments the new A context. The $A = \emptyset$ hypothesis ensures that no assignments to undefined parameters can occur. The mechanism for this is explained below.

The remaining small step inference rules are very similar to other imperative programming languages. The most complex rules not shown are the ones for parameter and port declarations. The parameter rule removes from A any assignments to the parameter being defined and updates L and M appropriately. The port declaration rule is similar. Since the records are removed from A as they are validated, if $A \neq \emptyset$ after an instance's module body is evaluated, then an assignment or connection was made to an undeclared parameter or port.

8.2 Evaluation of Use-Based Specialization

Evaluating the utility of use-based specialization is difficult since modules can leverage this feature in many different ways. However, one common way in which this feature is used across most modules is the automatic inference of port widths. Therefore, to evaluate use-based specialization, we count the number of port width parameters that were inferred.

Table VI. Port width parameters inferred by use-based specialization

Model Name	Inferred Port Widths	Connections
A	816	919
B	823	929
C	147	304
D	611	3975
E	984	4528

Machine models are labeled as in Table V.

Table VI shows the results from the measurement across five different machine models. The table shows the number of port width parameters inferred and the total number of connections. In all of the models examined, use-based specialization was used to infer *all* the port width parameters. As the results illustrate, the number of parameter values inferred was quite substantial. Without use-based specialization, in one case, 984 port width parameters would have to have been kept consistent with 4528 connections. Our experience with use-based specialization suggests that avoiding this unnecessary manual specification dramatically reduces the overhead in using reusable components.

9. IMPLEMENTATION OF TYPE INFERENCE

Using polymorphic components can burden users by dramatically expanding the customization required to use a component. Not only must the module's parameters be specified, but all the polymorphism must also be resolved. While some of this burden is lifted by the use-based specialization technique described in the previous section, explicit type instantiations can still be burdensome. As was mentioned in Section 5.4, to reduce user burden, the LSS compiler employs a type inference algorithm that analyzes the structure of a system's connectivity to infer the types of many ports and connections, thus automatically resolving the polymorphic types.

In this section, the details of the LSS type system and a formalization of the type inference problem will be presented. As was mentioned earlier, the inference problem is NP-complete and differs from type inference problems found in the literature. This section will prove the problem's NP-completeness, explore the effect of this complexity on actual type inference running times, and present a heuristic inference algorithm to automatically resolve polymorphism.

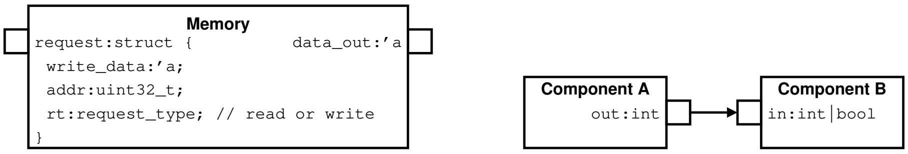
This section will conclude with experimental results that illustrate that type inference *dramatically* reduces any burden incurred by using polymorphism. Further, the results indicate that the inference algorithm can resolve polymorphism in complicated systems seen in practice in reasonable time. Taken together, these results demonstrate that using polymorphism in practice significantly enhances reusability while incurring little to no cost.

9.1 The Type System

The types supported in the LSS language are described by the following grammar:

Basic Types	τ	$::= \texttt{int} \mid \ldots \mid \tau[n] \mid \texttt{struct}\{i_1 : \tau_1; \ldots i_n : \tau_n;\}$
Type Variables	α, β, γ	$::= {}'identifier$
Type Schemes	τ^*	$::= \alpha \mid (\tau_1^* \mid \ldots \mid \tau_n^*) \mid \tau \mid \tau^*[n] \mid \texttt{struct}\{i_1 : \tau_1^*; \ldots i_n : \tau_n^*;\}$

Here, the basic types are the standard programming language types ($\tau[n]$ is the array



(a) Port interface for a simple memory.

(b) Connected components

Fig. 21. Components with types annotated.

type with elements of type τ) that would be communicated over ports at model run-time. The type schemes fall in two classes, recursive and non-recursive. The non-recursive type schemes are the collection of type variables and basic types. The recursive type schemes are disjunctive type schemes and aggregate types (such as structures and arrays) built from type schemes rather than basic types. When building modules and connecting instances users are permitted to annotate ports and connections with type schemes rather than directly using the basic types. These type schemes provide LSE with the polymorphism described in Section 5.4.

While recursive aggregate type schemes complicate type inference, the following example will illustrate why these type schemes are important. Consider the memory component shown in Figure 21(a). This component has a request port for accessing it and an output data port to return the results of read requests. As shown, the `request` port's data type needs to have some fields with fixed types for addressing and identifying the type of request and a polymorphic data field, whose type is specified with a type variable, to carry the data for any write requests. This same type variable is used on the `data_out` port to ensure that the data written to the memory has the same type as the data that is read out. If type schemes were not allowed in structure definitions (i.e. recursive type schemes were forbidden), a polymorphic memory component like this could not be built. Instead, polymorphism would have to be abandoned, reducing the memory's reusability or the request port would have to be broken up into many individual ports, making the memory cumbersome to use.

9.2 The Type Inference Problem

To automatically resolve the polymorphism for the type system described in Section 9.1, a type inference algorithm must identify a basic type for all type variables and ensure that a single type is selected from every disjunctive type scheme. When making such selections, the algorithm must respect each port's type scheme and ensure that connected ports share a common type. For example, consider the two components in Figure 21(b). Here, the `in` port on component B has the type scheme `int` | `bool`. Type inference must assign it the basic type `int` or the basic type `bool`. The `out` port on component A has type scheme `int` and thus can only be assigned type `int`. Furthermore, since `in` and `out` are connected, they must have the same type. This equality constraint forces the type inference algorithm to assign basic type `int` to port `in`. Since `int` is a basic type that corresponds to the type scheme `int` | `bool`, all conditions are satisfied and a unique valid type assignment has been found.

If no valid type assignment is possible (e.g. if the `in` port on component B had type scheme `float | bool`) then the system is said to be *over-constrained*. It is also possible to have a system with multiple valid type assignments (e.g. if the `out` port on component A also had the type scheme `int | bool`). Such systems are said to be *under-constrained*. Over-constrained and under-constrained systems are considered malformed systems and the type inference algorithm should report an error if such systems are encountered.

Formally, the type inference problem can be formulated as solving a set of constraints involving equality of type schemes. First, a type variable is created for every port in the system. Second, a core set of constraints is formed stating that these new type variables must be equal to the type scheme for the respective port. Third, the core set of constraints is extended by adding constraints formed by equating the type variables for any two ports that are connected. Finally, the type inference engine must resolve all the type schemes to basic types and assign values to all type variables according to the constraints. The legal constraints in the type system are given by the following grammar:

$$\text{Constraints } \phi \ ::= \ \top \ | \ \tau_1^* = \tau_2^* \ | \ \phi_1 \wedge \phi_2$$

$\top$ is the trivially true constraint, $\tau_1^* = \tau_2^*$ is a constraint asserting the equality between two type schemes, and $\phi_1 \wedge \phi_2$ is the conjunction of the sub-constraints ϕ_1 and ϕ_2.

THEOREM 1. *The type inference problem is in NP.*

PROOF. If we consider an instance of the type inference problem without disjunctive constraints, then the instance of the type inference problem is equivalent to an instance of a unification problem [Paterson and Wegman 1976]. The mapping between an instance of the type inference problem and the unification problem is simple. Type schemes are equivalent to terms, type constructors are equivalent to functions, type variables are equivalent to term variables, and concrete types are equivalent to constants. This unification problem can be solved in linear time and thus is in the complexity class P [Paterson and Wegman 1976].

For a given instance of the full type inference problem including disjunctive constraints, a non-deterministic algorithm need only select one type scheme from the disjunction in each disjunctive type scheme. Once this selection is made, the deterministic polynomial-time algorithm described above can be applied to verify the existence of a solution. Since the problem is polynomial-time verifiable, it is in NP. □

THEOREM 2. *The type inference problem is NP-hard.*

PROOF. To show the problem is NP-hard, we reduce any instance of the 3-in-1 monotone 3SAT problem, a known NP-complete problem [Garey and Johnson 1979], to the type inference problem. 3SAT is a SAT problem where one must decide if there exists an assignment of truth values to a set of boolean variables, B, such that each clause in a set of disjunctive clauses, C, with exactly 3 literals per clause, is satisfied. 3-in-1 monotone 3SAT is a 3SAT problem where each member of B only appears in non-negated form and only one literal in each clause may have a truth value of ‘1’.

The reduction proceeds as follows. Let the type `int` correspond to the truth value ‘1’, and let `bool` correspond to ’0’. For each variable $b_i \in B$ create a type variable $\alpha_{b,i}$. For each clause, $c \in C$, create a type variable α_c. Each clause has the form (b_i, b_j, b_k) where $b_i, b_j, b_k \in B$. For each clause add the constraint

$$\alpha_c = \texttt{struct } \{ \texttt{ x:}\alpha_{b,i}\texttt{; y:}\alpha_{b,j}\texttt{; z:}\alpha_{b,j}\texttt{;} \}$$

The only legal satisfying assignments for each clause is $S_0 = (1, 0, 0)$, $S_1 = (0, 1, 0)$, and $S_2 = (0, 0, 1)$. Let

$$S'_0 = \texttt{struct \{ x:int; y:bool; z:bool;\}}$$
$$S'_1 = \texttt{struct \{ x:bool; y:int; z:bool;\}}$$
$$S'_2 = \texttt{struct \{ x:bool; y:bool; z:int;\}}$$

To ensure that each clause is satisfied in a legal way, for each clause add the constraint $\alpha_c = S'_0 \mid S'_1 \mid S'_2$. This constraint ensures that each clause has a legal satisfying value (S_0, S_1, or S_2). The earlier clause ensures that boolean variables in the clause get the appropriate value based on which satisfying assignment is chosen. If the type inference problem for this set of constraints has a solution, the corresponding 3-in-1 monotone 3SAT problem is satisfiable. If not, the problem is unsatisfiable. Thus solving the type inference problem solves the corresponding 3-in-1 monotone 3SAT problem. □

COROLLARY 1. *The type inference problem is NP-complete.*

9.3 The Basic Inference Algorithm

The type system and inference problem presented here is similar to other type systems and inference problems presented in the literature. While some of these type inference problems are also at least NP-hard, the type systems and corresponding inference problems differ from the one in LSS and thus their type inference algorithms are not applicable. The type system and inference problem in languages such as Haskell [Peyton Jones et al. 2003] is very similar, but, to reduce the complexity of inference in practice, these languages exclude certain polymorphic interfaces which are rare for functions[9] [Odersky et al. 1995]. Unfortunately, this restriction is not desirable in a structural modeling environment since the corresponding port interface patterns are quite common. The Balboa [Doucet et al. 2002] structural modeling system shares a similar type system and inference problem to LSS but since Balboa only supports component overloading and not parametric polymorphism, its inference algorithm [Doucet et al. 2003] is also not applicable. On the other hand the Ptolemy structural modeling system supports parametric polymorphism, but not component overloading, so the type inference algorithm they propose [Xiong 2002] is also not directly applicable.

The substitution algorithm used for languages such as ML [Milner et al. 1997], however, can be extended for the presented type system by modifying the algorithm to handle the disjunctive type scheme. The algorithm determines if a satisfying solution for the constraints exists and, since the actual type assignments need to be known, also generates a typing context that maps type variables to basic types. This section presents an extension of this algorithm to handle the disjunctive type scheme and proves the correctness of the approach.

The basic ML-style substitution algorithm works by simplifying each term in the constraint, ϕ, and then eliminating it. As the constraints are simplified, the algorithm will create a new simpler constraint and then recursively process this new constraint. Simultaneously, the algorithm builds up a typing context, T, that maps type variables to type schemes. The recursion occurs based on the structure of the constraint. Therefore, there

[9]Note that this reduction in complexity is only for problems seen in practice, the type inference problem for Haskell is at least NP-hard [Henglein and Mairson 1991] despite these simplifications.

Table VII. Simple Substitution Algorithm

Constraint form	Operation
$(\tau^* = \tau^*) \wedge \phi$	$T_{\text{new}} \leftarrow T_{\text{old}}$ $\phi_{\text{new}} \leftarrow \phi$
$(\alpha = \texttt{struct \{ x1:}\alpha\texttt{; ...xn:}\tau^*_n\texttt{;\}}) \wedge \phi$	System is unsatisfiable
$(\alpha = \tau^*) \wedge \phi$, τ^* contains no disjunctive type schemes	$T_{\text{new}} \leftarrow [\alpha/\tau^*]T_{\text{old}} \cup (\alpha \mapsto \tau^*)$ $\phi_{\text{new}} \leftarrow [\alpha/\tau^*]\phi$
$(\tau^*_1\texttt{[n]} = \tau^*_2\texttt{[n]}) \wedge \phi$	$T_{\text{new}} \leftarrow T_{\text{old}}$ $\phi_{\text{new}} \leftarrow (\tau^*_1 = \tau^*_2) \wedge \phi$
$(\texttt{struct \{ x1:}\tau^*_{1,1}\texttt{; ...; xn:}\tau^*_{1,n}\texttt{;\}} =$ $\texttt{struct \{ x1:}\tau^*_{2,1}\texttt{; ...; xn:}\tau^*_{2,n}\texttt{;\}}) \wedge \phi$	$T_{\text{new}} \leftarrow T_{\text{old}}$ $\phi_{\text{new}} \leftarrow (\tau^*_{1,1} = \tau^*_{2,1}) \wedge \ldots \wedge$ $(\tau^*_{1,n} = \tau^*_{2,n}) \wedge \phi$
$\top$	Constraint satisfiable, Solution in T

must be a rule that handles each production in the grammar defining the constraint. The standard rules are shown in Table VII. If the constraint has the form shown in the left column of the table, then the action on the right hand column is taken (only the first matching rule, starting from the top of the table, is applied). Each of these rules operates by simplifying the constraint based on the definition of equality for the type schemes. Recall that τ denotes basic types, τ^* denotes type schemes, and $\alpha, \beta, \gamma, ...$ denote type variables. The notation $[y/x]Z$ means: substitute every occurrence of y in Z with x. Note that the state of the algorithm at any step can be summarized by the pair (T_i, ϕ_i) where T_i is the typing context and ϕ_i is the simplified constraint. The initial state is $(\emptyset, \phi)$.

The disjunctive constraint requires a new, non-standard rule and a modification to the form of constraints, since disjunctive type schemes are forbidden from the rule in the third row of the table. The rule on the third row of the table forbids disjunctive type schemes to ensure that the typing context, T, always maps type variables to concrete types, if all type variables had values. Allowing the disjunctive type scheme in the typing context would violate this property. We can avoid this difficulty by replacing every disjunctive type scheme with a type variable and adding a constraint asserting that the type variable is equal to the disjunctive type scheme. This simplified constraint involving disjunctive type schemes is handled by the following new rule. If the constraint is of the form $(\tau^* = \tau^*_1 \;|\; \ldots \;|\; \tau^*_n) \wedge \phi$, then n inference problems are created each starting in the state $(T_{\text{old}}, \phi_i \wedge \phi)$, where $\phi_i \equiv (\tau^* = \tau^*_i)$. Each problem is solved by recursively applying the algorithm. When the algorithm terminates for each subproblem, it reports that $\phi_i \wedge \phi$ was unsatisfiable or returns a typing context T_i. If, $\forall i, j,\ 1 \leq i, j \leq n$, ϕ_i and ϕ_j were satisfiable and $T_i \neq T_j$, then the system is under-constrained. The algorithm terminates if this occurs since only unique solutions are acceptable.

For the extended algorithm to be correct, two things must be true. First, the algorithm must ultimately terminate. Second, each step of the algorithm must be incrementally building a solution to the type inference problem. These two properties are stated formally in the next two theorems. Note that these theorems and proofs are extensions of existing theorems and proofs [Harper 2002].

THEOREM 3. *The algorithm's state sequence is finite.*

PROOF. To each state (T, ϕ), we will assign a pair (n, s), where n is the number of type variables in ϕ and s is the sum of the number of type schemes in ϕ. If (T_i, ϕ_i) is satisfiable, each rule shown in Table VII takes (T_i, ϕ_i) and converts it into a state (T_{i+1}, ϕ_{i+1}), where $n_{i+1} < n_i$ or both $n_{i+1} = n_i$ and $s_{i+1} < s_i$. Consider each row of the table. The first row, removes two type schemes from ϕ_i. The third row reduces the number of type variables by one. The fourth row removes two type schemes from ϕ_i, as does the fifth row. All rows either decrease s (without increasing n) or decrease n. Thus, the sequence of states form a strictly decreasing sequence of ordered pairs of integers(the pairs (n, s)). Since the sequence is lower-bounded ($n \geq 0$ and $s \geq 0$), the sequence will converge in a finite number of steps.

If some state in the sequence is unsatisfiable, then the algorithm terminates, and thus the theorem is trivially true.

Upon encountering a disjunctive constraint, the algorithm recursively applies itself to many subproblems. Each of the subproblems contains one fewer type scheme and thus, by the metric introduced earlier, is simpler than the current problem. Therefore, inductively, none of the subproblems will lead to an infinite sequence of states. Therefore, the algorithm will not produce an infinite sequence of states for any initial state. □

The next theorem will show that the final typing context T provides a solution to the initial constraint. However some useful notation will be introduced first. The application of a typing context S to a type scheme ϕ involves recursively substituting all the type variable to type scheme mappings in the type context S into the type schemes of ϕ. This operation will be denoted by $\overline{S}(\phi)$. Next, we will write $S \models \phi$ if the typing context S satisfies the constraint ϕ. Formally, this relation is inductively defined as follows:

$$\begin{array}{rl} S \models \top & \text{always} \\ S \models \tau_1^* = \tau_2^* & \text{iff } \overline{S}(\tau_1^*) \cap \overline{S}(\tau_2^*) \neq \emptyset \\ S \models \phi_1 \wedge \phi_2 & \text{iff } S \models \phi_1 \text{ and } S \models \phi_2 \end{array}$$

Note that in the second rule we are treating type schemes like sets. The type scheme $\tau_1^* \mid \ldots \mid \tau_n^*$ is considered to be the set $\{\tau_1^*, \ldots, \tau_n^*\}$. All other type schemes are considered to be singleton sets.

Finally, given a state (T, ϕ), we will call S a solution for that state if $S = T \cup U$ and $S \models \phi$ for some U whose domain is disjoint from the domain of T.

THEOREM 4. *If the algorithm transitions from state (T, ϕ) to (T', ϕ'), then S is a solution for (T, ϕ) if and only if S is a solution for (T', ϕ').*

PROOF. We will prove this statement by induction on the structure of the transitions. The transitions presented in Table VII are almost identical to those for programming languages such as ML and the truth of this theorem is well known for those transitions [Harper 2002].

For a transition based on the disjunctive rule, we have $\phi = (\tau^* = \tau_1^* \mid \ldots \mid \tau_n^*) \wedge \phi''$ and $\phi' = \top$. Assume that S is a solution to the state $(T, (\tau^* = \tau_1^* \mid \ldots \mid \tau_n^*) \wedge \phi'')$. Since $S \models (\tau^* = \tau_1^* \mid \ldots \mid \tau_n^*) \wedge \phi''$, then $S \models \tau^* = \tau_i^* \wedge \phi''$ for some i by the definition of the $\models$ relation. Therefore S is also a solution to the state $(T, \tau^* = \tau_i^* \wedge \phi'')$. The algorithm will transition in many steps from the state $(T, \tau^* = \tau_i^* \wedge \phi'')$ to $(T', \top)$. By the inductive hypothesis, S is also a solution to the state $(T', \top)$.

Now, assume that S is a solution to the state $(T', \top)$. We must show that S is also a solution to the state $(T, (\tau^* = \tau_1^* \mid \ldots \mid \tau_n^*) \wedge \phi'')$. Without loss of generality, we will assume that the algorithm will transition in many steps from the state $(T, \tau^* = \tau_1^* \wedge \phi'')$

to $(T', \top)$. By the inductive hypothesis, we have that S is a solution to the state $(T, \tau^* = \tau_1^* \wedge \phi'')$. This implies that $S = U \cup T$ for some disjoint U and that $S \models (\tau^* = \tau_1^* \wedge \phi'')$. By the definition of the $\models$ relation, this implies that $S \models (\tau^* = \tau_1^* \mid \ldots \mid \tau_n^*) \wedge \phi''$. By definition, we have that S is a solution to the state $(T, (\tau^* = \tau_1^* \mid \ldots \mid \tau_n^*) \wedge \phi'')$. $\square$

Theorem 3 proves that the algorithm always terminates. Further Theorem 4 proves that each step of the algorithm constructs an incremental solution to the inference problem. Therefore, if the algorithm terminates with the constraint equal to $\top$, then the typing context forms a solution to the initial state and thus to the initial type inference problem. When the algorithm terminates with a constraint other than $\top$, the terminating constraint is over-constrained and thus, by Theorem 4, so too is the initial state. Finally, recall that we also reject states where we detect *multiple* solutions. Theorem 4 tells us that these multiple solutions are all valid for the initial constraint also. Therefore, the initial constraint system is under-constrained.

9.4 Type Inference Algorithm Heuristics

As Section 9.5 will demonstrate, the basic algorithm just described has prohibitively large run times, which is not surprising given that the problem is NP-complete. The exponential running time is due to the subproblem exploration required by the disjunctive constraints. This section describes an additional rule to simplify and two heuristics to reduce the number of subproblems that need to be solved, drastically reducing run-time.

9.4.1 *Disjunctive Constraint Simplification.* The additional rule involves simplifying disjunctive constraints without actually creating and solving subproblems. Consider a constraint of the form $((\tau_{1,1}^* \mid \ldots \mid \tau_{1,n}^*) = (\tau_{2,1}^* \mid \ldots \mid \tau_{2,n}^*)) \wedge \phi$. This constraint can obviously be simplified by eliminating type schemes, $\tau_{1,j}^*$, that do not have a compatible type scheme in $\tau_{2,k}^*$, and vice versa. To do this, the algorithm needs to recognize compatible type schemes. While recognizing all incompatible type schemes is equivalent to the original type inference problem, certain incompatibilities are easy to verify. For example, any basic type τ has only two compatible type schemes, τ and α. Two `struct` type schemes where the element names do not match or where the number of elements differ are also incompatible. Similar rules can be constructed for arrays.

9.4.2 *Processing Constraints Out-of-order.* The first heuristic to reduce the number of subproblems to be solved delays the creation of subproblems due to disjunctive constraints as long as possible. This is allows other constraints to be simplified *before* creating subproblems to address a disjunctive constraint. These simplifications may make the disjunctive-constraint-simplification rule applicable, eliminating the subproblem creation entirely, or significantly reduce the size of the subproblems by solving common terms in the constraints only once.

The proposed heuristic to accomplish this causes the algorithm to evaluate the simplification rules in multiple phases. In the first phase, the constraint terms are treated as a list, and each constraint term is simplified in-order. If any constraint has a leading disjunctive type scheme, the constraint is reordered to delay evaluation of the disjunctive constraint term. This works because a constraint $x \wedge y \wedge \phi$ may be reordered to $y \wedge x \wedge \phi$ since the $\wedge$ operator is commutative. This iteration over constraint terms proceeds until no simplifications can be made without handling a disjunctive constraint term via subproblem creation. In the second phase, the first disjunctive constraint term remaining is handled via the subproblem creation used in the simple algorithm.

9.4.3 *Partitioning the Constraint.* The second heuristic leverages the independence of subproblems in type inference algorithm. Consider the following constraint:

$$('\mathtt{a} = \mathtt{int}|\mathtt{bool}) \ \wedge \ ('\mathtt{a} = \mathtt{int}|\mathtt{char}) \wedge \tag{1}$$
$$('\mathtt{b} = \mathtt{int}|\mathtt{bool}) \ \wedge \ ('\mathtt{b} = \mathtt{int}|\mathtt{char}) \wedge \tag{2}$$
$$('\mathtt{c} = \mathtt{int}|\mathtt{bool}) \ \wedge \ ('\mathtt{c} = \mathtt{int}|\mathtt{char}) \tag{3}$$

In this case there are six disjunctive type schemes with two possible types, thus the basic algorithm would need to solve $2^6 = 64$ subproblems. The heuristics presented thus far would reduce this to $2^3 = 8$ since half of the disjunctive type schemes can be handled by disjunctive constraint simplification after subproblems are created for the other half of the disjunctive type schemes. The number of subproblems can be reduced further still by recognizing that constraint terms given by (1), (2), and (3) each refer to disjoint sets of type variables and can thus be solved independently. Applying this technique would yield $2^2+2^2+2^2 = 12$ subproblems without disjunctive constraint simplification or $2+2+2 = 6$ subproblems with disjunctive constraint simplification.

In general, any constraint can be partitioned into multiple constraints by placing terms of the original constraint into different sets such that if two constraints are in different sets, they refer to non-overlapping sets of type variables. Each of these problems can be solved separately since a solution is affected by a mapping for a particular type variable only if the constraint references that type variable. After performing such a partitioning, if any one of the subproblems is unsatisfiable, then the whole system is unsatisfiable since no typing context which contains mappings for the type variables in the subproblem will satisfy the sub-constraint. If all the subproblems are satisfiable, the disjoint union of each subproblem's typing context is the typing context that solves the original constraint. In general the basic algorithm will need to explore $m \cdot n$ subproblems if it contains a disjunctive type scheme with m options and a disjunctive type scheme with n options. If these type schemes are independent, after partitioning, only $m+n$ subproblems need be explored. Thus, as the number of independent disjunctive constraints becomes larger, the savings of partitioning increase exponentially. Partitioning the constraint after reordering but before attempting to solve any subproblems increases the likelihood of finding independent disjunctive type schemes and is the best place to perform partitioning.

9.5 Evaluation

This section measures the effectiveness of the algorithm and heuristics presented earlier by evaluating them on several LSE models. Note that these models were not created for these experiments but are actual models used for research and instruction.

To evaluate the effectiveness of type inference in easing the use of polymorphism, the number of explicit type instantiations with type inference is compared to that without type inference. The results are shown in Table VIII. As can be seen from the table, far fewer instantiations are needed with type inference significantly reducing the burden on the user. In two of the models studied, only 8 out of approximately 100 type instantiations were user-specified. This shows how type inference can nearly eliminate any overhead associated with polymorphism while preserving all of its benefits.

To determine if the inference algorithm presented is usable in practice, the run times of the algorithm with and without the heuristics presented in the previous section were mea-

Table VIII. Evaluation of Type Inference and the Type Inference Algorithm

Model Name	Explicit Type Instantiations w/o Type Infer.	Explicit Type Instantiations w/ Type Infer.	Basic Algorithm Run Time	Run Time with Heuristics
A	115	8	> 12 h	6.54s
B	116	8	> 12 h	6.58s
C	38	30	14.9s	0.12s
D	162	71	> 12 h	1.78s
E	147	63	> 12 h	4.72s

Machine models are labeled as in Table V.

sured. Table VIII also presents these results. The model specification language compiler and type inference algorithm are implemented in Java. The run times are all measured using Sun's Java VM 1.4.1_02 on an unloaded 3.0GHz Pentium 4 machine with 2 GB of RAM running RedHat Linux 9 with RedHat kernel version 2.4.20-20.9smp. From the table it is clear that the type inference times without the heuristics are impractical often exceeding 12 hours. However, the algorithm with heuristics executes in just a few seconds, thus making it usable for models seen in practice.

10. MITIGATING THE REUSE PENALTY

While LSE provides for the use and creation of flexible reusable components, common wisdom dictates that reuse comes at a performance price. This section will show, however, that this is not the case. To do this, this section will first quantify the price, which we call the *reuse penalty*, by comparing the simulation speed of equivalent models built using flexible, reusable components in LSE and single-use, model-specific components in SystemC. The section will then present several optimizations that leverage the static analyzability of LSE models to eliminate the reuse penalty.

10.1 The Reuse Penalty

To measure the performance effects of using reusable components, we compared the simulation speed of two simulation models of the same simple 4-way out-of-order processor implementing the Alpha ISA having an instruction window of 16 instructions, a reorder buffer of 32 instructions, perfect caches, and perfect fetch. The models used were:

(1) A SystemC model using *custom* modules for each stage (fetch, decode/rename/commit, execute) plus a module to contain the other modules. These modules were designed specifically for this configuration and were designed for *speed*.
(2) An LSE model using modules from a library of *reusable* components. While speed was a consideration when building the LSE modules, the modules were *not* built for this particular configuration and were designed primarily for *flexibility*.

SystemC is used for comparison because it is a true concurrent-structural system and is gaining wide acceptance for architectural modeling[10].

[10] Note that in addition to the differences between the components used in the models, SystemC uses the discrete-event (DE) model of computation (MoC) while LSE uses the heterogeneous synchronous reactive (HSR) MoC. The reuse penalty measured by this experiment is *not* incurred by the selection of MoC. In fact the HSR MoC can significantly outperform the DE MoC on identical machine models [Penry and August 2003]. A more detailed discussion of MoCs is presented later in this section.

Table IX. Performance of models

Model	Mean cycles/s	Speedup	Build time(s)
Custom SystemC	53722	–	49.06
Reusable LSE	40649	0.76	33.9

Table X. Model characteristics

Model	Instances	Signals	Non-edge-sampled Signals
Custom SystemC	4	71	32
Reusable LSE	11	489	423

Table IX shows the performance (measured in cycles/sec) and build time of each of the simulation models. The speedup of the LSE model versus the SystemC model is also shown. All models were compiled using gcc 2.96 for x86 with flags `-g -O2`. Version 2.0.1 of SystemC was used; it was compiled with the same compiler using the default installation flags. The models were run on nine benchmarks chosen from the SPEC CPU2000 and MediaBench suites. All these benchmarks were compiled for the Alpha ISA for OSF Unix with gcc 2.95.2 19991024 (release) using flags `-static -O2`. Runs took place on a dual-processor 2.4 GHz Xeon system with 2 GB of physical memory running Redhat 7.3 (Linux 2.4.18-5smp). Further experiment details can be found in the references [Penry and August 2003]. Note that none of the optimizations described in the next sections were applied to the LSE model in this experiment.

From the table, we see that the reusable LSE model suffered a 24% slowdown (.76 speedup) when compared to the custom SystemC model. Examination of the two models illustrate several significant differences between them. Table X shows the number of module instances, signals, and non-edge sampled signals present in each model. Notice that the reusable LSE model has more instances and significantly more signals. This increase in instances and signals is caused by several factors:

—Reusable modules tend to be finer grained than custom components. In particular, reusable components often separate combinational logic and state elements into different modules. The finer-grained components are more easily composed in various configurations to achieve a wide range of behavior.

—Increased numbers of modules require more signals, since the modules need to communicate with one another.

—Custom modules tend to use custom control interfaces that require fewer signals. For example, if information is always removed from a multiple-output buffer in FIFO order, only a count of how many items to remove needs to be used for control rather than individual remove signals.

—Custom modules avoid non-edge-sampled signals by doing as much work as possible at the clock edge. Generally, the only non-edge-sampled signals are timing-control signals. Since reusable modules often separate combinational logic and state elements into different modules, both data and timing-control signals are often non-edge-sampled.

The increase in the number of signals, particularly the non-edge-sampled signals, causes additional overhead during simulation. Much of the overhead arises in code that handles communication and concurrency suggesting that optimizations to the implementation of concurrency and communication can mitigate the reuse penalty. Since the semantics of this communication and concurrency are specified by the model of computation (MoC), the next section will describe what a model of computation is and provide an overview of the HSR MoC to clarify discussions of the optimizations described later.

10.2 Models of Computation

Components written for concurrent-structural modeling systems are written as if they are executing concurrently with other components. However, when these systems are compiled, they must be translated into a sequentially executing program. The modeling system's *model of computation* defines the semantics of the concurrent execution and specifies rules for how the components in a model can be scheduled for execution so the illusion of concurrency is maintained.

The choice of model of computation can limit the reusability of components. For example, the MoC used by EXPRESSION [Mishra et al. 2001] and Asim [Emer et al. 2002] forces manual partitioning of components in some situations [Vachharajani et al. 2002]. On the other hand, the discrete-event model of computation used by systems such SystemC, VHDL, and Verilog is extremely flexible and forces no such partitioning. Unfortunately, the flexibility of the discrete-event MoC incurs significant overhead when used for simulation [Penry and August 2003].

LSE uses the Heterogeneous Synchronous Reactive (HSR) model of computation [Edwards 1997]. This model is an extension of the Synchronous Reactive model (used in languages such as Esterel [Berry and Gonthier 1992]) for situations where the precise input-to-output data flow of blocks of code are unknown (such as when the block is being treated as a black box). As will be seen, the HSR MoC provides sufficient flexibility to avoid the mapping problem, but can be analyzed and optimized to produce an efficient simulator.

10.3 The Heterogeneous Synchronous Reactive MoC

With the HSR model of computation, components communicate by manipulating signals. The values sent on these signals are ordered by some partial order. There must be a least value, which will be denoted as *unresolved*, which is "less than" all other values. Code blocks that manipulate signals must compute monotonic functions on the signal values; if input values increase (relative to the partial order), output values must increase or remain the same. Changes to signal values are seen instantly by all receiving blocks.

Recall from Section 6 that each connection in LSE is actually composed of three subconnections: a data connection, an enable connection, and an acknowledge connection. For the purposes of the HSR MoC, each connection in LSE can carry one of three signal values: *unresolved*, *high*, or *low*. The partial order on these values is simple: *unresolved* is less than both *high* and *low*, and all other signal values are not comparable. Note that in addition to this signal value, the data connection between two components also carries a data value when it is *high*. The monotonicity required by the HSR MoC is on the signal values and *not* on the data values. Note, however, that once a data signal has taken a *high*-value, neither the signal value nor the corresponding data value may change.

Computation within a time step takes place by setting all signal values to *unresolved* and invoking blocks until the signal values converge. The schedule of invocations can be either static (determined at system build time) or dynamic (as a signal changes values, its receivers are scheduled, as in Discrete-Event). Note that each time a block is invoked, it need not change every output signal value. The same block may be reinvoked later in the same time step so that it may process changed input and produce additional output. Since blocks can be reinvoked, the HSR model of computation does not require repartitioning of blocks and, thus, avoids the pitfalls of other more restrictive MoCs.

10.4 Optimizations for HSR Systems

To implement the HSR model of computation, LSE provides a scheduler that determines the order of execution for code blocks[11]. Edwards [1997] proposed a static scheduling algorithm for HSR systems. This algorithm analyzes the connectivity of the system and statically determines an invocation order for blocks. Since the invocation order is determined statically and is independent of data values transmitted, the static schedule could potentially reduce run-time overhead since no data structures or computation are required to determine which block to execute next. Unfortunately, the scheduling problem is NP-hard, and consequently the scheduling algorithm has prohibitively high running times for large models.

In this section, several improvements to Edwards' scheduling algorithm are discussed. These optimizations rely on the ability to statically analyze the structure of the model. LSE's scheduling algorithm uses a novel approach that falls back to a dynamic schedule for portions of the static schedule that are too difficult to resolve. The LSE simulator generator will then interleave these dynamic sections in the appropriate places in the static schedule. This optimization and other related techniques are discussed in the remainder of this section.

10.4.1 *Dynamic sub-schedule embedding.* The Heterogeneous Synchronous Reactive (HSR) model, as proposed by Edwards, allows optimal static schedules to be generated. These schedules are optimal with respect to the amount of information present about the input to output dependencies of blocks in the system. In the absence of information, all outputs of a block are assumed to depend upon all inputs of the block. When there are far fewer real dependencies than the assumed dependencies, it can happen that the "optimal" schedule actually executes blocks far too many times in an effort to guarantee convergence. We will take advantage of this property in a moment.

Edwards' basic static scheduling algorithm is:

(1) Build a directed graph where each signal is a node and an edge (u, v) implies that signal u's value is needed to compute signal v. This graph will *not* be acyclic in the presence of limited dependency information because the timing-control signals form cycles.
(2) Break the graph into strongly-connected components (SCCs). Topologically order the SCCs.
(3) Partition each SCC of more than one signal into two sub-graphs called the head and the tail. Pick an arbitrary schedule for the head. Schedule the tail recursively. Add to the schedule a loop containing the tail's schedule followed by the head's schedule; this loop executes as many times as there are signals in the head. Follow the loop with the tail's schedule again.

[11] In LSE a code block and a module are not the same; a block is the smallest unit of scheduling granularity, but a module can contain more than one block. In particular, control functions are separate blocks. Note that the partitioning of modules into code blocks is done for efficiency. A module implemented as a single code block does *not* need to be partitioned by the user based on use as required of blocks in EXPRESSION or Asim.

The key to the algorithm is the partitioning in the third step. This raises the question of how the SCC should be partitioned. Trying all partitions to find the optimal one requires time exponential in the number of signals. Edwards presents a branch-and-bound algorithm and suggests some further heuristics to prune the search space, but shows empirically that the algorithm remains exponential.

An exponential-time algorithm to find the optimal schedule is not particularly useful for large models. However, the places in the graph where the algorithm breaks down (large SCCs) are precisely the locations where information about dependencies is lacking. Real synchronous hardware does *not* usually have cyclic dependencies (there are some distributed arbitration schemes that do, but these are relatively rare). Thus, the cycles within large strongly-connected components are not generally real dependence cycles and an "optimal" static schedule will not be truly optimal.

In such a situation it is better to "give up" on the static schedule *for just the signals in large SCCs* and to embed a dynamic schedule into the static schedule at the location of the SCC. Doing so prevents the scheduler from taking exponential run-time; it may also improve simulation time. For SCCs containing 16 or more signals, LSE gives up immediately. For smaller SCCs, LSE tries all partitions to find the optimal one. Note that *average* execution time is being considered; Edwards' original work needed to bound the *worst* case time and therefore dynamic scheduling would have been less appropriate there.

10.4.2 *Dependency information enhancement.* If no knowledge of input to output dependencies is available for any block, the generated schedule is generally one large dynamic section. This occurs because the timing-control signals form cycles in the dependency graph. We use three techniques to increase the amount of dependency information available:

—LSE partially parses the control functions to find input/output dependencies and constant control signals. Dependencies upon constant control signals are removed. Computation of such constants is removed completely from the schedule and handled directly by the framework at the start of the time step.
—We optionally annotate ports with an "independent" flag in the module definition indicating that the input signals of these ports do not affect any outputs.
—We optionally annotate modules with their input/output dependencies. This can be tedious, and would be unacceptable for modules not in a highly reusable library, but the effort is amortized over the number of times the module gets reused and is always useful. Modules without the annotation are assumed to have all outputs depending upon all inputs.

Note that the relationship between dependency information and scheduling is counterintuitive. When information is completely missing, it is easy to schedule: the schedule is dynamic. When information is completely present, it is also easy to schedule; the dependency graph would likely be acyclic. It is when some, but not all, information is present that scheduling becomes difficult. The power of LSE's modified HSR scheduling algorithm lies in the way in which it can gracefully adapt to the available information.

10.4.3 *Code specialization.* As noted in Section 4, module instance code is generated from templates. This provides the ability to specialize the code generated by creating different implementations of the port API for each port of each module. The API implementations are specialized for:

Table XI. Performance of reusable models

Cycles/sec (speedup vs. slowest)				
Schedule Type	Control Functions Analyzed?	Annotations		
		None	Port	All
Dynamic	No	40649 (1.00)	40693 (1.00)	40904 (1.01)
	Yes	47794 (1.18)	47860 (1.18)	47821 (1.18)
Static	No	* 40657 (1.00)	* 41377 (1.02)	* 41306 (1.02)
	Yes	* 47850 (1.18)	* 47098 (1.16)	57046 (1.40)

* Framework embedded dynamic schedule in a static schedule.

—Constant control signals

—Static vs. dynamic scheduling of receiving code blocks

—Diversity of receiving blocks across port connections: if all connections to a port are to the same block, code to update variables and do scheduling (if any) is simpler.

10.5 Evaluation of optimizations

In this section, the effectiveness of LSE's optimizations is evaluated by measuring the running time of the LSE model built with reusable components with different optimizations enabled. The experimental setup and reporting is the same as in Section 10.1; the only difference is that different combinations of optimizations are tried.

Note that no meaningful comparison can be made with Edwards' original scheduling algorithm because of its exponential run-time. An attempt to schedule the LSE model using Edwards' original algorithm did not terminate after 18 hours of run-time. Such a large run-time would be impractical for design exploration. As a consequence, all static scheduling in these experiments is done using LSE's dynamic sub-schedule embedding which had a maximum compile time of 36.21 seconds (only 7% longer than without optimization).

Table XI shows the performance of the *reusable* LSE model when information enhancement and scheduling are varied. Notice that the role of information is very important; a 40% speedup can be obtained by analyzing the control functions and using module input/output dependencies for static scheduling. Even with dynamic scheduling there is still an 18% speedup to be obtained from control function analysis. The effects of analysis and module annotations are synergistic; only when both are present can a fully static schedule and the maximum speedup be obtained.

Note that when all the information is used for static scheduling, the LSE model built from *reusable* components is 6% faster than the SystemC model built from *custom* components. These results show that the combination of all optimizations with the HSR model of computation has allowed the reuse penalty to be completely eliminated when compared with a conventional Discrete-Event model. Notice further, that this elimination is only possible by analyzing the static structure of the model at *compile-time*. This emphasizes that the capability for static analysis is a key design element.

11. CONCLUSION

This article presented the design and implementation of the Liberty Simulation Environment (LSE), 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.

APPENDIX

A. QUESTIONS FOR SUBJECTS

This appendix lists the questions that each subject was asked regarding the machine models in the experiment described in Section 2.2. Note that subjects were told that though they recognized the code in the sequential simulator, the code had been modified from the stock version of the code. More details regarding questions can be found in the references [Vachharajani and August 2004].

The questions are as follows (note each question was asked twice, once for the sequential C simulator and once for the hardware-like model):

(1) **Issue Windows and Reorder Buffers**
  - (a) In the machine modeled in the configuration question3/machine1.xxx, Examine file question3/machine1.lss lines xxx through yyy. This is the highest level code that describes the issue window/reservation stations.
    Does the machine described have a unified issue window (excluding the load store queue), or a distributed set of reservation stations (still ignoring the load store queue).
  - (b) How many entries does each reservation station/issue window have?
  - (c) In order to recover from speculation, the machine in this configuration also keeps track of the issue order of instructions. Is it possible to have an instruction in the reorder buffer but not in the issue window? That is to say, can there be more decoded instructions in flight (not committed) than the cumulative size of the reservations stations/issue window. (Hint: The code for the commit stage of the machine is in pipestages.lss, lines xxx to yyy.)

(2) **Branch Resolution Policy** In the machine modeled in the file question6/machine1.xxx, what is the branch resolution policy of the machine? In particular:
  - (a) Identify the line or lines of code which identify (detect) mis-speculated branches. (Hint: Look at lines xxx-yyy of machine1.xxx)
  - (b) In what stage of the pipe are misspeculated branches identified?
  - (c) Is this the same stage which initiates branch recovery (Recovery is initiated when the machine stops fetching from the wrong path)? (Hint: Look at lines xxx-yyy of machine1.xxx)

(d) If not, then what stage initiates branch recovery?
(e) Do instructions reach the pipeline stage where recovery is initiated in-order?
(f) If the answer to question 5 was no, is it possible to begin recovery on a later branch instruction (later in program order) before initiating recovery on an earlier branch?
(g) If the answer to question 6 was no, what hardware structure ensures that the branches are processed in order? How does that structure communicate with the pipeline stage?

REFERENCES

AUSTIN, T. M. 1997. A User's and Hacker's Guide to the SimpleScalar Architectural Toolset (for toolset release 2.0).

BERRY, G. AND GONTHIER, G. 1992. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming 19,* 2, 87–152.

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. June.

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 Transactions on Modeling and Computer Simulation 8,* 4 (October).

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

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*.

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*.

EDWARDS, S. A. 1997. The specification and execution of heterogeneous synchronous reactive systems. Ph.D. thesis, University of California, Berkeley.

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 Computer 0018-9162*, 68–76.

GAREY, M. R. AND JOHNSON, D. S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W H Freeman & Co, New York, NY.

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)*.

HARPER, R. 2002. *Programming Languages: Theory and Practice*. Draft.

HENGLEIN, F. AND MAIRSON, H. G. 1991. The complexity of type inference for higher-order lambda calculi. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, 119–130.

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 2001)*.

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*. 220–242.

MILNER, R., TOFTE, M., HARPER, R., AND MACQUEEN, D. 1997. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, MA.

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 (ISSS)*. 256–261.

ODERSKY, M., WADLER, P., AND WEHR, M. 1995. A second look at overloading. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*. 135–146.

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*. http://www.systemc.org.

PATERSON, M. S. AND WEGMAN, M. N. 1976. Linear unification. In *Proceedings of the eighth annual ACM symposium on Theory of computing*. ACM Press, 181–186.

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*.

PEYTON JONES, S. ET AL. 2003. Haskell 98 language and libraries: the revised report. *Journal of Functional Programming 13,* 1.

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 CONVOT, B. 1995. OO-VHDL Object-Oriented Extensions to VHDL. *IEEE Computer*.

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

VACHHARAJANI, M. AND AUGUST, D. I. 2004. A study of the clarity of functionally and structurally composed high-level simulation models. Tech. Rep. Liberty-04-01, Liberty Research Group, Princeton University. January.

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*. 271–282.

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

Received January 2004;