Microarchitecture Modeling for Design-space Exploration

Manish Vachharajani

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Electrical Engineering

November 2004

Abstract

To identify the best processor designs, designers explore a vast design space. To assess the quality of candidate designs, designers construct and use simulators. Unfortunately, simulator construction is a bottleneck in this design-space exploration because existing simulator construction methodologies lead to long simulator development times. This bottleneck limits exploration to a small set of designs, potentially diminishing quality of the final design.

This dissertation describes a method to rapidly construct high-quality simulators. In particular, it examines structural modeling as a means to reduce construction time because it eliminates redundant effort required to manage design complexity in many modeling approaches, including that of programming a simulator in a sequential language. The dissertation also describes how to overcome common limitations in structural modeling that increase construction time by precluding amortization of component specification effort across models via component-based reuse. First, some time-consuming to specify design portions do not benefit from reuse, the logic for managing stall signals (i.e., timing-control) chief among them. Second, components flexible enough to enjoy reuse are often not reused in practice because of the number of details that must be understood in order to instantiate them. This dissertation addresses these issues by:

- Presenting new insight into timing-control that allows it to be partitioned amongst reusable components.
- Describing the application of programming language techniques (such as polymorphism and aspect-oriented programming) to allow creation of easy to reuse flexible components.
- Allowing models to be statically analyzable in the presence of the above features.

The techniques presented in this dissertation are embodied in the Liberty Simulation Environment (LSE). LSE users have seen over an order of magnitude reduction in model

development time. Their models were of high-quality; they were accurate, had adequate simulation speeds, and were compatible with static techniques for visualization and optimization.

Short model construction times allow more ideas to be explored in less time. This leads to shorter product time-to-market and more thorough design-space exploration. This also allows researchers to evaluate new design ideas faster, to efficiently evaluate ideas in the context of many designs, and, perhaps, develop a more fundamental understanding of microarchitecture due to these broader evaluations.

Acknowledgements

First, I thank my advisors, Sharad Malik and David August, for their support during my tenure at Princeton. I learned much from my discussions with them, their advice, their guidance, and their example. The quality of this dissertation, my research, and my Princeton experience would be greatly diminished if not for them.

For many fruitful discussions, their time, their effort, and their support of LSE, I thank Jason Blome, Ram Rangan, Matthew Bridges, Spyridon Triantafyllis and the rest of the Liberty Research Group. I especially thank David Penry and Neil Vachharajani for their innumerable contributions to the development of LSE and keen insight into many problems encountered during its design and implementation. I also thank all the external users and early adopters of LSE for their support and feedback. In particular, I thank Paul Willmann, Michael Brogioli, and Vijay Pai for adopting LSE early and providing us with detailed feedback. I am also in debt to all those who attended the Liberty tutorials and contributed to the user study of LSE presented in this dissertation.

I also thank my many friends both in and out of Princeton, they have made my time in graduate school a most enjoyable and memorable experience. I thank the lovely Dana Bhatnagar for her love, constant encouragement, limitless support, and patience with me while I completed this dissertation. I can no longer imagine life without her. Most of all I thank my parents for their love, sacrifice, and devotion. They have provided the foundation for all that I have accomplished and all that I ever will; I am forever in their debt. I dedicate this thesis to them.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In digital hardware system design, the more meaningful design alternatives that are properly considered, the more likely designers are to identify a successful design. Since prototyping a candidate design is prohibitively expensive, designers rely instead on models to evaluate design alternatives. Analytical models have many desirable properties. For example, they usually provide results quickly and sometimes allow simultaneous consideration of a vast design-space [1, 50]. Unfortunately, current analytical modeling methods are only sufficient to provide accurate guidance for special cases, such as particular applications running on a specific class of machines [35] or exploring a limited number of parameters on a specific design-space [1]. As a result, designers generally construct high-level (e.g., microarchitecture level) software simulation models for feedback.

1.1 Simulators Built with Sequential Languages

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 simulator construction methodology.

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 microarchitectural 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 [13]. The belief is that modifying a validated simulator will be easier and result in an accurate derivative. However, as this dissertation will explain, simulator modification suffers from the same problems as simulator construction; it is time-consuming and error-prone. Worse, the quality of the original simulator 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 challenge of manually mapping a design to a sequential simulator, and the resulting difficulties, is called the mapping problem. Chapter 2 covers building simulators using sequential languages and consequences of the resulting mapping problem in detail.

1.2 Architecture Description Languages

An alternative to writing a simulator in a general purpose sequential language is to use a purpose built processor modeling language. Many such systems have been proposed, and most authors refer to these systems as architecture description languages or ADLs [7, 17, 18, 21, 22, 23, 37, 40, 45, 47]. These ADLs provide a number of advantages over simply writing using a sequential language. First, they more clearly convey certain architectural, and sometimes microarchitectural ideas. Second, many of these systems reduce the complexity of the specification by inferring aspects of the design. Third, these systems often

provide descriptions that can be analyzed to construct a compiler for the design or build an optimized simulator that is faster than hand-coded ones [45].

Unfortunately, these systems are not without problems. First, many of the systems still suffer from the mapping problem or variants of it. In fact, the very design elements introduced to make instruction set architecture (i.e., architectural ideas) ideas clear cause these systems to have the mapping problem and obscure microarchitectural ideas. Second, the systems often make modeling easier by making assumptions about the underlying hardware, greatly limiting the class of designs to which they can be applied.

Chapter 3 explores the strengths and weaknesses of these ADLs in detail.

1.3 Concurrent-structural Modeling Systems

The concurrent-structural approach [28, 38] is an approach that eliminates the mapping problem by simply eliminating the manual mapping [57]. This approach involves a language which allows designers to directly express the composition of the hardware in terms of components and 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 microarchitectural ideas, is easy for designers to understand, and exposes model/design mismatches.

Unlike in the manually-coded simulator approach, reuse can be quite effective in the concurrent-structural approach. In concurrent-structural models, reuse at the component level is an attractive way to reduce model construction time [10, 52] *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.

Unfortunately, existing concurrent-structural modeling systems tools force a trade-off

between the ability to easily *build* reusable components and the ease of *using* such components [55]. In current systems, this puts a high overhead on reuse in the form of components that are difficult to reuse because they require manual specification of too many parameters. The implementation of features that allow for easy construction of reusable components in these systems precludes general, automatic mechanisms for inferring the values of many of these parameters. As a result, users are left with either too few reusable components or reusable components that are too hard to use. This reduces reuse in practice, and thus negates its benefits.

In addition to this, some portions of a hardware design's specification do not benefit from reuse in concurrent-structural systems. The portion of the control logic that determines when and what computation to stall in a processor (i.e., timing control) is a prime example. Timing control does not benefit from reuse because it is non-local in nature. The timing-control logic must be aware of all components, their interactions, and sometimes their semantics. Changes to one part of a processors datapath often requires updating the stall conditions in the global controller. The global nature of this timing control makes it difficult to partition into one or more reusable model components. Consequently, in existing concurrent-structural systems, users are forced to manually specify this complex control for each design, foregoing reuse.

Chapter 4 examines concurrent-structural systems, the trade-off between construction and use of reusable components, and the modularity of timing control in more detail.

1.4 Contributions

This dissertation shows that it is possible to dramatically reduce the time needed to create a high-quality model specification (e.g., a model that is accurate, can be visualized, and can be compiled to an efficient simulator) by exploiting component-based reuse in a concurrent-structural environment. This is accomplished by first analyzing existing modeling systems

(including sequential programming languages) to identify the root cause of their shortcomings. The dissertation shows that mapping a microarchitecture to a sequential program is slow and error-prone and why reuse cannot allow the cost of simulator development to be amortized when using this approach (Chapter 2). This dissertation then proceeds by analyzing the strengths and weaknesses of various non-structural and pseudo-structural ADLs to identify their strengths and weaknesses (Chapter 3). Finally, this dissertation examines true concurrent-structural systems to see why they still do not create an environment which encourages reuse (Chapter 4).

Based on this analysis, this dissertation then presents the design and implementation of techniques that enable a concurrent-structural modeling system that avoids the pitfalls but preserves many strengths of existing systems (Chapter 5, 6, and 7). To evaluate these ideas, they were implemented in the context of the Liberty Simulation Environment (LSE), a new high-level modeling framework. Since LSE incorporates the techniques developed in this dissertation, it allows users to exploit component-based reuse in practice and has a model compiler that can compile their specifications into an efficient simulator. LSE avoids the mapping problem because it 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 existing and novel programming language techniques presented in this dissertation. These techniques include parametric polymorphism, component overloading, structure-based type inference, and aspect-oriented programming. LSE also incorporates a control abstraction developed in this work to simplify the specification of timing control. This control abstraction functions similarly to the control mechanisms used in asynchronous digital hardware design methodologies to manage synchronization of computation in the absence of a clock. Since these techniques have been carefully designed to permit static analysis of models, LSE can exploit static analyses such as optimizations for simulators generated from models.

Though young, LSE has already been used to rapidly produce accurate models that have

been compiled to efficient simulators. Highlights include a simulator for Intel's Itanium 2 processor, developed at Princeton University by a single student in only 11 weeks, that predicts performance to within 3% for a variety of benchmarks as well as a validated model of a 3COM TIGON-2 network interface controller constructed at Rice University by two students in only 6 weeks.

This dissertation concludes by summarizing contributions, highlighting remaining shortcomings, and discussing promising avenues for future research (Chapter 8).

Chapter 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 that 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. This type of encapsulation and communication is called *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

with arguments and receiving return values. This type of encapsulation and composition is called *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 chapter, 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 chapter 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 sequential simulator by modifying an existing one is difficult, illustrating that the cost of building and validating sequential 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 se-

quential 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 from *implicit* communication channels between the caller and callee function. The arguments to the callee are implicit outputs of the caller and the callee return value and implicit input. The communication is implicit because it is not part

of the caller's declared communication interface, namely its arguments and return values. Further, the recipient of data sent on these implicit outputs and the sender of data received on these implicit inputs is determined by the function named as the callee in the code of the caller. Thus, the user is not free to reconnect these implicit inputs and outputs freely as they can in with structural composition. Furthermore, unlike structurally composed components, 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 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 and obfuscated by 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 pro-

gramming 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 that 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 2.1(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 2.1(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 2.1(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.

```
1  foreach simulation cycle
2     do commit stage
3     do writeback stage
4     do ex stage
5     do issue stage
6     do fetch stage
```

(a) original main simulator loop

```
1  instr[1..n] = first n instructions in queue
2  for each instr[i] that was fetched
3     if instr[i] is a branch
4        fetch source registers
5        if branch unit is available
6           compute the target PC
7           issue instruction to the branch unit
8           dequeue instr[i]
9     if instr[i] is an ALU op
10       fetch source registers
11       If ALU is available
12          issue instruction to the ALU
13          dequeue instr[i]
```

(b) original issue stage

```
1  foreach simulation cycle
2     do commit stage
3     do writeback stage
4     do ex stage
5     do issue stage
6     do fetch stage
7     foreach instr[i]
8        if instr[i].issued
9           dequeue instr[i]
```

(c) modified main simulator loop

```
1  instr[1..n] = first n instructions in queue
2  for each instr[i] that was fetched
3     if instr[i] is a branch
4        fetch source registers
5        if branch unit is available
6           compute the target PC
7           issue instruction to the branch unit
8           instr[i].issued=TRUE
9     if instr[i] is an ALU op
10       fetch source registers
11       If ALU is available
12          issue instruction to the ALU
13          instr[i].issued=TRUE
```

Issue Logic

(d) modified issue stage

Figure 2.1: Sequential simulator code for a 5-stage superscalar processor.

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 2.1(c) and 2.1(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 2.1. 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 2.1(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 [13].

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

2.2 Simulator Construction and the Mapping Problem: Model Clarity

The previous discussion describes why accurate sequential simulators are difficult to build. Correctness is especially hard to ensure because the simulator is very hard to understand. The semi-formal user experiment presented in this section supports the claim that sequen-

tial simulators are difficult understand and thus difficult to validate.

2.2.1 Experimental Setup

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 in Chapter 5, 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*.

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 [4]. The code in `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 [26]. A refined version of this model was released along with the Liberty Simulation Environment in the package `tomasulodlx` [32].

Subjects received different versions of the machine models to ensure that the effects observed were not due to a particularly difficult to understand hardware policy in a particular model. There were 4 sequential simulator models:

- Stock `sim-outorder.c` which has a unified issue window and flags mispredicted branches in the decode stage, begins recovery in the write-back stage. Note that flagging mispredicted branches in the decode stage involves emulating instructions at decode which provides results that would not be available in hardware. This information is used to prevent recovery for mispredicted branches that will be squashed.

- A modified `sim-outorder.c` which had a unified issue window, but identified and resolved mispredicted branches in the commit stage, in-order.
- A modified `sim-outorder.c` which had a unified issue window, but identified and resolved mispredicted branches in the write-back stage, supporting recovery of mispredicted branches that would eventually be squashed.
- A modified `sim-outorder.c` which had a distributed set of reservation stations and the stock `sim-outorder.c` branch semantics.

There were 2 structural models built in LSE:

- A machine with a unified set of reservation stations with mispredicted branches identified and resolved at commit.
- A machine with a distributed set of reservation stations with the above branch resolution policy.

The base LSE model was built for instructional and demonstration purposes and thus the machine was modeled at a fairly low-level and closely resembles the hardware. This made the model an excellent reference point for clarity. There are fewer LSE models than sequential simulators because the low-level of the base LSE model (the machine actually computed results in the pipeline instead of pre-computing the results in the decode stage) made the construction of models that used data that would be unavailable in hardware difficult.

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 2.1. The questions in the questionnaire are shown in Section A.1.

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

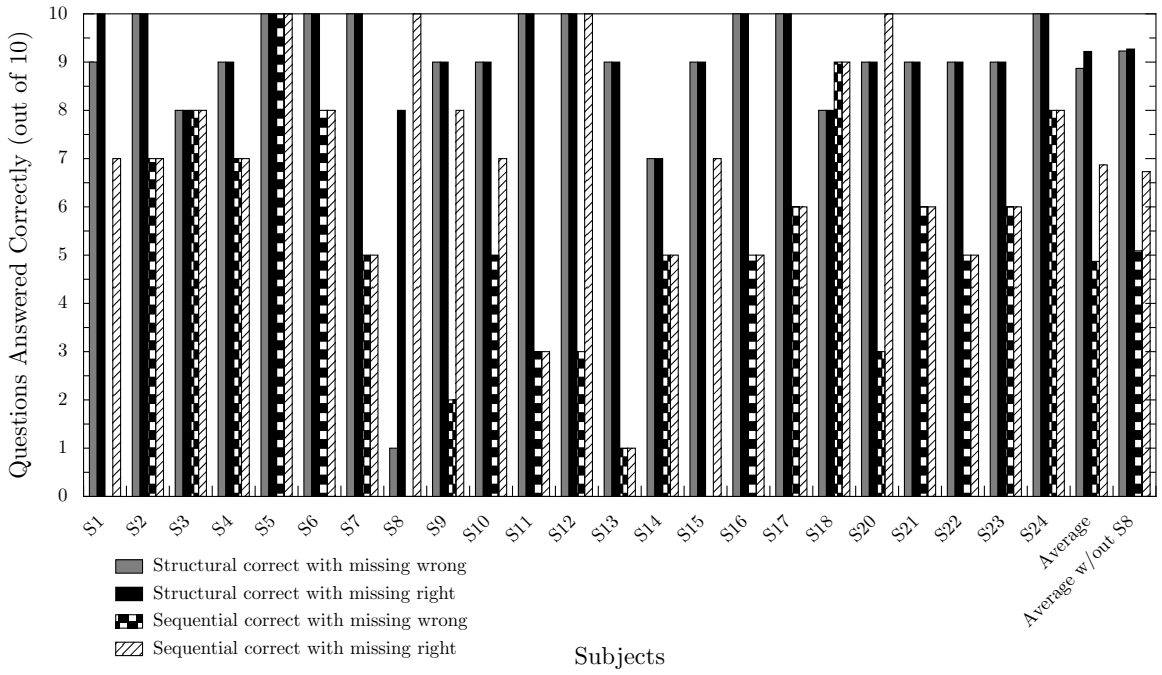
Table 2.1: Subject responses to the background questionnaire.

Note that finding qualified subjects unaffiliated with the Liberty Research Group for this experiment was challenging given the requirements. Subjects had to be very familiar with computer architecture and also had to be familiar with LSE. Though growing in popularity, LSE is still a young system, making the second constraint the most difficult to satisfy. Only 24 of all LSE users could be recruited as subjects for this experiment. As will be seen, however, even with 24 subjects the results are quite dramatic.

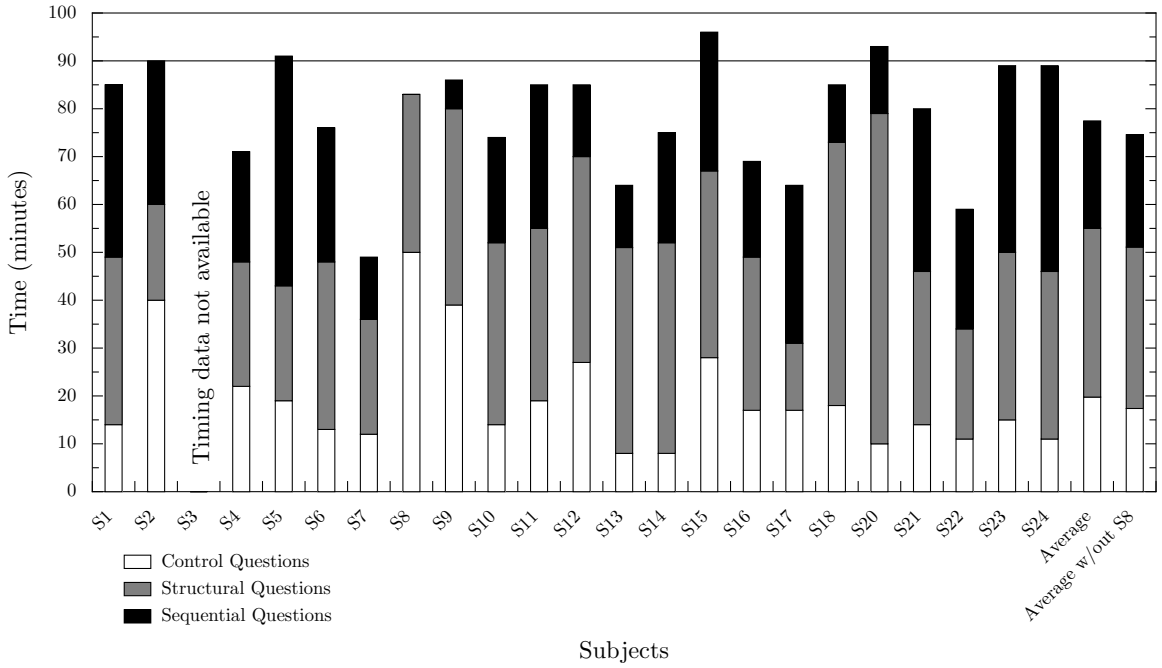
Subjects were given 90 minutes to answer 2 control questions, 2 multipart questions for the structural model, and 2 identical multipart questions for the sequential simulator. These questions, with placeholders for the line numbers, are shown in Section A.2. The control questions were used to determine if the subjects had basic familiarity with LSE since all but one subject had less than one week of experience with the tool. The answers to the control questions indicated that all subjects, except subject S19, understood LSE enough for the purposes of this experiment. Accordingly, the data presented here excludes subject S19. It is clear from the questionnaire results in Table 2.1 that all remaining subjects were familiar with the C programming language, the language with which the sequential model was constructed.

2.2.2 Experimental Results

Figure 2.2(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 all but a few cases subjects were able to 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 8.87 questions correctly with



(a) Number of correctly answered questions, by subject.



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

Figure 2.2: Results of the simulator clarity experiment.

the structural model versus 4.87 for the sequential model. Even if unanswered questions are assumed correct for *only* the sequential simulator, almost all subjects answered more questions correctly with the structural model. The only subjects who do better with the sequential simulator, under these circumstances, are subjects S8 and S20. Subject S20 simply failed to respond to 7 out of 10 of the questions for the sequential simulator. S8 spent an unusually long time on the control questions and thus did not have time to answer any questions regarding the sequential simulator. In fact, S8 only completed the first two parts of the first non-control question for the structural model (questions 1(a) and 1(b)) as numbered in Section A.2 and no questions for the sequential simulator. Clearly subject S8 is an outlier.

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

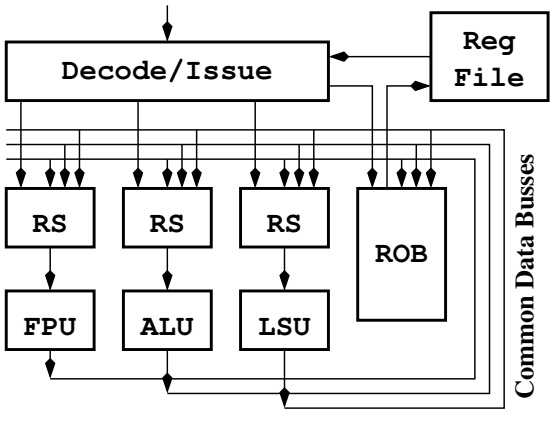
Figure 2.2(b) summarizes the time taken to answer each class of questions: control questions, questions for the structural model, and questions for the sequential model. With S8 excluded, the average time taken per question for the structural model is greater by about 10 minutes when compared to the time for questions about the sequential simulator. Some of this gap is due to the fact that some subjects did not complete all the sequential simulator questions, and the time taken for unanswered questions does not count toward the

average. Despite this, the increased number of correct responses for the structural model is not likely due to the extra time spent on those questions; many of the total times are well below the 90 minute time limit. One conclusion to be drawn from the timing data is that the sequential simulator is extremely misleading. Subjects voluntarily spend less time answering the sequential simulator questions despite plenty of extra time. Presumably, this is due to their confidence in their rapid analysis. Yet, the subjects frequently mischaracterize properties of the machine modeled. Timing data for subject S3 was unavailable because the subject failed to record the data during the examination.

Note that the experiment is skewed in favor of the sequential simulator. First, subjects were asked the LSE question immediately before being asked the same question for the sequential simulator. Thus, subjects could use the hardware-like model to understand what to look for in the sequential simulator. Second, many of the subjects were familiar with the stock `sim-outorder.c` simulator. Third, all subjects had many years of experience using the C language (the language used for `sim-outorder.c`) and less than a week of experience with LSE (the tool used to build the structural model). Fourth, no subject had ever seen a full LSE processor model before the experiment. Finally, the testing environment did not permit subjects to use the LSE visualization tool [6] to graphically view block diagrams of the structural specification. Experience indicates that this tool significantly simplifies model understanding; subject responses to the questions indicated that much of their time answering questions about the structural model was spent drawing block diagrams. Despite all this, the results clearly indicate that sequential simulators are significantly more difficult to understand.

2.3 Reuse and the Mapping Problem

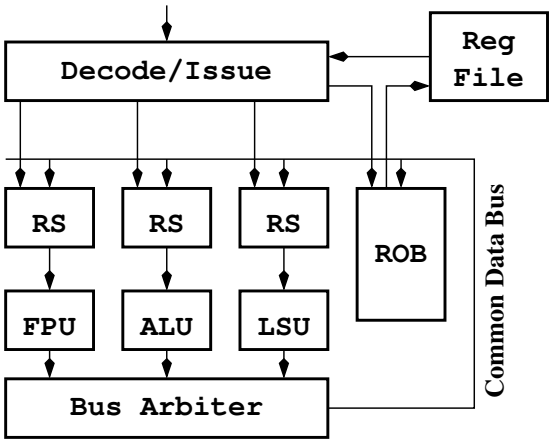
A tempting approach to allow rapid construction of simulation models in the face of the previously described difficulties is to amortize the cost of model construction via whole-



(a) Original machine

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

(b) Pseudo−code for original machine



(c) Limited write−back 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

Figure 2.3: The structure and pseudo-code for two Tomasulo-style machines.

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 2.1, which models a machine that uses Tomasulo's dynamic scheduling algorithm [54]. Figure 2.3 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, 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. It seems that one need only modify the loop termination condition to cause the loop to exit if the writeback bandwidth has been exceeded. Figure 2.3(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 2.3(d)) processes the requests. Thus, as shown in Figure 2.3(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.

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 informal experiment that gauges how rapidly an existing sequential simulator could be *correctly* modified was conducted. Specifically, a subject was asked to

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 RUU into a separate issue window and reorder buffer

Table 2.2: Descriptions of the modeled microarchitectural variants.

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 2.2. 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[1] 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

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

Configuration	LSE Model			Sequential Model		
	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

Table 2.3: Time spent and code changed for modifications from the baseline configuration.

modification took. Any time required to resolve discrepancies between the output of two models was charged to the model that was deemed to be incorrect.

The results of the experiment are summarized in Table 2.3. 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 Summary and Prognosis

When designing a microarchitecture, designers manage complexity by dividing the design into smaller components each performing simpler computations. Each component has a well-defined communication interface in the form of a declared set of ports. Full system behavior is realized by specifying the communication between components by connecting up the component ports. In this method communication is orthogonal to computation; each

component can perform an arbitrary computation and the designer is free to connect any component port to any other compatible component port.

Building a simulator for such systems in a sequential language such as C or C++ (i.e., building a sequential simulator) has much appeal. The arguments and return values of functions seem analogous to ports and function bodies can perform arbitrary computation. Furthermore, these languages can model any architecture and most designers are proficient in some sequential language resulting in a short learning curve for using the tools.

Unfortunately, the examples and data presented demonstrate that manually mapping a concurrent, structural microarchitecture to a *correct* sequential program can be quite difficult because functions *do not* encapsulate computation like a structural component. The data shows that sequential simulators are difficult for users to understand and time-consuming for users to modify. Others have noted that there is a problem with the accuracy of simulators, but disagree on the source of the problems [5, 9, 13, 20]. The results in this chapter indicate that the mapping problem is the fundamental cause of many 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. Others have proposed a number of alternative means of modeling. These systems are explored in Chapters 3 and 4.

Chapter 3

Non-structural and Pseudo-structural Architecture Description Languages

As an alternative to writing a simulator in a sequential language such as C or C++, researchers have proposed a number of special purpose architecture description languages (ADLs). These languages improve the efficiency with which processor models can be created and some allow analysis and concise specification of the processor's instruction set.

ADLs fall into two broad categories. Behavioral ADLs and microarchitectural ADLs (μADLs). Behavioral ADLs are concerned with specifying instruction set behavior, while μADLs are concerned with modeling microarchitectural properties, such as instruction timing, power consumption, and resource utilization. In this section, ADLs will be classified into three categories: behavioral ADLs, instruction-centric μADLs, and pseudo-structural μADLs[1]. This chapter examines these three ADL categories to identify strengths and weaknesses. In particular, the sections below will evaluate how well the ADLs preserve the general applicability of sequential languages and to what extent the ADLs address the mapping problem.

[1]For convenience, when it does not cause confusion, the term ADL will be used to refer to μADLs as well.

3.1 Behavioral ADLs

Behavioral ADLs are those that are designed to describe instruction set *behavior*, from instruction semantics to calling conventions, but *not* instruction timing or other microarchitectural detail. Since these ADLs are not designed to model microarchitecture level details, they are not of direct relevance for this dissertation. However, for completeness, a brief description of these ADLs is included below.

In their simplest form, behavioral ADLs allow designers to specify a list of instructions and the semantics for each instruction. Usually, the only structural information represented in these ADLs are the programmer visible structures such as the main memory and register files. A common feature of these languages is the ability to factor out the common portions of instruction specifications, such as effective address calculations or instruction decoding. This idea, first implemented in nML [18], works by allowing users to hierarchically specify instructions. At the top level of the description are the instructions themselves. These instruction descriptions can reference partial instructions (PIs) declared elsewhere. Each PI can in turn include other PIs [18, 51]. Each PI invocation behaves like a BASIC subroutine invocation or an argument free C preprocessor macro.

Behavioral ADLs often incorporate a number of other specification features. For example, ISDL [22], allows users to explicitly specify constraints on which instructions may be issued or fetched together. This allows a compact representation of the coding constraints for VLIW like machines. In addition to instruction semantics, CSDL [47] adds support for calling conventions, and a formal specification of assembly language syntax and complex instruction encoding rules.

3.2 Instruction-centric μADLs

Instruction-centric μADLs are those which have the user model the timing and behavior of a processor microarchitecture by specifying processor behavior on an instruction by instruc-

tion basis. This provides a very natural way to specify many architectures. Architectures in which all execution behavior is synchronized with a serial instruction stream are extremely easy to specify in these languages. Such architectures include RISC architectures as well as VLIW machines (though not all languages support every such architecture).

Aggressive processor designs often have performance critical operations that occur independently of or are only loosely coupled to the flow of instructions through the processor (e.g., data prefetching, instruction prefetching, and cache line replacements in non-blocking caches). Unfortunately, these operations either cannot be modeled in these languages, or constructing a model that includes these operations requires designers to face the mapping problem (discussed in Chapter 2). Since each instruction-centric μADL can vary quite a bit in the manner in which the model is specified, the remainder of this section examines a few representative instruction-centric μADLs and explains how each suffers from at least one of these two shortcomings.

3.2.1 Resource-based Instruction-centric μADLs

The simplest instruction-centric μADLs are, in fact, not designed for detailed simulation at all. They are, instead, designed to expose to a compiler how instructions occupy pipeline resources. For example, in Maril [7], an instruction definition for an 'add' instruction may appear as follows:

```
1  %instr Add r, r, r (int)
2          {$3 = $1 + $2}
3          {IF; ID; EX; MEM; WB} (1,1,0)
```

Line 1 declares the Add instruction to take three integer register operands (denoted by the `r`'s and the `(int)`). Line 2 specifies the semantics of the add instruction. Line 3 specifies the resource utilization, cost, and issue latency of the add instruction. The first part of the line is of the most interest. Each identifier specifies a resource occupied by the instruction. The semicolon separates a clock cycle. Thus, in this description, the `Add`

instruction occupies the `IF` resource in the cycle it is fetched, the `ID` resource in the next cycle, the `EX` resource in the cycle after, and so on [7, 51].

HMDES [21], used in the IMPACT [27] compiler toolset, also models a processor pipeline by describing resource utilization on an instruction-by-instruction basis. However, unlike Maril, HMDES can model VLIW as well as RISC processors and allows for far more complex resource specifications. In particular, HMDES allows for multiple resource usage specifications for a single instruction as well as a hierarchical specification of resource utilization to allow compact specification of complex pipelines.

While it is difficult to build a cycle accurate simulator using just these resource descriptions, it is possible to build a simulator by using the resource utilization for the main pipeline and modeling the control logic, caches, and other hardware using another specification system. The IMPACT compiler's Lsim simulator combines HMDES in a C simulator to achieve such a result.

HMDES allows the specification of fairly general pipelines and eliminates the mapping problem for pipeline specifications it can describe. Unfortunately, it does have limitations. For example, HMDES cannot describe certain types of issue constraints [46] and does nothing to address the specification of processor structures outside the in-order portion of the pipeline (e.g., the caches or scheduling logic in an out-of-order processor). Unfortunately, these structures have a significant effect on performance and must be modeled. Note that the strategy used in Lsim to model these structures while using HMDES for the main pipeline suffers from the mapping problem since it relies solely on a sequential language for modeling processor structures.

3.2.2 The LISA Architecture Description Language

The LISA architecture description language [40] is a instruction-centric μADL developed at the University of Aachen in Germany. It is designed to allow user specification of architecture and microarchitecture from which a compiler and simulator can be generated.

LISA has the user break down instruction behavior into a single operation per instruction per pipeline stage. At the conclusion of execution, each operation specifies which operation to activate in next clock cycle. The framework uses this information along with a number of assumptions to automatically generate the pipeline control. Processor behavior not directly tied to instruction behavior, such as prefetch mechanisms and line replacements in non-blocking caches, can be modeled using a "main" operation that executes in each clock cycle and is not associated with any instruction.

In the Compiler Design Handbook [51], Qin et al. give a sample specification for LISA, a reduced version of which is shown below. Users first specify a set of pipelines for the different instruction types. For example, the pipeline declaration for a 5 stage RISC machine would be as follows:

```
PIPELINE pipe={FE; ID; EX; MEM; WB}
```

The decode stage for arithmetic instructions looks as follows:

```
1  OPERATION arithmetic IN pipe.ID {
2    DECLARE {
3      GROUP opcode=(ADD || ADDU || SUB || SUBU)
4      GROUP rs1, rs2, rd = {fix_register};
5    }
6    CODING {opcode rs1 rs2 rd}
7    SYNTAX {opcode rd "," rs1 "," rs2}
8    BEHAVIOR {
9      reg_a = rs1;
10     reg_b = rs2;
11     cond = 0;
12   }
13   ACTIVATION {opcode, EX}
14 }
```

Lines 1-5 declare the operation description and define some useful groupings, `opcode`, `rs1`, `rs2`, and `rd`. Line 6 describes the instruction coding and line 7 the assembler syntax

(for compiler construction). Lines 8-12 describe the decode behavior for arithmetic instructions, which is to read the registers and stash their values in `reg_a` and `reg_b`. Notice the activation record at the end of the listing. This line tells LISA that upon execution of this operation, the operation for the `EX` stage of the opcode corresponding to the arithmetic instruction being decoded should be activated next. If `opcode` was `ADD` then an operation declared as follows would be activated.

```
1  OPERATION ADD IN pipe.EX
2  {
3    CODING {0b001000}
4    SYNTAX {"ADD"}
5    BEHAVIOR {alu = reg_a + reg_b; }
6  }
```

Notice that this description style is well suited to simple, single issue, in-order machines. The level of abstraction is excellent and specification rapid, as reported in the literature.

Unfortunately, the language is only well suited for these architectures. If a design has many operations that execute concurrently with the pipeline, but are only loosely coupled to instruction execution, the user must encode most of such behavior in the main operation. Since the code within an operation specification is simply sequential code in a C-like language, this approach suffers from all the issues related to the mapping problem discussed in the previous chapter, precluding rapid construction of accurate models for these designs. Furthermore, it is not clear if LISA can be used to model all processors, especially those with multiple issue or out-of-order execution, limiting the applicability of the language.

3.2.3 The MADL Architecture Description Language

The MADL μADL is an approach developed at Princeton University [45]. Once again, MADL is designed to allow the rapid specification of models (for both architecture and microarchitecture), construction of efficient simulators, and automated construction of compilers. The MADL language has users build a model by specifying, for each operation,

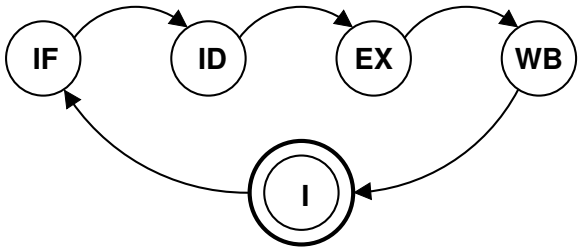


Figure 3.1: An Operation State Machine (OSM) for an add instruction in a five stage pipeline.

what is called an operation state machine (OSM). Each state in the OSM can specify computation that is to occur in each state. In simple designs, each instruction corresponds to a single operation. For example, in a five stage pipeline, the add instruction could be a single operation with the OSM shown in Figure 3.1. Each state corresponds to a stage in the pipeline, with the start state being the idle state.

Each of the OSMs (one for each dynamic operation in flight) executes in parallel. Before transitioning to a new state, an OSM will attempt to procure machine resources. These resources are modeled as tokens and allocation of tokens is controlled by a set of *token managers*. For each token request, the token managers must decide if this request will be granted. When the instruction is finished with the resource, the OSM will release the token back to the appropriate token manager. Each resource typically has a token manager, but token managers may coordinate across cycles via global state.

As an example, consider a model of a five stage pipeline RISC machine. In this model, the add instruction shown in Figure 3.1 would request the token manager managing the decode stage resources for a token corresponding to the decoder. The token manager would be implemented to only have a single token thus allowing only one instruction in the decode stage at any given time.

The MADL approach is quite elegant because it partitions instruction execution concurrency and microarchitectural concurrency. Instruction concurrency is modeled using OSMs and microarchitecture concurrency is encapsulated in the token managers. This separation allows MADL to model a larger variety of machines when compared to LISA. MADL can

model superscalar and VLIW machines, though out-of-order execution may pose problems.

Unfortunately, the token managers are typically specified using sequential code with no hardware-block like encapsulation of microarchitecture structure and computation. Thus the specification of microarchitectural concurrency, such as prefetchers, caches, and other units, looks much the same as the examples in the previous chapter. Thus, these specifications suffer from the mapping problem and its consequences. Furthermore, as described, each token manager must respond to a token request without examining the other requests that will occur in a given cycle, precluding any form of request based arbitration within a cycle. This latter point is easily remedied by changing the token manager interface. The former point is more difficult to address without substantial additions to MADL. Many of these additions would likely be along the lines of those proposed in this dissertation. In fact, integrating a MADL like language for specification of control designed to enforce instruction dataflow semantics with a system like the one proposed in this dissertation is a promising area for future work.

3.3 Pseudo-structural ADLs

Instruction-centric μADLs suffer shortcomings because they attempt to partition microarchitecture behavior around instructions rather than hardware structure. Pseudo-structural μADLs, on the other hand, have designers specify microarchitectures by partitioning computation in a fashion similar to that of hardware. Blocks of computation are encapsulated in components and are then interconnected via the components' ports, just like hardware blocks. On the surface, these systems have all the properties needed to address the mapping problem and seem to provide a platform for reuse. Unfortunately, each of these systems imposes restrictions on the type of concurrency and communication permitted. These restrictions cause the mapping problem to resurface, precluding wide ranging reuse.

3.3.1 The UPFast ADL

Of the Pseudo-structural μADLs, UPFast [37] is perhaps the most limited. Users specify a machine by declaring a set of pipelines along which instructions will execute, much like in LISA. For each of these pipeline stages, the user defines a set of time annotated procedures (TAPs) which describe the behavior of the stage. The UPFAST framework ensures that control information stored in piperegisters and a variety of other needed information is transmitted to each TAP. Each TAP may directly specify instruction behavior or rely on portions of an ISA description specification within the UPFAST ADL. Typically, operations common to all instructions, such as writing back to the register file, are handled in the TAPs directly. Instruction specific behavior is handled by the ISA behavior specification.

To model portions of the microarchitecture not directly in the main pipeline, or accessed by multiple pipeline stages, the system provides the ability to instantiate *artifacts*. Each artifact is instantiated with a keyword in the system (thus limiting the type of artifacts supported). Artifacts include register files, caches, and main memory. The TAPs, which define pipeline behavior, communicate with the artifacts via port-like interconnections. The artifact will respond to the data on these ports in some number of cycles determined by the artifact itself and its instantiation parameters.

The TAPs themselves must be manually ordered by the user to ensure that they are executed in an order such that the artifacts are able to respond to requests in the same cycle if needed. Unfortunately, TAPs may execute only once per cycle, and thus this requirement causes many of the issues in the mapping problem to resurface. In particular, TAP partitioning becomes a serious problem. It is not clear whether artifacts can be partitioned to deal with any inter-artifact communication issues. These mapping issues, along with the limited flexibility arising from the lack of user-defined artifacts makes UPFAST's utility limited to a narrow range of architectures.

3.3.2 The EXPRESSION ADL

The EXPRESSION ADL [23] is a pseudo-structural μADL because users specify a microarchitecture by interconnecting user-defined and predefined components to describe the processor pipeline. The system operates much like a concurrent-structural modeling environment, as described in the first section of the next chapter.

The toolset is flexible; it provides tools that assume standard pipeline semantics to infer control. Users may also forgo the tool and explicitly specify the control interface and logic. If specifications have a certain standard structure and rely on the framework to infer the control, the system can also generate a compiler for the specified machine.[2]

Unfortunately, EXPRESSION requires that each component only be invoked once per clock cycle. This means that if a component has an input which is combinationally dependent on one of its outputs the component must be partitioned (as with sequential simulators) in order to manage this connection pattern. Experience with reuse in true concurrent-structural systems indicates that this paradigm can be common, especially with a control abstraction mechanism described in Chapter 5. For example, a common paradigm in this control abstraction works as follows:

1. Component A sends data to Component B for processing.

2. Component B examines this data and determines that processing cannot proceed this cycle and signals this fact to Component A by sending data to one of A's inputs.

3. Component A then updates its internal state based on this message to retry the operation next cycle.

In this example, the input of Component A that receives B's cannot-proceed signal is combinationally dependent on the data sent to B by A. Thus, this back and forth exchange is not possible without partitioning a component in EXPRESSION.

[2]This mode of operation is similar to MIMOLA [31, 64] and UDL/I [2], though EXPRESSION does not infer the instructions themselves as done in UDL/I and MIMOLA)

From this discussion, we can see that EXPRESSION suffers from portions of the mapping problem. In particular, component's must be partitioned due to intra-component communication patterns specified in the design. Note that relying on EXPRESSION's control inference does not address this issue since doing so restricts EXPRESSION to modeling in-order processors with standardized pipelines.

3.3.3 The Asim Framework

Asim [17] was originally developed at Digital Equipment Corporation and is now in use at Intel corporation for modeling a variety of processors. The tool was designed specifically to prevent users from "cheating" by using computation results before they would be available in real hardware.

Asim models are C++ programs that utilize a variety of classes to interface with instruction traces and instruction set emulators. Components in Asim are objects, derived from the appropriate base class. Each component in Asim can declare a set of ports and a user specifies a machine model by interconnecting these ports.

To prevent the "cheating" described above, each port in Asim is annotated with a delay indicating the number of cycles that a component must wait before using the value on a port. For example, the output port of a cache component may be labeled with a delay of 5 cycles. This means that users must wait 5 cycles after the cache produces a value before reading it. The framework enforces this rule by not making data available until the annotated number of cycles have elapsed.

Asim is a powerful framework and supports some degrees of reuse. In particular, virtual functions and inheritance can be used to extend base components to employ new functionality. Furthermore, different implementations that conform to a particular interface can be interchanged. This allows designers to try a number of implementations to identify the best design for a component [17].

Unfortunately, Asim requires that all ports have at least a unit delay for automatic

scheduling of concurrency to work. Any computation that must occur within a clock cycle must occur through sequential function invocation, once again leading to the mapping problem and limited reusability of components. It is possible to connect a port with zero delay, however, in this case the user is forced to schedule and partition the code manually for each specification, which once again leads to the mapping problem. This lack of scheduling with zero delay ports and the lack of a control abstraction limits reuse and ease of model construction.

3.4 Summary and Prognosis

This section evaluated three ADL classes (behavioral ADLs, instruction-centric μADLs, and pseudo-structural μADLs) in terms of generality, flexibility, and how well they avoid the mapping problem. In summary, each ADL provides significant advantages over the sequential simulator approach though none completely address all the issues. Behavioral ADLs allow rapid specification of instruction set semantics. Unfortunately, these ADLs do not allow microarchitecture specification and thus are not useful for microarchitecture level modeling or design-space exploration.

Instruction-centric μADLs allow rapid modeling of some architectures. Some of these ADLs even support extraction of information for automated compiler construction. Unfortunately, these ADLs also have limitations. Many trade-off flexibility for their rapid modeling times. Of course, for a general design-space exploration this is undesirable since a reasonable exploration could easily lead outside the domain supported by these tools. Furthermore, many of these tools suffer from the mapping problem when modeling hardware not tightly coupled to the main pipeline.

Pseudo-structural μADLs are the most promising of all the tools for general purpose modeling in design-space exploration. The best of them support encapsulation of computation that is *mostly* orthogonal to the communication interface. Unfortunately commu-

nication is not completely orthogonal to communication since these systems restrict the concurrency and connection patterns in such a way that certain interconnection patterns require partitioning of the computation within a component. This precludes wide scale reuse due to the mapping problem.

Note that while none of the presented ADLs provides a general platform that completely avoids the mapping problem, the closer the ADL constructs match concurrently executing hardware components with port-like communication, the fewer problems we encounter. This clearly points the way to a solution to the mapping problem, true concurrent-structural modeling systems. These systems are the focus of the Chapter 4.

Chapter 4

True Concurrent-structural Approaches

From previous chapters, it is clear that a simulation system that avoids the mapping problem must be fully concurrent and support structural composition. These systems are known as concurrent-structural modeling systems. This chapter begins with an overview of concurrent-structural systems, discusses reuse in existing concurrent-structural systems, and finishes with a discussion on reusing and modeling timing-control in these systems.

4.1 Overview of Concurrent-structural Modeling Systems

Concurrent-structural modeling systems are those systems in which computation is encapsulated in concurrently executing system components that communicate values to one another by sending and receiving data on a predefined communication network. Typically, each component defines a set of input and output ports and continuously computes its outputs given the current values of its inputs and internal state. Components whose input ports are connected to a particular output port receive the value sent on it and accordingly update their internal state and outputs. Synchronous digital hardware, where state update is synchronized to a global clock, is an example of a concurrent-structural system. Concurrent-structural modeling systems are the most natural way to model such hardware and are often used for this purpose. Figure 4.1(a) shows a structural model that adds two numbers to-

gether. Figure 4.1(b) shows a possible textual description of the structural model.

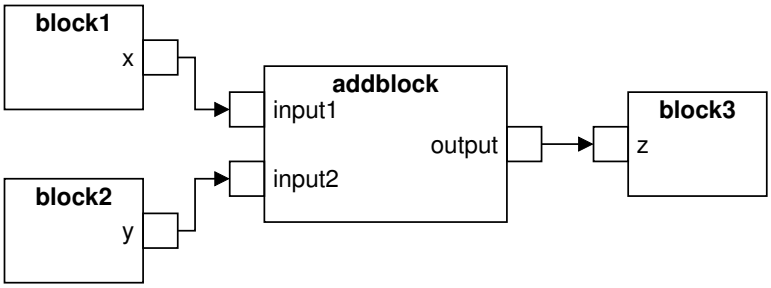
In *high-level* concurrent-structural models, the focus of this work, the primitive components' port I/O relation (e.g., the function an arithmetic logic unit (ALU) will compute) is specified using normal function-invocation based code. Contrast this to low-level concurrent-structural systems (such as Verilog [53]) in which components' port I/O relation is specify via interconnections of low-level primitives such as transistors, gates, or a static sequence of basic logical operations.

In high-level models, despite the use of function-invocation code *within* a particular component, communication *between* components occurs structurally. This distinction clearly separates functionality and communication, a hallmark of concurrent-structural systems. Continuing the earlier example, Figure 4.1(c) shows a possible description of the `adder` component's behavior.

4.2 Reuse in Concurrent Structural Models

Concurrent-structural modeling fully addresses the mapping problem since it provides the same encapsulation as provided in hardware. As a result, the complexity of the simulator can be managed using the same divide and conquer strategy used in other parts of the design process (thus avoiding the time-consuming and error prone re-divide and re-conquer required in sequential simulators).

Concurrent structural modeling itself does not *prevent* the construction and use of reusable components. In practice, however, simply not preventing component-based reuse is insufficient. Hardware designs feature much common functionality. However, each design contains customizations of these common behaviors. In many hardware designs, this customization is practical since much of the hardware must be custom built anyway due to changing process technology. In order to allow component reuse across models of these designs, high-level concurrent-structural modeling systems need to possess certain capa-



(a) Structural Addition

```
...
/* Instantiate adder */
instance addblock:adder;

/* Connect ports */
block1.x -> addblock.input1;
block2.y -> addblock.input2;
addblock.output -> block3.z;
```

(b) Structural Code

```
component adder {
  ...
  void compute() {
    int x, y, z;
    x = receive(input1);
    y = receive(input2);
    z = add(x,y);
    send(output, z);
  }
}
```

(c) Behavioral Code

Figure 4.1: A Generic Structural Specification.

bilities which *enable* component-based reuse [52]. These capabilities include:

Parameterizable Components - the ability to customize component properties with parameters. Example: a cache component whose replacement policy can be selected from an enumerated set of predefined policies.

Structural Customization - the ability to customize hierarchical structure with parameters. This allows existing components to be reused hierarchically to create a *flexible* component. Example: customizing the mix of functional units and bypass connection specification in a structurally specified reusable CPU core.

Algorithmic Customization - the ability to inherit and augment the behavior of an existing component with an algorithm. Example: customizing arbitration logic inside a bus arbiter component.

Polymorphism - the ability to support reuse across types.

Parametric Polymorphism - the ability to create and use component models in a datatype independent fashion. Examples: queues, memories, and crossbar switches that support all types.

Component Overloading - the ability for a component's implementation to be automatically selected from a family of implementations that support different datatypes. 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: An ALU with an implementation family that operates on integer and floating point numbers.

Instrumentation - the ability to insert probes into a model without modifying the internals of any component. This allows models to be reused to satisfy different data collection needs. Examples: instrumentation for performance measurement, debugging, or visualization.

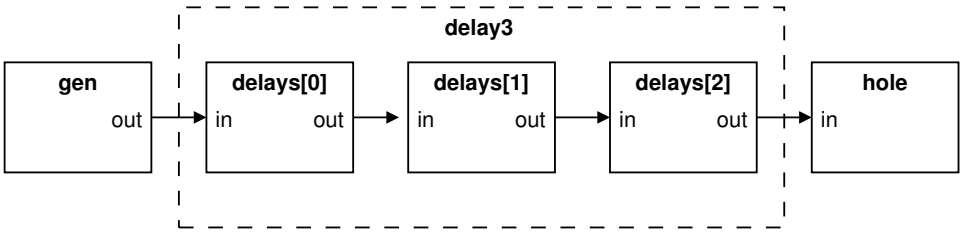
Static Model Analysis - the ability to analyze the model for optimization and user convenience. Example: type inference to resolve polymorphic port types.

Capability	Static Structural		Concurrent-structural OOP	
	Theory	Practice	Theory	Practice
Parameters	yes	yes	yes	yes
–Structural			yes	yes
–Algorithmic	yes	yes	yes	yes
Polymorphism	yes	yes	yes	yes
–Parametric	yes		yes	yes
–Overloading	yes	yes		
Instrumentation	yes		yes	
Static Analysis	yes	yes		

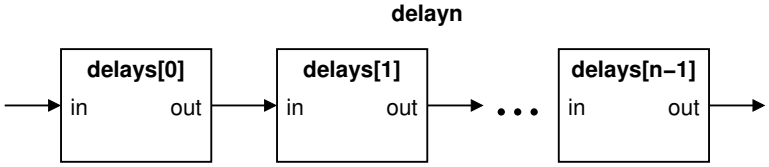
Table 4.1: Capabilities of existing methods and systems.

These capabilities fall into two categories, the first category contains capabilities that allow components to attain enough flexibility so that they can adapt to variations across designs and thus be reused. The second category of features include those that are needed to reduce the overhead of building and using flexible components. These features are needed because the degree of reuse achieved by a component in practice in dramatically affected by how easily it is configured to match the needs of a design model [44]. In the list shown in Table 4.1, all items but static analysis are there to enable sufficient component flexibility. Static analysis is needed to reduce the overhead associated with using the other features.

The following two sections relate the above abilities the two most common structural 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 chapter will identify which of the above capabilities are supported in each of these methods and also highlight potential pitfalls present in these methods. Chapter 5 will use the insight gained to describe how to build a concurrent-structural modeling system (in particular, the Liberty Simulation Environment) that overcomes the pitfalls in both these systems while retaining almost all the advantages. Table 4.1 can be used as a reference during the discussion.



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



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

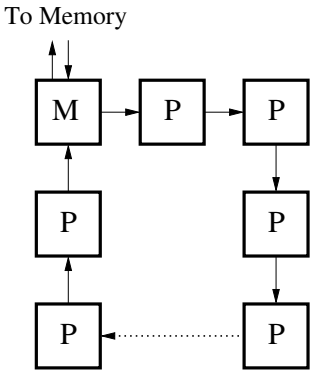
Figure 4.2: Block diagrams of chained delay components.

4.2.1 Static Structural Modeling

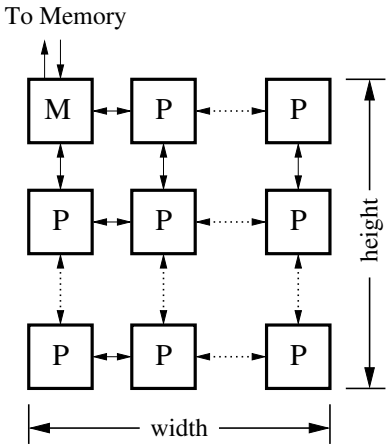
Static structural modeling systems are concurrent-structural modeling systems that statically describe a model's overall structure. Models in these systems often resemble netlists of interconnected components, and typically these tools have drag-and-drop graphical user interfaces to construct models. Examples of such tools are Ptolemy II with the Vergil interface [28], HASE [12], MIMOLA [64], and UDL/I [2].

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 [28] and type inference to resolve the polymorphic types [61]. Models could even be instrumented using aspect-oriented programming (AOP) [30] 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 4.2(a).



(a) Block diagram for the ring topology.



(b) Block diagram for the grid topology.

Figure 4.3: Block diagrams of a multi-processor component supporting grid and ring topologies and a variable number of processors.

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 4.2(b), where the length of the chain is a parameter since static structural systems provide no mechanism to iteratively connect the output of one block to the input of the next a parametric number of times. As a result, to permit flexibility, this simple hierarchical design would have to be discarded in favor of a more complex implementation of a primitive component implemented using a sequential programming language.

Implementing the primitive component for this simple example may not be difficult, but consider the implementation of a parameterized component capable of modeling the class of multiprocessor configurations shown in Figure 4.3. This component would need a parameter to control the topology of the processor interconnection (to select between a grid or a ring) as well as parameters to control the number of processors in the ring and parameters to control the height and width of the grid in the grid topology. This flexible component would also have to be modeled as a primitive component since parameters cannot control a hierarchical component's sub-structure, just as in the previous example. However, building a primitive component is tantamount to writing an entire multiprocessor simulator in a sequential programming language. As shown in Chapter 2, building a single processor model in this fashion is difficult. A multiprocessor model simply compounds the problems further.

Note that some static structural modeling systems may provide idioms for common patterns, such as chained connections, grids, or rings. However, the fundamental lack of general mechanisms to parametrically control model structure through any algorithms a user may choose still remains. This deficiency ultimately restricts the flexibility of components built hierarchically and forces users to build large primitive components.

4.2.2 Modeling with Concurrent-structural Libraries in OOP

A promising concurrent-structural modeling approach 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 (e.g., the SystemC [38] approach). 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 par-

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

Figure 4.4: Concurrent-structural OOP pseudo-code for an n-stage delay chain.

ticular 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 4.2(b)) seen in the last section could be built by composing n single-cycle delay components as shown in the pseudo-code in Figure 4.4. The complex multiprocessor component shown in Figure 4.3 could be built by composing processor and memory controller components as shown in Figure 4.5.

Unfortunately, run-time composition of structure provides component flexibility by precluding static analysis of model structure. This makes using these flexible components

```
 1  class MultiProcessor {
 2    public InPort responseIn;
 3    public OutPort requestOut;
 4
 5    Processor[] procs;
 6    MemoryControllor memctrl;
 7
 8    MultiProcessor(int numProcs, int height, int width,
 9                   enum {ring, grid} topology ) {
10      procs = new Processor[numprocs];
11
12      memctrl.requestOut.connect(requestOut);
13      responseIn.connect(memctrl.responseIn);
14
15      if(topology == ring) {
16        for(int i = 0; i < numprocs - 1; i++) {
17          procs[i].dataOut[0].connect(procs[i+1].dataIn[0]);
18        }
19        memctrl.dataOut.connect(procs[0].dataIn);
20        procs[numProcs-1].dataOut[0].connect(memctrl.dataIn[0]);
21      } else if(topology == grid) {
22        memctrl.dataOut[0].connect(procs[0].dataIn[0]);
23        procs[0].dataOut[0].connect(memctrl.dataIn[0]);
24        memctrl.dataOut[1].connect(procs[x.width-1].dataIn[2]);
25        procs[x.width-1].dataOut[2].connect(memctrl.dataIn[1]);
26
27        for(int x = 0; x < width; x++) {
28          for(int y = 0; y < height; y++) {
29            if(x != 0 || y != 0) {
30              if((x - 1 >= 0) && (x - 1 != 0 || y != 0)) {
31                procs[y*width+x-1].dataOut[0].connect(
32                  procs[y*width+x-2].dataIn[1]);
33                procs[y*width+x-2].dataOut[1].connect(
34                  procs[y*width+x-1].dataIn[0]);
35              }
36              /* Repeat above for x+1, y-1, and y+1 */
37            }
38          }
39        }
40      }
41    }
42  };
```

Figure 4.5: Concurrent-structural OOP pseudo-code for a multi-processor component.

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 infer these values by analyzing the structure of the machine. 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 [41], 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.

4.2.3 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 [14] modeling environment supports algorithmic specification and component overloading by running type inference at run-time. However, Balboa and

its type inference algorithm do not support parametric polymorphism [15]. The remaining chapters in this dissertation use the Liberty Simulation Environment (a new concurrent structural modeling tool) as a vehicle to describe how to gain the full benefits of both static structural modeling and concurrent-structural OOP modeling in practice.

4.3 Control Abstraction

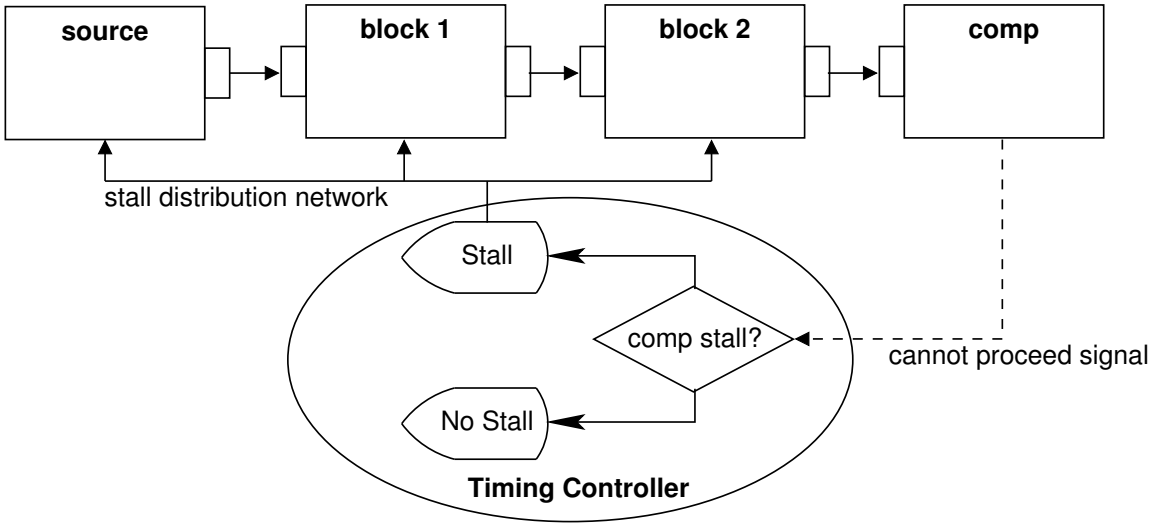
Up to this point, this chapter has focused on identifying the reuse-enabling features present in existing systems. However, facilitating rapid accurate modeling requires that a system also provide mechanisms for simplifying parts of a design that do not benefit from reuse.

Timing control (i.e., the portion of the control logic that detects stall conditions and distributes stall signals in a hardware designs) is a complex, and thus difficult to specify, portion of designs because the timing control logic of a design depends on global system knowledge. This global knowledge makes it unlikely that a timing controller for one design will work for a even a slightly different design; even small local design changes can drastically affect what computation and signals a controller must generate.

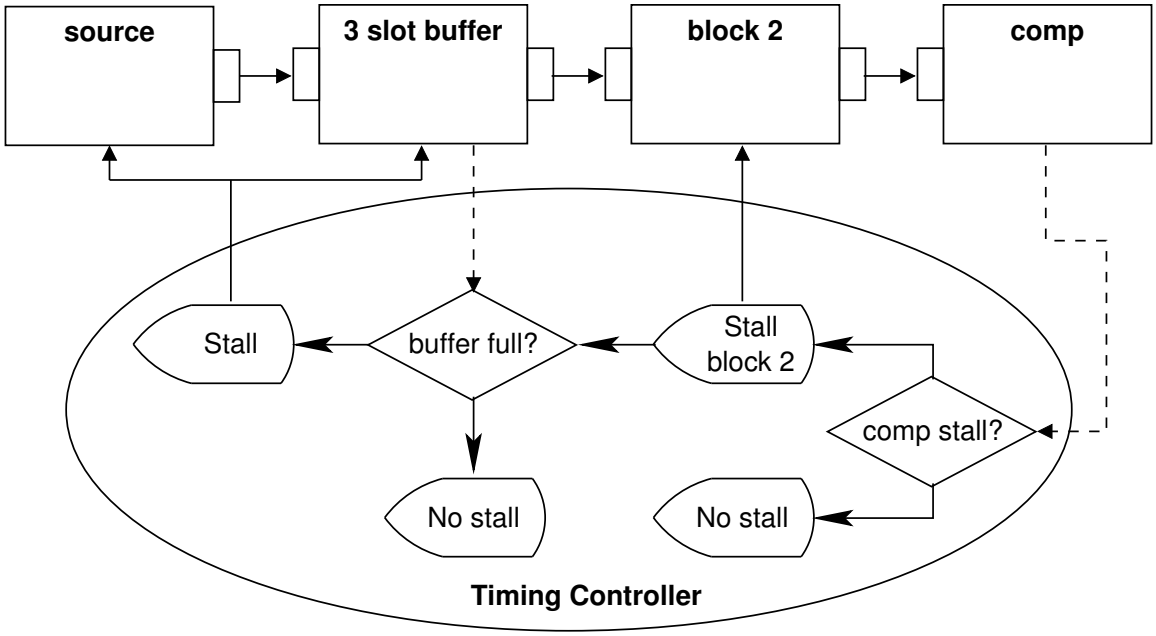
To see this, consider the two similar datapaths shown in Figure 4.6. The first datapath is shown Figure 4.6(a). Suppose that the `source` component generates a message on each cycle (neglecting stalls). Also, suppose that, within the same clock cycle, `block 1` performs computation on this data and passes on the results to `block 2`, which in turn passes its results to the block labeled `comp`. Also, suppose that the `comp` block then examines this data and determines whether it can process the data this cycle. Finally suppose that the `source`, `block 1`, `block 2`, and `comp` components update their state using their inputs at the clock edge[1] (again neglecting any stalls).

Now, suppose that the semantics of the computation performed by the depicted datapath prevents `comp` from proceeding with its computation when it receives certain inputs when

[1]The common clock signal feeding all components in the diagram is not shown.



(a) Simple datapath



(b) Simple datapath with 3-slot buffer.

Figure 4.6: A pair of slightly different datapaths.

in certain states. When this condition occurs, `comp` will assert the "cannot proceed" signal, shown in the figure. When this signal is asserted, we would like all previous blocks in the data path to stall so that the operation can be retried next cycle. (This back-pressure type of stall is very common in processor designs.) The logic to accomplish this is shown in the flowchart in the figure. The controller shown will generate a stall signal whenever the "cannot proceed" signal is high (i.e., the controller will detect that a stall is needed) and then distribute this stall signal to the appropriate components, in this case `source`, `block 1`, and `block 2`. This distribution is achieved via the stall network shown in Figure 4.6(a).

Now consider the similar datapath shown in Figure 4.6(b). This datapath is identical to the previous one, except that `block 1` has been replaced by a 3-slot buffer. In this datapath, when `comp` cannot proceed, `block 2` should be stalled, but `source` and the buffer should only be stalled when the buffer is full. The logic to handle this variant of back-pressure stalling is shown in the flow chart in the oval Figure 4.6(b).

Notice that both the stall distribution network and the timing controller are significantly different, despite only a small change in the datapath. In more complex specifications, with hundreds of components, similar small changes can require even larger changes to the control logic. From this, it is clear that a monolithic timing controller will be difficult to reuse. Furthermore, modularizing the controller into smaller reusable components is also difficult since it uses global knowledge of the datapath and system semantics.

Existing concurrent-structural systems provide no mechanism to simplify specification of timing control nor a mechanism to modularize it. However, due to the shortcomings described earlier in this chapter, the majority of components in these systems are not reused in practice anyway, thus having abstractions to reduce timing control specification overhead is less critical; reimplementing components from scratch, including the timing controller, is the norm.

Since this dissertation is focused on systems that will see wide reuse of components in practice, the need for control abstraction becomes more pressing. Section 5.4 will discuss

a timing-control abstraction that allows modularization and reuse of a large portion of the timing-control logic across designs.

4.4 Summary

Concurrent-structural modeling systems are those in which users build models by specifying and interconnecting concurrently executing components, in a similar fashion to when specifying the actual hardware. *High-level* concurrent-structural modeling systems are those in which the behavior of these components are specified via standard function invocation based code; contrast this with low-level systems in which this behavior is specified via constructs such as bit-wise logical operators.

Concurrent-structural modeling systems eliminate the mapping problem (described in Chapter 2) because they simply eliminate the manual mapping. As a result, the encapsulation provided by components is identical to that of hardware blocks.

With the mapping problem eliminated, reusing components across a range of designs (i.e., component-based reuse) to amortize component construction costs is attractive. However, even similar hardware designs often have minor variations in the behavior of similar components. Thus, to be reusable, components must be parameterized so that their behavior can be adapted to fit the peculiarities in each design. Unfortunately, the number and complexity of parameters that are needed for each reusable component can make it too difficult to understand and use in practice. While static analyses can infer parameter values, concurrent-structural systems that allow practical construction of flexible components are designed in a way that precludes these static analyses.

In addition to this unfortunate trade-off, existing concurrent-structural systems do not provide a means to modularize or simplify the specification of computations that require global knowledge. Timing control (i.e., the portion of the control logic that detects and distributes stalls) is a particularly complex portion of a design that falls in this category.

Chapter 5 describes how to design a language that overcomes both these shortcomings. The section describes this in the context of the Liberty Simulation Environment, the system in which these ideas have been first implemented.

Chapter 5

The Liberty Simulation Environment

Up to this point, this dissertation has focused on analyzing and cataloging the strengths and weaknesses of existing systems. This analysis culminated in Chapter 4 with a list of features needed for low-overhead reuse in concurrent-structural systems and a detailed explanation of why building a system that incorporates these features is challenging.

This chapter presents design elements that allow a concurrent-structural system to simultaneously provide *all* the capabilities described in Chapter 4. While some of these design elements are interesting in isolation, many of the major ones are interesting only when the system is considered as a whole because many challenges arise due to design element *interactions*. Additionally, each design element is easier to understand in a concrete setting. Thus this chapter presents the aforementioned design elements as a whole, in the context of the Liberty Simulation Environment (LSE) and its structural specification language, the Liberty Structural Specification language (LSS). During the discussion of this system and language, the chapter highlights the novel design elements and the problems they address. This chapter will also highlight *implementation* challenges created by the design choices made. These challenges will be addressed in subsequent chapters. In addition to major design elements, this chapter will also cover minor design elements that cumulatively make life easier for both users and builders of flexible components.

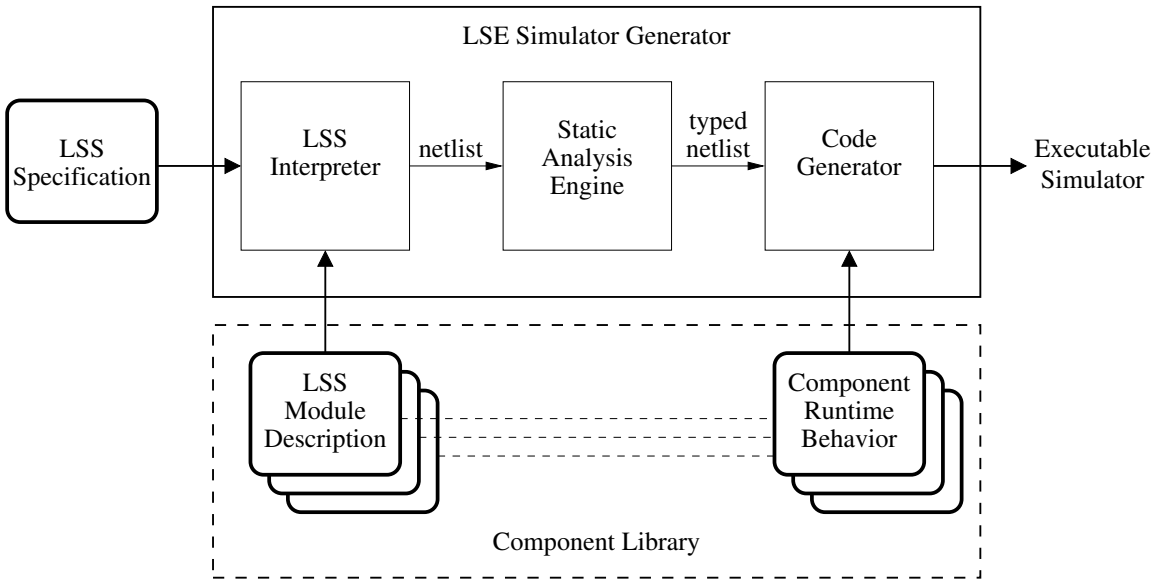


Figure 5.1: Overview of the simulator generation process in LSE.

5.1 Specification and Compilation of Components

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.

LSE avoids the mapping problem by allowing components to communicate structurally (like other pure concurrent structural systems), 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 4.4. However, unlike modeling in concurrent-structural OOP, the LSS code only describes the model's structure and *not* its run-time behavior. Run-time behavior is specified in a separate description language.

This separation is a key design element that allows model structure to be used for compile-time static analysis. Since behavior code is separate from LSS code, LSS code

can be evaluated as shown in in Figure 5.1, thus generating the system's static structure at compile time. Currently, LSE analyzes this structure to reduce specification overhead (described in Chapters 6 and 7) and to optimize the built simulator performance [41].

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 (but not its behavior). 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. During this discussion, take note of how leaf modules separate specification of interface from behavior.

5.1.1 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 5.2(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 leaf module behavior code is separated from the structural specification for two reasons. First, by separating behavioral code, a leaf module's code can be written in a purpose built behavior specification language (BSL)[1], optimized for specifying how values arriving

[1] At the time of this writing, LSE uses a stylized version of C as the BSL, but the LSS language and the

```
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.

Figure 5.2: Delay element declaration and use in LSS.

on input ports are combined with internal state to produce values on the instance's output ports (instead of LSS which is optimized for specifying structure and customization). Second, since the BSL code is evaluated separately, the BSL code can use the module parameter values specified by the user in LSS and the parameter values that were inferred by LSS, to customize the behavior of a particular instance.

Figure 5.2(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 5.2(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

techniques presented in this dissertation are not dependent on the specific BSL used.

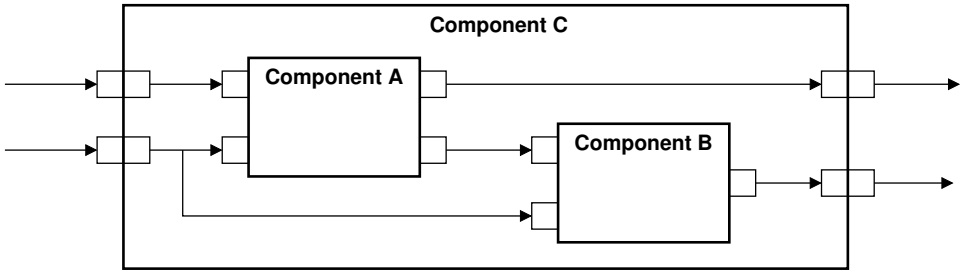


Figure 5.3: Hierarchical component composition in LSS.

Figure 5.2(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.

5.1.2 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 5.3). These module sub-instances execute *concurrently* and define the hierarchical module's behavior.

5.2 Low-overhead Reuse in LSE

This section describes LSE's features that allow for easy reuse of flexible components. When discussing LSE features, the text highlights challenges in the features' implementation. Later chapters will discuss the details of technology developed to address these challenges. During the discussion, notice how many of these challenges arise because of the inclusion of two or more existing well-known language features.

5.2.1 Structural Parameters

Recall from Chapter 4 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 5.4(a). This code defines a module that models an arbitrary depth delay pipeline (Figure 4.2(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 sub-instances 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 4.2(b). Notice how the general purpose C-like for-loop causes the length of the connection chain to vary with the parameter `n`.

Figure 5.4(b) shows how the `delayn` module can be used to create a 3-stage delay

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

(a) The LSS module declaration.

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

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

Figure 5.4: n-stage delay chain declaration and use in LSS.

pipeline. The module is instantiated on line 3, its `n` parameter is set on line 5, and finally the instance is connected on lines 7 and 8. The block diagram of the resulting system is the same as in Figure 4.2(a).

5.2.2 Extending Component Behavior

Chapter 4 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 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 of the same instance, 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. Note that run-time variables and this userpoint are unique features of LSS, needed because of the separate behavior and structure compilation phases.

5.2.3 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.

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*. Since scalable interfaces are the common case for flexible components, having every port behave in this fashion reduces the overhead of specifying components.

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 5.5 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 5.4. While the `delay` module (which was used to build the `delayn` module) supports multiple connections to its `in` and `out` ports,

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

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

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

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

Figure 5.5: Modified `delayn` module and a sample use.

the `delayn` module internally connects only one port instance to the chain of `delay` modules. If a connection were made to more than one port instance of either port on the `delayn` module, it would be ignored since it is internally unconnected.

Figure 5.5(a) shows the `delayn` module extended to support connections to multiple port instances, and Figure 5.5(b) shows a sample use of the module. Notice that many connections are now made from the `in` port to the head of the delay chain (line 19 of Figure 5.5(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 5.5(b)), the parameter would have the value `5`.

LSS also supports leaving ports unconnected (a port interface with zero size). A module can detect whether or not a port is connected and customize its behavior accordingly. These *unconnected port semantics* allow modules to have rich communication interfaces without burdening a user with the responsibility of connecting all the ports. Without such a feature, it may be tempting to replicate simple functionality rather than reusing a complex module with many ports that are unnecessary for a given situation.

Parameterized Interface Definition

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

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

Figure 5.6: Use-based specialization exporting additional parameters

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 5.6. 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 2.3 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 Chapter 6.

5.2.4 Polymorphic Interfaces

In addition to flexible interface definitions, LSE also supports modules with polymorphic interfaces. To support a robust set of flexible components, a system should support two types of polymorphism (as described in Chapter 4): parametric polymorphism and component overloading. Accordingly, LSS supports both forms of polymorphism.

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 5.4(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 5.4(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:

```
 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  };
```

Figure 5.7: An overloaded ALU module interface.

```
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 datatype independent.[2] 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.

Component Overloading

Component overloading is useful when defining a component that supports more than one data type on a particular port, but needs to be customized based on which type is actually used. Component overloading is supported with *disjunctive types*. A disjunctive type, denoted as `type1 | type2` in LSS, specifies that the entity with this type may statically have type `type1` or `type2`, but not both simultaneously.[3] Depending on which type is actually selected, a different module implementation will be selected.

[2] 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.

[3] 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.

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 5.7. Each port of the ALU is defined using the disjunctive type `int | float` (lines 2,3, and 5 of Figure 5.7). Lines 7 and 8 in this example force all the ports of the component to have the same type.[4] 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.

Type Inference

In order to reduce the designer's overhead in using polymorphic components, polymorphism can be resolved via type inference based on the structure of the model, as is done in LSS. 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 type inference problem in the presence of disjunctive constraints. The problem is also NP-complete which suggests that it may be prohibitively expensive

[4] note that *''portname* is a type variable that is constrained to be the type of the corresponding port. The double-tick is used to avoid name-space conflicts with user-defined type variables.

to implement. Fortunately, a heuristic inference algorithm can keep compile times reasonable. Details regarding the LSS type inference problem, a proof of its NP-completeness, and details of the heuristic type-inference algorithm will be provided in Chapter 7.

5.2.5 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) [30]. 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.

5.3 A Comprehensive LSE Example Specification

Figure 5.8 shows a sample LSS description of the writeback stage of the machine shown in Figure 2.3(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 Chapter 5.4, connections in LSE are always point-to-point. Thus, two `tee` modules need to be instantiated (lines 6 and 7) to handle the common data bus net and the ALU output net because these nets must fan out. Lines 17-30 connect the various ports on the modules to the appropriate ports on other

```
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 }
```

Figure 5.8: An LSE specification of the writeback stage of the machine shown in Figure 2.3, with an additional connection from the ALU to the LSU.

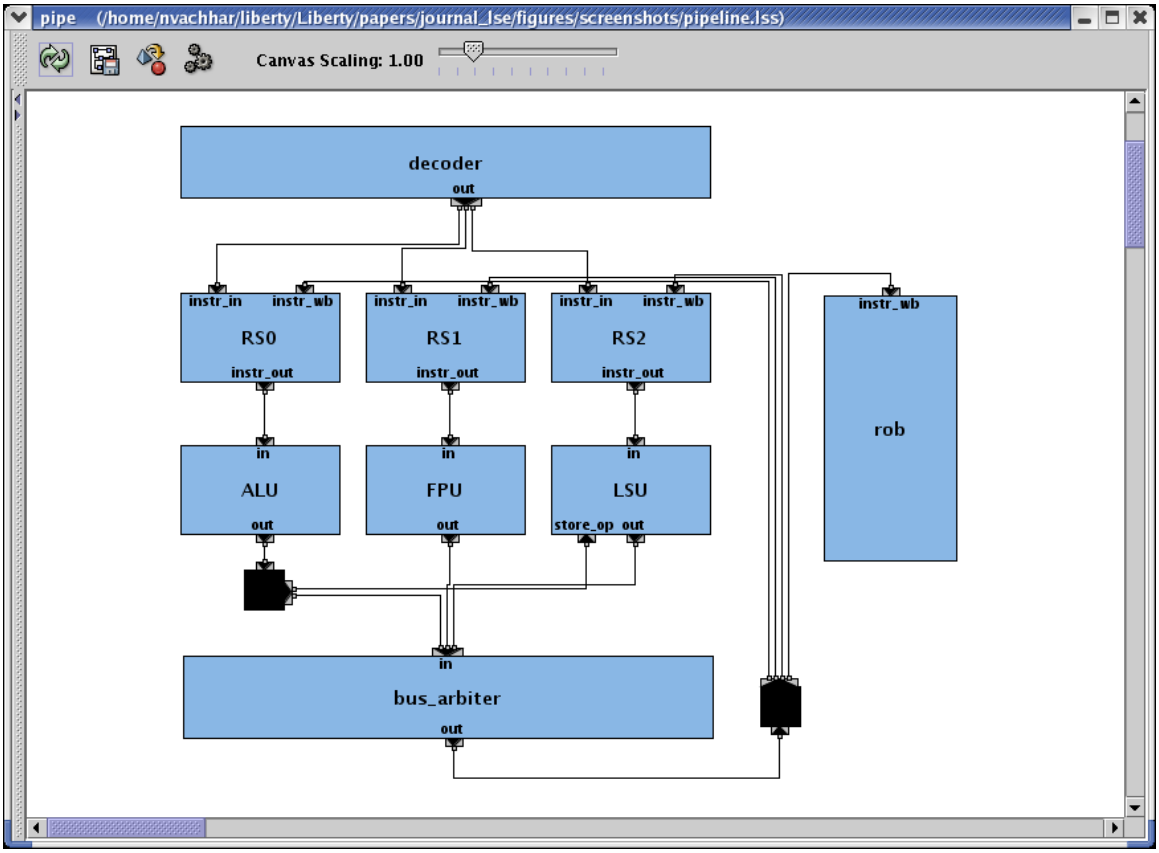


Figure 5.9: Screenshot of the LSE Visualizer showing the model from Figure 5.8.

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

```
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 >>>;
```

Figure 5.10: Customization of an `arbiter` module for round-robin arbitration

compile time, the structure of the described machine can be visualized in a graphical tool, as shown in Figure 5.9, and can be analyzed so that the generated simulators may be optimized [41]. 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 5.10 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 5.10, a general arbitration module is instantiated. 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 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

```
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 }
```

Figure 5.11: Collector to monitor bus arbiter output

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 5.11. 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 [32].

5.4 Timing Control Abstraction

The previous part of this chapter describes how 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. A concrete example of this phenomenon was shown in Chapter 4.

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.

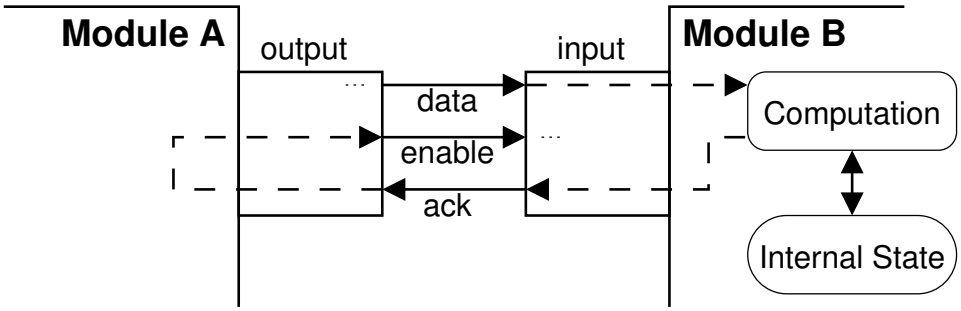


Figure 5.12: Connection with standard control flow semantics.

Since LSE is designed as a microarchitecture-level rather than ISA-level 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.

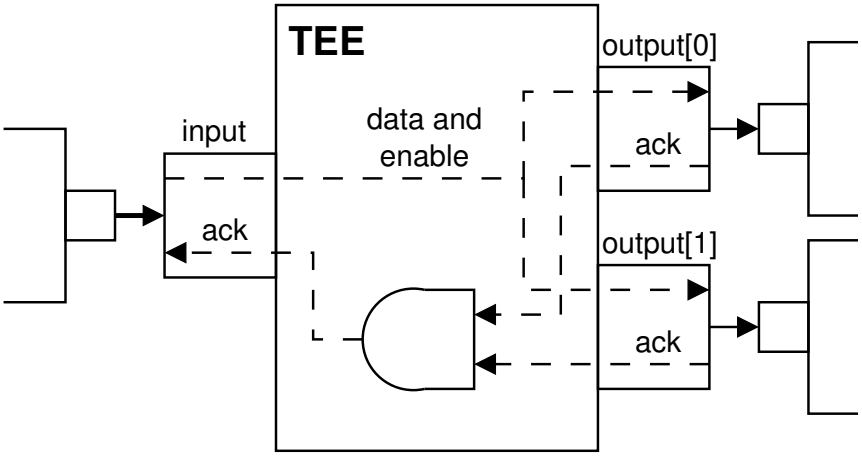
To distribute stall information, regardless of whether it is structural or semantic, LSE exploits the connectivity of the components in the system. Components that generate stalls can communicate this information to their upstream neighbors. Those that do not need or do not generate stall information 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.

Analogous to asynchronous hardware, each connection in an LSE description actually specifies three subconnections as shown in Figure 5.12: a DATA and an ENABLE signal in

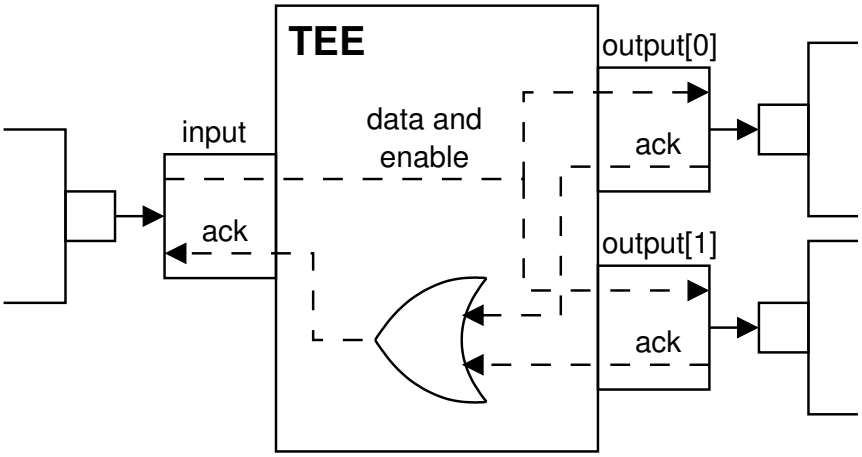
the forward direction and an ACK signal in the reverse direction. As its name implies, the DATA subconnection carries the data from the sending module (Module A in the figure) to the receiving module (Module B). The ENABLE signal, when asserted, indicates that the transmitted data should be used to perform state update. Finally, the ACK signal indicates that the receiving component is able to accept and process the sent data.

A typical communication, which all takes place in a single clock cycle, is shown in Figure 5.12. In this communication, 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 5.13(a). By default, it takes the incoming ACK signal from all of its outputs, computes the logical AND of these values, and passes that result out through the ACK wire on the input port. With these semantics, a module connected to the input of a `tee` 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 5.13(b). With these new semantics, the sending module will see

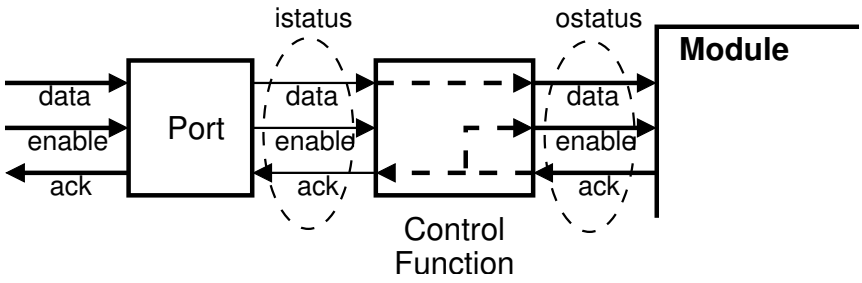


(a) Default Semantics



(b) Modified Semantics

Figure 5.13: The `tee` module's control semantics options.




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

Figure 5.14: Control function overriding standard control.

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 to inject a semanitc stall 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 that require 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 5.14, 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 5.8. 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 5.14 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.

Note that these control semantics were subsequently re-developed by the MicroLib project, an effort to build reusable component libraries for SystemC [42]. The fact that they independently came to exactly the same control signaling protocol suggests that it is indeed a robust, yet manageable, mechanism for abstracting the timing controller.

5.5 Experience with LSE

This section provides evidence to support the thesis that the described techniques, embodied in the Liberty Simulation Environment (LSE), do indeed yield a system that allows users to rapidly construct high quality models. To do this, the section first describes the short model development times experienced by LSE users and the accuracy of a some validated models. Second, the section briefly describes some of the benefits of the static analyzability of LSE models. Third, the section highlights the quality of models produced by presents some general experiences with LSE. Finally, the section presents data that quantifies the amount of component-based reuse observed across LSE models. Later chapters in this dissertation will evaluate the effectiveness of particular LSE features in isolation.

5.5.1 Modeling Speed and Accuracy

To date, LSE has been used to model a variety of microarchitectures in a number of research groups. Within the Liberty Research Group, it has been used to model several machines including an IA-64 processor core, a chip multiprocessor model that utilizes that core, two Tomasulo-style machines that execute the DLX instruction set, and a model that is cycle-equivalent to the popular SimpleScalar [4] `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.

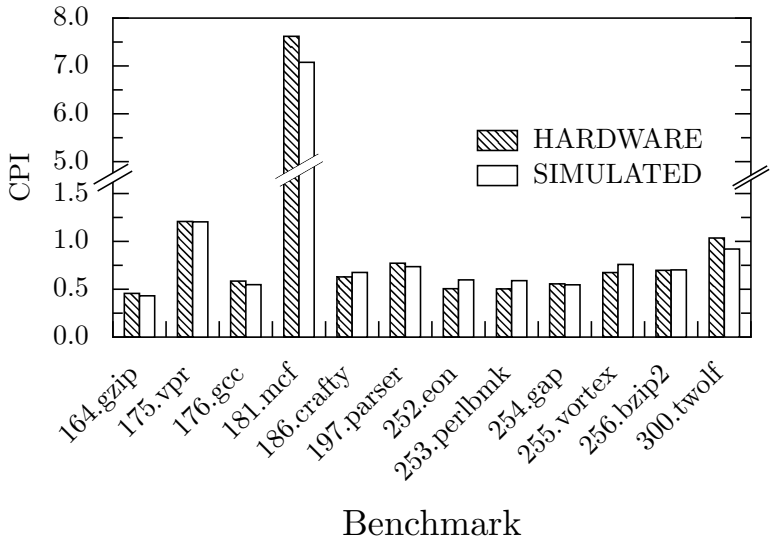


Figure 5.15: Performance of actual Itanium 2 hardware versus model predictions.

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 short. By comparison, SimpleScalar represents at least 2.5 developer-years of effort [3]. 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.

A model of particular interest is a validated Itanium 2 simulator built by a single student in only 11 weeks. Figure 5.15 shows the performance predictions of this model versus the actual hardware for a range of benchmarks.[5] As can be seen from the figure, not only was the model built rapidly, but it is quite accurate, predicting the performance to within 3% on average. The CPI of hardware was measured using the Itanium 2's performance counters while running the SPEC benchmark on the reference input. The simulator CPI was computed by running benchmarks to completion using the SMARTS sampling system [60]. The sample size was 1000 instructions with 4000 instructions of warmup per sample; The period of the sampling was set to achieve at least 10,000 samples. Note that the output of each run was checked against the reference outputs to ensure correct execution.

[5]The Itanium 2 model and the corresponding data presented in Figure 5.15 were provided courtesy David A. Penry

While these models and development times provide compelling evidence in support of LSE's efficacy and this thesis, the models described were constructed by users directly involved in the development of LSE. The true test is how well external users fare. Fortunately, external users have fared well with LSE. For example, students at Rice University used LSE in the SPINACH project to construct a validated model of 3COM's TIGON-2 network interface controller. This model is also accurate to within a few percent in predicting packet throughput and was constructed by two students in approximately 1.5 months [59]. The students were also able to rapidly construct a model of a future high-performance 10 Gigabit ethernet NIC containing a variable number of processors and complex interconnect networks [59].

The model development times for all of these models is significantly shorter than those reported for similarly detailed models with similarly sized modeling teams. The validated models are also accurate. Prior efforts to build validated models have more time and not achieved the same level of accuracy [13].

5.5.2 Benefits of Static Analyzability

To allow high-quality models, one of the major goals in the design of LSE was to ensure that model structure was still analyzable at compile-time. As we have seen, the structure of the model can be used to reduce the designer effort required to instantiate reusable components by inferring parameter values based on structure. There are many other benefits, two of which are discussed below.

Visualization

Since the model structure (i.e., the interconnection of components) is one of the most important aspects of a structural model, ideally, this structure should be easy to specify and understand. While a textual format often works best for specification, a visualization is better for understanding structure. Static analyzability enables a separate tool, such as LSE's

visualizer developed by the Liberty Research Group, to automatically produce such a visualization. We have already seen one such visualization in Figure 5.9. Blome et al. present many other visualizations and discuss how the visualizer can be used to improve student understanding of architectural concepts via LSE models [6].

Simulator Optimization

A common drawback of concurrent-structural models constructed with large numbers of reusable components is that simulators built from these models often execute too slowly to be useful. LSE's static analyzability, however, allows its simulator generator to build an optimized simulator from a model. During simulator generation, the generator is aware of each module's parameters and interconnection patterns. The generator uses this information to optimize leaf module BSL code specializing the modules based on how they are used. For example, in the BSL code, counted loops whose bounds are parameter values may be fully unrolled, optimized, and then scheduled for increased instruction-level parallelism (ILP).

To further improve performance, LSE's simulator constructor also exploits the structure of the model to produce an optimal execution schedule for the concurrency in a model. Edwards presents an algorithm to produce schedules that minimize the worst-case execution time of systems, such as LSE, that use the heterogeneous synchronous reactive model of computation [16]. This algorithm relies on compile-time knowledge of the interconnection of components to understand the possible data flow among components. The algorithm operates by first partitioning a model's signal data-dependence graph into a set of strongly-connected components (SCCs). SCCs are scheduled in topological order and each SCC is then partitioned into a head and tail sub-graph. The head is given an arbitrary schedule; the tail is scheduled by a recursive application of the algorithm. If an optimal partitioning is performed in each recursive application, the schedule contains the minimum number of component invocations in the worst case. Unfortunately, finding such an optimal parti-

tioning leads to unreasonably large run-times (the problem is NP-complete) and does not optimize the metric of interest for simulators: average-case execution time.

LSE employs a variant of this algorithm, developed by Penry and August, that maximizes average case execution time and avoids exponential run-time [41]. In this variant, if finding an optimal partitioning for a graph proves too costly, the scheduler inserts a dynamic sub-schedule corresponding to this tail graph in the overall static schedule. This avoids the exponential scheduling times. Furthermore, the graphs that are difficult to partition also lead to the most pessimistic schedules in the original algorithm; every *potential* data dependence in the graph must be satisfied. In Penry and August's approach, since the embedded sub-schedule is dynamic it can detect when all data dependences have been satisfied at run-time. The embedded dynamic schedule uses this knowledge to invoke components only as many times as needed to satisfy *true* data dependences. This usually resolves signal values in fewer invocations than the worst-case schedule for difficult to partition sub-graphs.

With its optimizing simulator generator and scheduler, LSE is currently able to provide performance on par with existing concurrent-structural systems, despite aggressive reuse of flexible components.

5.5.3 General Experiences

While the primary impetus for LSS and LSE was a desire to build a practical platform a rapidly constructing accurate models, the tools have proved capable of improving the quality of models in other ways as well. In particular, the structural modeling methodology informs researchers and designers about unanticipated design issues. The following few paragraphs will relate a few experiences where LSE allowed users to gain interesting insight into the architectures being modeled.

Renaming with Predication

When building the first out-of-order Itanium models using LSE, a student working on the model had an interesting question about how renaming and predication[6] interact. The out-of-order Itanium machine being modeled was to perform register renaming at an early stage in the pipeline. The student working on the model was curious as to how unknown predicate values should be handled in the early rename stage. Traditional renaming schemes assume that the producer of a particular register value in a dynamic instruction stream is uniquely known early in the pipeline (neglecting any potential branch mispredictions). Predication violates this assumption since an instruction only defines the value of its destination register if the predicate register has the value 1. Mechanisms to address this problem have been explored in the literature [11, 33, 58, 63] but there is still no widely accepted solution.

From this example, we see that the structural model highlighted a critical design concern that had been overlooked. Not all models give this benefit. In previous, non-structural models, this issue had been masked because the predicate value was produced early, stashed in a global data structure, and then used assuming the value would be ready in the final hardware. In the structural model, it was clear that there were cases where the instruction producing the predicate value was still in flight at the rename stage and thus no predicate value would be available.

Mysterious Performance Counters

During the construction of the Itanium 2 model (Model F described earlier), one of the strategies used to validate the simulator was to implement virtual hardware performance counters in the simulator and compare the values of these performance counters with the

[6] A predicated instruction set architecture allows each instruction to source a one bit register, called a predicate register, that indicates whether or not an instruction should execute. For example, an add instruction $rd \leftarrow rs_1 + rs_2$ would also take a third predicate source operand, ps, denoted $rd \leftarrow rs_1 + rs_2 < ps >$. If the value in the register denoted by ps is 0 when the instruction executes, then the execution of the add instruction is suppressed and rd keeps its old value, otherwise, rd gets updated as it would in a non-predicated instruction set architecture.

performance counters in the hardware. This process proved quite effective in building a validated model, as seen from Figure 5.15.

During the initial study of the architecture's memory subsystem, some performance counters frustrated all efforts to decipher their precise meaning; the documentation from Intel also appeared to be in error. However, late in the development of the model, the student working on the model came to understand the role of these mysterious performance counters. Based on this understanding, the model was made to match the values of some of these counters.

The counters were actually documented correctly, however, the stalls they were counting were present to ensure correctness due to obscure corner cases. The details of the stall condition are difficult to understand without a detailed understanding of both the IA-64 ISA and the Itanium 2 microarchitecture. Building the structural model provided this level of understanding. Reading documents about the microarchitecture did not.

It is unlikely that building a sequential simulator would have illuminated the meaning of these counters because the details would likely have been approximated away; the counters monitor a condition that arises due to complex concurrent behavior that is extremely tedious to map to a sequential language. This "approximation without understanding" can lead to serious oversights when designing a microarchitecture. Structural models seem to naturally highlight difficult to implement areas of a design and the modeler is made aware of any blind approximations.

5.5.4 Quantity of Component-based Reuse

Table 5.1 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 these numbers are over-conservative. To improve code clarity, some models in the table contain a large number of trivial modules that exist

Table 5.1: 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
F	183	18 (3)	19	4.95 (8.32)	82%	18
Total	1324	69 (19)	39	12.26 (22.83)	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 [4].
D An out-of-order processor core for IA-64.
E Two of the cores from D sharing a cache hierarchy.
F A validated Itanium 2 simulator

solely to wrap other components. Since these modules are quick to write, usually taking less than 5 minutes per module, models that use large numbers of them are unnecessarily penalized. The numbers in parentheses in the table show reuse statistics when these modules are ignored. Notice that ignoring these modules greatly improves Model A's, Model B's, and Model F's reuse statistics.

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 23 times, indicating considerable reuse.

The significant reuse described in the table is largely a result of LSE's features to reduce specification overhead. The SimpleScalar model, which was built before many of the LSS features were available, contains the largest number of 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 dissertation 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 chapters will describe the novel techniques needed to implement LSE and its features for low-overhead reuse. In these chapters, the features discussed will be evaluated on the above models to determine each feature's contribution to the reduction of specification overhead.

5.6 Summary

Designing a concurrent-structural system that supports static analysis of models while also allowing the construction of flexible hierarchical components is non-trivial. However, by selecting the appropriate features and managing their interactions, it is possible to construct such a system.

Compile-time (i.e., static) knowledge of model structure is possible, even when allowing use of arbitrary code to build that structure. This is accomplished by separating code that specifies component behavior (i.e., the port I/O relation) from code that specifies model structure (i.e., component instantiations, interfaces, and interconnections). This separation permits a two-phase compilation process. In the first phase, the structure specification code is evaluated to produce the model netlist. In the second phase, the compiler uses this information to infer parameter values and other model properties. This information is then used to parameterize and compile the model's behavior code and produce the final simulator.

One way to use the static information to parameterize a component is to automatically instantiate concrete types for polymorphic components. This type inference is based on the simple observation that two connected ports must have the same data type. Unfortunately, in the presence of component overloading, this kind of type inference is non-trivial. In fact, Chapter 7 will show that the type inference problem is NP-complete and requires a specially designed heuristic algorithm to make type inference practical.

Another way to use the static information is to allow a component to customize itself based on how it is connected and used in the system. For example, the system can automat-

ically discover the number of read ports a register file component should have by observing the number of connections the user makes to the read register interface of of the component. The hierarchical register file component can then customize its sub-structure based on the number of connections the system reports. Unfortunately, this type of *use-based specialization* presents a dilemma if a hierarchical component's constructor uses arbitrary code to specifying its sub-structure. This issue is addressed in Chapter 6.

In addition to use-based specialization and type inference, knowledge of static structure enables other useful features. These include automatic visualization of said structure, the ability to insert probes into the model by referencing its structure, and the ability to optimize generated simulators based on parameter values and connection patterns. The compiler-directed optimizations are the most important because they are needed to ensure that simulators built from models are efficient [41].

In addition to requiring low-overhead use and construction of flexible components, modularizing timing control (i.e., the portion of control logic that detects and distributes stalls), despite its global nature, is a critical design element in a system that enables reuse in practice. Fortunately, by observing that timing control can be partitioned into stall distribution, semantic stall detection, and structural stall detection, it is possible to reuse a good portion of the logic. Structural stall detection code can be rolled into pre-existing reusable components, and knowledge of the model datapath can be exploited to automatically generate the stall distribution logic. To avoid artificial restrictions, users should be allowed to override any inferred control behavior. Automatically generating semantic stall detection logic in a general way is left to future work. However, once detected, these stalls can be injected into the automatically constructed distribution network.

All the features described above (the contributions of this dissertation) drove the design of and are incorporated in the Liberty Simulation Environment (LSE) and its structure specification language, the Liberty Structural Specification language (LSS). Users of this system have experienced rapid model development times, improved model accuracy, and

revelations about their designs. For example, a single student was able to construct a model of Intel's Itanium 2 processor in only 11 weeks; the model predicts hardware CPI to within 3%. This and other results presented in this chapter show that LSE gains the benefits desired; the tool is general (i.e., not limited to a particular class of processors), permits substantial reuse of components, yields models that are easier to validate, and permits users to gain insight into their design by building the model.

The remaining chapters in this dissertation will address implementation challenges highlighted in this chapter, draw conclusions based on the results of this work, and outline promising future directions.

Chapter 6

Implementation of Use-Based Specialization

As was mentioned in Section 5.2.3, 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 chapter describes the implementation challenges raised by use-based specialization and presents the implementation of use-based specialization in LSE.

6.1 Evaluation Semantics

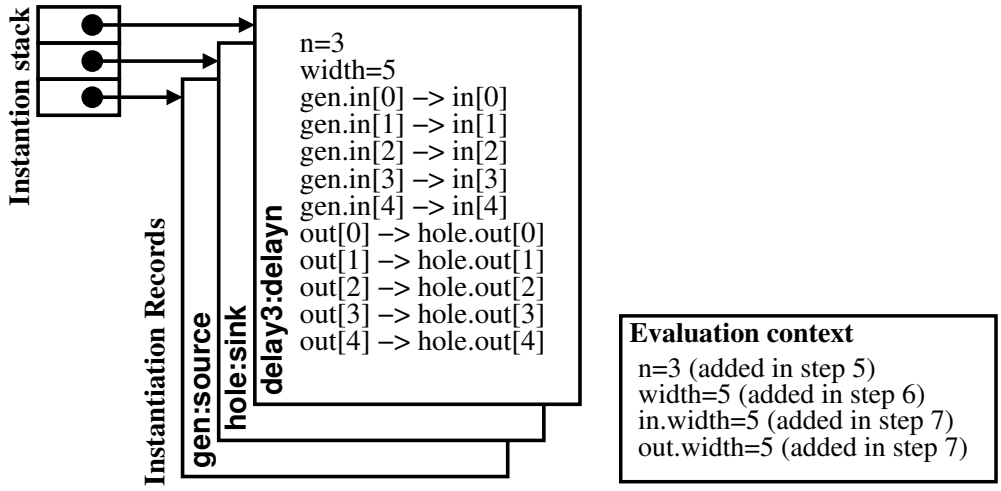
Use-based specialization requires that the module body has access to values that are defined by the usage of the module instance (e.g. the number of connections to a port and values of all explicitly specified parameters). However, use-based specialization also allows the module's interface (i.e. the module's ports and parameters) to depend on the same values.

Thus, use-based specialization requires deferring module body evaluation until after the module is instantiated and used, however, conventional evaluation requires that the module body be evaluated before it can be used so that its interface is known. To remedy this circularity, LSS uses the novel evaluation semantics described in this section.

Clearly, evaluation of the module body cannot occur as soon as an instance is created since the module body depends on the values of the module's parameters. Thus, rather than invoking the module body when creating a new instance, the name of the newly created instance and the module from which it was instantiated are pushed onto an instantiation stack.

Code continues to execute from the current module body, and whenever an assignment to a subfield of a sub-instance (e.g. Line 5 in Figure 5.4) is encountered, the assignment is recorded as a potential parameter assignment. In a similar fashion, whenever any connection is made to a subfield of a sub-instance, the connection is recorded as a potential port connection.

When the current module body finishes evaluation, the instance at the top of the instantiation stack is popped off and its module body executed. Whenever the module body declares a parameter, 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 gets 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 completes, the potential subfield assignment and potential connection records are checked to make sure no non-existent parameters or ports were referenced for this instance. Additionally, all the parameters are checked to ensure that they have some value.



(a) Instantiation stack. (b) Context for delay3.

Figure 6.1: LSS interpreter state.

The following example will illustrate the execution of the code shown in Figure 5.4.

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

2. Line 2-3. The same is done for the `hole` and `delay3` modules.

3. Line 5-6. The interpreter records the assignments to potential parameters `n` and `width`.

4. Line 8-9. The interpreter records the connections made.

At this point, the top-level code has finished and so the module at the top of the instantiation stack is popped and its constructor evaluated. Figure 6.1a 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 5.5.

5. Line 2. Check to see that `n` has an integer value in the record. If so, add it to the evaluation context based on the value in the record. The evaluation context is shown in Figure 6.1b.

6. Line 3. Do the same for the `width` parameter.

7. Line 5-6. Record the fact that connections to the `in` and `out` port are valid, compute the port widths, and add the *portname*.width parameters to the context.

8. Line 12-21. Execute the code, recording the instantiations, assignments, and connections as before.

Once this code is finished, the interpreter ensures that no connections were made to non-existent 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` modules pushed onto the stack during the evaluation of the instance `delay3`, is popped off the stack and evaluated.

Note that these semantics are not the same as lazy evaluation [25]. The semantics above *require* that the evaluation of a module body be delayed until the instantiating module body completes. There are no language primitives that allow an instantiator to read the values of data produced by the instantiated module body. Contrast this with lazy evaluation in which expression evaluation is delayed until the results of a particular evaluation are requested (usually implicitly using the the result of the expression).

The above semantics can be specified more formally by describing the execution semantics for the LSS language as an evolution of program states. Execution semantics expressed in this way are typically called *small-step semantics* [24]. The complete small-step semantics has numerous state transition rules that closely resemble the semantics for common imperative programming languages. Therefore, in this chapter, only the state transitions that relate to the implementation of use-based specialization will be described. This chapter uses the notation and terminology that is used by Harper [24], unless defined differently below.

The 7-tuple, (M, I_s, L, A, B, e, S), will represent the state of an executing LSS program. M is the netlist of the design as it is known at the current point in program eval-

uation. 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 state transition function for LSS program states are expressed using propositional logic. In the transition rules below, items that appear above the horizontal bar represent the hypothesis of a logical statement. The items that appear below the bar represent the conclusions of the statement. The notation $q_0 \rightarrow q_1$ is used to denote that q_0 can transition to state q_1. The conclusions of all the transition rules identify all the legal transitions. Since any given state of an LSS program satisfies the hypothesis for at most one transition rule, the transition rules define the state transition function. More information on using propositional logic to represent program semantics can be found in the references [24].

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

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

This rule pushes the constructor for instance i onto the stack of constructors I_s that must be evaluated and continues evaluating the statements in the current statement list. (Note that $i \rhd I_s$ denotes a stack with the element i at the top and the stack I_s below it. $\mathsf{dom}(L)$ is an operator that returns the domain of L.) 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 instance constructor.

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

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, if $A \neq \emptyset$ after a module finishes evaluation, then an assignment or connection was made to an undeclared parameter or port.

6.2 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 6.1 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

Model Name	Inferred Port Widths	Connections
A	816	919
B	823	929
C	147	304
D	611	3975
E	984	4528
F	523	1395

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 [4].
D An out-of-order processor core for IA-64.
E Two of the cores from D sharing a cache hierarchy.
F A validated Itanium 2 simulator.

Table 6.1: Port width parameters inferred by use-based specialization

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 designer effort in using reusable components.

6.3 Summary

Explicitly specifying values for all parameters when using flexible, reusable components, can be so cumbersome that they are not reused in practice. Fortunately, it is possible to infer the values of many of these parameters, but doing so in a language that allows creation of flexible hierarchical components requires novel evaluation semantics. This is because, in traditional evaluation, a component cannot be used until it is instantiated. Upon instantiation, a component's constructor, which depends upon parameter values to create sub-structure, must be evaluated. However, with use-based specialization, the system must know how the component has been used before parameter values are known.

To resolve this circular dependence, this chapter presented novel evaluation semantics that delay the execution of the constructor from the time a component is instantiated to until after all information about component usage is known. In the interim, any operations involving the component are recorded into a table and checked for correctness after the component constructor has executed. To prevent any circularity in the evaluation order, an instantiating component may not *require* any values produced in an instantiated component's constructor.

Overall, use-based specialization is able to eliminate many redundant parameter specifications. In one particular configuration, this saves the user the trouble of keeping 984 interface-size parameters consistent with 4528 connections in the model. This savings reduces the overhead of using flexible components and makes them more likely to be reused in practice.

Chapter 7

Implementation of Type Inference

Using polymorphic components can burden users by 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 chapter, explicit type instantiations can still be burdensome. As was mentioned in Section 5.2.4, 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 chapter, 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 [56]. This chapter 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 chapter 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.

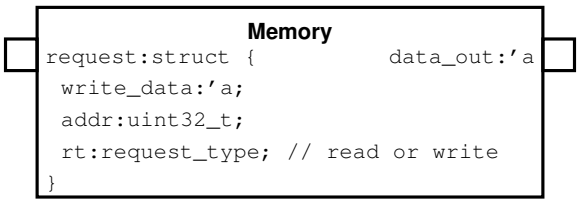
7.1 The Type System

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

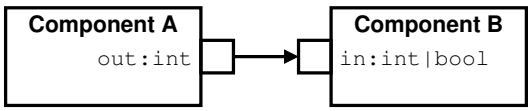
$$
\begin{array}{lllll}
\text{Basic Types} & \tau & ::= & \texttt{int} \mid \ldots \mid \tau[n] \mid \texttt{struct}\{i_1 : \tau_1; \ldots i_n : \tau_n;\} \\
\text{Type Variables} & \alpha, \beta, \gamma & ::= & 'identifier \\
\text{Type Schemes} & \tau^* & ::= & \alpha \mid (\tau_1^* \mid \ldots \mid \tau_n^*) \mid \tau \mid \tau^*[n] \mid \\
 & & & \texttt{struct}\{i_1 : \tau_1^*; \ldots i_n : \tau_n^*;\}
\end{array}
$$

Here, the basic types are the standard programming language types ($\tau[n]$ is the array 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 5.2.4.

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 7.1(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



(a) Port interface for a simple memory.



(b) Connected components

Figure 7.1: Components with types annotated.

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.

7.2 The Type Inference Problem

To automatically resolve the polymorphism for the type system described in Section 7.1, 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 7.1(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 compatible 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:

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

$\top$ 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.

To prove NP-completeness of the type inference problem, the following decision problem is considered.

Problem. *Given a constraint ϕ, is it possible to assign a compatible basic type for each type scheme in ϕ such that each term of ϕ is simultaneously satisfied?*

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 [39]. 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 [39].

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 show how to reduce any instance of the monotone 1-in-3 SAT problem, a known NP-complete problem [19], to the type inference problem. 3-SAT 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. Monotone 1-in-3 SAT is a 3-SAT 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 $b_i, b_j, b_k \in B$. 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

$$
\begin{aligned}
'S_0 &= \texttt{struct} \ \{ \texttt{x:int; y:bool; z:bool;}\} \\
'S_1 &= \texttt{struct} \ \{ \texttt{x:bool; y:int; z:bool;}\} \\
'S_2 &= \texttt{struct} \ \{ \texttt{x:bool; y:bool; z:int;}\}
\end{aligned}
$$

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. To ensure that boolean variables in the clause get the appropriate value based on which satisfying assignment is chosen, add the following constraint for each clause:

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

If the inference problem for this set of constraints has a solution, the corresponding monotone 1-in-3 SAT problem is satisfiable. If not, the problem is unsatisfiable. Thus, solving the inference problem solves the corresponding monotone 1-in-3 SAT problem. □

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

Proof. The problem is in NP and is NP-hard, thus by definition it is NP-complete. □

The type system and inference problem presented here is similar to other type systems and inference problems. The type system is, in fact, very similar to that of Haskell. However, the Haskell problem is undecidable in general [49]. There exist restricted versions of the type system that are decidable [36, 48]. Unfortunately, of these, the restrictions that yield acceptable computational complexity [36] are not desirable in a structural modeling environment since they forbid common port interface typings. For other restricted versions of the type system, we know of no heuristic algorithms that are appropriate for instances of the problem that arise in the structural modeling domain.

The Balboa [14] structural modeling system shares a similar type system and inference problem but since Balboa only supports component overloading and not parametric polymorphism, its inference algorithm [15] 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 [61] is also not directly applicable.

The type inference problem also seems like it could be easily mapped to SAT. However, a straight-forward mapping with a bounded width binary encoding for each type that is uniform across all problem instances is not possible. This is because one can arbitrarily nest `structs` and arrays leading to an unbounded number of types across all problem instances. It should be possible to identify a bounded width binary encoding for the set of types that can occur in a particular problem instance [43], however, this is also non-trivial and left to future work.

7.3 The Basic Inference Algorithm

Fortunately, the substitution algorithm used for languages such as ML [34] 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 must be a rule

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

Table 7.1: Simple Substitution Algorithm

that handles each production in the grammar defining the constraint. The standard rules are shown in Table 7.1. 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 $(\tau^* =$

$\tau_1^* \mid \ldots \mid \tau_n^*) \wedge \phi$, 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 Harper's proofs for the corresponding theorems about MinML [24].

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 7.1 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:

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

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 7.1 are almost identical to those for programming languages such as ML and the truth of this theorem is well known for those transitions [24].

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.

7.4 Type Inference Algorithm Heuristics

As Section 7.5 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 and two heuristics, inspired by heuristics used for pruning the search tree in many other tree-based algorithms, which dramatically reduce the

number of subproblems that need to be solved, which in turn drastically reduces run-time.

7.4.1 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.

7.4.2 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 sim-

plifications 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.

7.4.3 Partitioning the Constraint

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

$$(\texttt{'a} = \texttt{int}|\texttt{bool}) \quad \wedge \quad (\texttt{'a} = \texttt{int}|\texttt{char}) \wedge \tag{7.1}$$

$$(\texttt{'b} = \texttt{int}|\texttt{bool}) \quad \wedge \quad (\texttt{'b} = \texttt{int}|\texttt{char}) \wedge \tag{7.2}$$

$$(\texttt{'c} = \texttt{int}|\texttt{bool}) \quad \wedge \quad (\texttt{'c} = \texttt{int}|\texttt{char}) \tag{7.3}$$

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 (7.1), (7.2), and (7.3) 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.

7.5 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 7.2. 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 measured. Table 7.2 also presents these results. The model specification language compiler and type inference algorithm are implemented in Java. The run times are all measured using

Table 7.2: 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
F	101	38	> 12 h	2.76s

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 [4].
D An out-of-order processor core for IA-64.
E Two of the cores from D sharing a cache hierarchy.
F A validated Itanium 2 simulator.

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.

7.6 Summary

Many blocks in a hardware design perform tasks that are independent of the data they are manipulating. Queues, memories, arbiters, and other routing logic often falls into this class of component. To enable a single component to be reused across all instantiations of these blocks, a modeling system should support parametrically polymorphic components. Unfortunately, each data type parameter in a polymorphic component adds overhead to the use of the component since the user is forced to specify a type instantiation for each type parameter. Fortunately, this overhead can be eliminated by utilizing commonly known type inference algorithms that infer these type instantiations from the component usage [62].

Other blocks used in hardware often have similar functionality, but can only operate on a fixed class of data types. For example, a set of ALUs that have the same operations but with each operating on a different floating point format. A system supporting low-overhead reuse can allow the user to instantiate an *overloaded* ALU component and then automatically select the ALU implementation with the correct data types based on the context in which the ALU is used [15].

Supporting both of these features simultaneously is possible, but non-trivial. This chapter showed that the type inference problem is NP-complete when both overloading and parametric polymorphism are supported. The chapter then presented a type inference algorithm that can infer types in this scenario. Finally, the chapter provided heuristics that keep type inference times reasonable for models seen in practice. Overall type inference itself saves the user considerable effort, For example, in one model evaluated, 108 of 115 type parameters were inferred. The heuristics reduced the type inference time for this model from more than 12 hours to 6.54 seconds.

Chapter 8

Conclusions and Future Directions

This dissertation focused on techniques to enable rapid construction of models through low-overhead component-based reuse while supporting compilation of the model to an efficient simulator. This chapter draws conclusions based on the ideas presented in this dissertation and discusses future work.

8.1 Conclusions

Many previous approaches have attempted to build systems that allow rapid simulator construction. However, some have been constructed with reuse and generality as a secondary criterion. Other have examined reuse and rapid construction but have focused on one or two large reuse features and excluded others.

This dissertation, on the other hand, focuses on reuse and rapid model construction but takes a more holistic approach and identifies a set of features that all work synergistically. For example, the control abstraction discussed in this dissertation is fairly useless if users must re-implement components due to the inability to build polymorphic or highly parameterized components; in this scenario, custom control interfaces are a small burden compared to the overall cost of reimplementation. Conversely, polymorphic parameterizable components generally fail to adapt to the range of control interfaces needed across a

range of designs in the absence of an adequate control abstraction. It is only when both the ideas are taken together that they are truly effective.

To evaluate the quality of these ideas, they were implemented as the Liberty Simulation Environment (LSE). Results and experience with LSE show that this synergy allows users to rapidly construct models by exploiting component-based reuse. Users have used LSE to rapidly construct models and perform design-space exploration. They have reported model development times for complex models that are well over an order of magnitude shorter than conventional approaches.

8.2 Future Directions

While the work in presented in this dissertation provides a good foundation for design-space exploration, it is far from complete. A host of new capabilities, features, and techniques is required to build the ultimate design tool. Some of these simply require implementation, engineering, and perseverance. Other challenges, some of which are described below, are much more daunting.

High-speed Simulation

The work presented here and embodied in LSE provide a good foundation for design-space exploration, but simulation speed of concurrent-structural systems remains a challenge that has not been fully addressed. Currently, aggressive optimizations and concurrency scheduling allow LSE to provide performance on par with existing concurrent-structural systems, despite aggressive reuse of flexible components. This is achieved through the use of compiler optimizations for module code and by exploiting the static schedulability of the heterogeneous synchronous reactive model of computation, for which LSE employs a novel static scheduler that maximizes average case, instead of worst-case execution [41]. These compiler optimizations and scheduling rely on the ability of the description compiler to

transform the generated simulator code based on compile-time knowledge of component parameters and interconnections. Providing a set of techniques by which a system can support practical component-based reuse while still permitting this type of compile-time analysis was a major goal of this work.

Unfortunately, concurrent-structural systems, including LSE, still yield simulators that are over an order of magnitude slower than many hand coded approaches. It should be possible to reclaim much of this lost performance through more aggressive structure based analysis and a more advanced scheduling system. This dissertation, provides the foundations of such work; Penry and August [41] provide a novel scheduling algorithm and apply a number of existing techniques leveraging this foundation, but much of the research and engineering required to fully close performance gap remains undone.

Validation

As seen in this work, moving to a concurrent-structural paradigm with validated reusable components can allow rapid construction of an accurate model. Unfortunately, much work remains to be done to validate models. There is a body of work on verifying the correctness of processor specifications with regard to instruction set semantics [8, 29]. However, work that specifically verifies performance properties of the processor is lacking. Worse still, it is often unclear which design factors truly influence program performance until the model is built. This is due to the fact that no general analytical models exist to describe the interaction of various processor components and running programs. Much work is still needed in this area, especially in the formalisms used to describe processor performance. With these formalisms in place, work can begin on systematic validation and verification of design performance criteria. A first step is to develop a method to systematically validate the performance predictions of a high-level model and then ensure that the design is consistent with the high-level model at each lower level of abstraction.

High-level Models as a Design Entry Point

Another major challenge is providing an environment that behaves as a design entry point to later design stages. SystemC provides such an environment for traditional behavioral modeling without reuse. The SystemC methodology relies on a one-by-one refinement of each component to lower and lower levels of abstraction until a synthesizable RTL level description is realized. After each component is refined. This strategy is very similar to the ones proposed by other EDA researchers, such as those working on System Verilog. This strategy, however, requires that components have interfaces that support this lowering.

For rapid design-space exploration, this is problematic. The appropriate interfaces for refinement usually require low-level data types and firm protocols between components so that the interface remains constant through lowering. In the design methodology described in this dissertation, high-level design and the interfaces and protocols are an abstraction of the actual low-level signaling interface. Lowering the data types during refinement is not an option since the lowered component will then be incompatible with the non-lowered components. To obtain a full working model designers are forced to globally lower all components. In addition to data types, the component interface itself is also an abstraction of the low-level interface in order to extend the reusability of components. In particular, the control abstraction presented in this dissertation is essential for high degrees of reuse. Unfortunately, this abstract interface does not resemble the true control interface in the final hardware. Thus lowering requires massive interface changes, again destroying the incremental lowering strategy. Some user guided automated lowering or synthesis strategy is required for a lowering methodology to work.

Appendix A

Questions Measuring the Clarity of Simulators

This appendix details the questions asked of subjects to obtain the data presented in Section 2.2.

A.1 Background Questionnaire

As discussed in Section 2.2, each subject was given a questionnaire to ascertain their background. The questionnaire, show, each was given a questionnaire to determine their familiarity with computer architecture, programming languages, and existing simulation environments, in particular, SimpleScalar. The questionnaire appears below.

1. How many years of experience do you have in computer architecture or related areas?
2. If you have a PhD, did you do your PhD thesis in computer architecture?
3. Have you ever written a simulator for a processor microarchitecture?
    (a) Please list some of the major features of the microprocessor (e.g., out-of-order issue, speculation, etc.)

(b) In which language/tool was the simulator built (e.g., C, LSE, Verilog, ...)

4. Have you ever designed your own processor core?

   (a) Please list some of the major features of the microprocessor (e.g., out-of-order issue, speculation, etc.). If more than one, choose the most advanced.

   (b) Was the design fabricated or was it part of a course project where the chip was never laid out and fabricated?

5. Which publicly available/commercial processor simulation tools have you used?

6. How long have you been using the C or C++ programming language?

7. How often do you program/look at C/C++ code?

8. How long have you been using the Liberty Simulation Environment (LSE)?

9. How often do you program/look at LSE code?

10. Which of the given exercises did you complete?

A.2 Exam Questions

Below are the control and experimental questions asked of subjects in the study described in Section 2.2. Note that the text describing the purpose of each question was not revealed to the subjects. Also, note that placeholders are used for the line numbers since different subjects received different machine models, as discussed in Section 2.2.

Control Questions

A.2.1 Chained Delay

Purpose

This question covers basic LSE syntax and semantics to ensure that the subjects understand LSE. The specifications are simply delay chains, the first has a delay of 3, the second a delay of 5.

Question

(a) In the configuration below, if a signal is present on port in of instance delay in cycle 0, on what cycle does the data appear on the port out of instance delay?

```
module delayelement {
  using corelib;

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

  instance delay1:delay;
  instance delay2:delay;
  instance delay3:delay;

  in -> delay1.in;
  delay1.out -> delay2.in;
  delay2.out -> delay3.in;
  delay3.out -> out;
};

instance delay:delayelement;
```

(b) If a signal is present on port in of instance delay in cycle 0, on what cycle does the data appear on the port out of instance delay?

```
module delayn {
  using corelib;

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

  var i:int;
  parameter n:int;

  if(n <= 0) {
    punt("Error, parameter n must be greater than or equal to 1");
  }

  instance delays:instance ref[];
  for(i=0;i<n;i++) {
    delays[n] = new instance(delay,"delay${i}");
  }
  in -> delays[0];
  for(i=0;i<n-1;i++) {
    delays[i].out -> delays[i+1].in;
  }
  delays[n-1].out->out;
};

instance delay:delayn;
delay.n=5;
```

A.2.2 Hierarchical Modules

Purpose

This question tests users understanding of hierarchical modules and userpoints in LSE. The mystery specification `foo` is a hierarchically constructed demultiplexor. The `in1` signal is

the data input, the `in2` signal is the control that selects which output will get the input, and the `out` signal is the output of the demux. All subjects had seen this example previously, if they had completed the exercises designed to help them understand LSE. Note that the demux used in real specifications is a leaf module in the LSE library.

Question

This question will center around the following module description:

```
module foo {
  using corelib;

  inport in1:bool;
  inport in2:int;

  outport out:bool;

  instance fanout1:tee;
  instance fanout2:tee;
  instance gates:instance ref[];

  in1 -> fanout1.in;
  in2 -> fanout2.in;

  var i:int;
  for(i=0;i<3;i++) {
    gates[i] = new instance(<<<gates${i}>>>, gate);
    fanout1.out -> gates[i].in;
    fanout2.out -> gates[i].control;
    gates[i].out -> out;
    gates[i].gate_control = <<<
      if(LSE_signal_extract_data(cstatus[0]) ==
         LSE_signal_something) {
```

```
      return (*cdata[0]) == ${i};
     } else if(LSE_signal_extract_data(cstatus[0]) ==
              LSE_signal_nothing) {
      return 0;
     } else {
      return -1;
     }
    >>>;
  }
};
```

(a) In a few words please describe the input/output behavior of the data signals of this module. (You don't need to worry about the enable and ack signals).

(b) In the following configuration that uses the module defined above, what is printed out by the collector in timestep 0. What is the data status (LSE_port_something, nothing, or unknown) of each in port (in[0], in[1], ...) at the end of cycle 0. (Note that the data collector is only called if data is received on the port).

```
using corelib;

instance input1:source;
instance input2:source;
instance munger:foo;
instance hole:sink;

input1.create_data=source::create_constant("TRUE");
input2.create_data=source::create_constant("1");

input1.out ->:bool munger.in1;
input2.out ->:int munger.in2;

LSS_connect_bus(munger.out,hole.in,3);
```

```
collector in.resolved:hole {
  record = <<<
    if(LSE_signal_data_is_known(status) &&
       LSE_signal_data_is_present(status) &&
       !LSE_signal_data_is_known(prevstatus)) {
      printf(LSE_time_print_args(LSE_time_now));
      printf(": porti=%d %d\n", porti, *datap);
    }
  >>>;
};
```

Experiment Questions

A.2.3 Issue Windows and Reorder Buffers

Purpose

This question is designed to identify how easy it is to identify large machine structures in a simulator. The answer to this question depends on which version of the specifications the subjects saw.

Question

(a) In the machine modeled in the configuration question3/machine1.xxx, Examine file question3/machine1.xxx lines nnn through mmm. 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.xxx, lines nnn to mmm.)

A.2.4 Branch Resolution Policy

Purpose

This question is designed to measure how easy it is for subjects to identify policy decisions normally thought of as sequential algorithms, such as what happens when a branch needs to commit. For this question, the correct answer depends on the version of the configuration the user saw.

Question

In the machine modeled in the file question5/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 mmm-nnn of pipestages.xxx)

(b) In what stage of the pipe are mis-speculated branches identified?

(c) Is this the same stage which initiates branch recovery (Recovery is initiated when the machine stops fetching from the wrong path)?

(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?

Bibliography

[1] S. G. Abraham and S. A. Mahlke. Automatic and efficient evaluation of memory hierarchies for embedded systems. In *Proceedings of the 32nd annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pages 114–125, November 1999.

[2] H. Akaboshi. *A Study on Design Support for Computer Architecture Design*. PhD thesis, Department of Information Systems, Kyushu University, Japan, January 1996.

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

[4] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, February 2002.

[5] B. Black and J. P. Shen. Calibration of microprocessor performance models. *IEEE Computer*, 31(5):59–65, May 1998.

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

[7] D. G. Bradlee, R. R. Henry, and S. J. Eggers. The Marion system for retargetable instruction scheduling. In *Proceedings of the ACM SIGPLAN 1991 Conference on*

Programming Language Design and Implementation (PLDI), pages 229–240, June 1991.

[8] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessors control. In *Proceedings of the sixth International Conference on Computer-Aided Verification (CAV)*, volume 818, pages 68–80, Stanford, California, USA, 1994. Springer-Verlag.

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

[10] L. Charest and E. M. Aboulhamid. A VHDL/SystemC comparison in handling design reuse. In *Proceedings of 2002 International Workshop on System-on-Chip for Real-Time Applications*, July 2002.

[11] W. Chuang and B. Calder. Predicate prediction for efficient out-of-order execution. In *Proceedings of the 17th Annual International Conference on Supercomputing (ICS)*, pages 183–192. ACM Press, 2003.

[12] P. Coe, F. Howell, R. Ibbett, and L. Williams. A hierarchical computer architecture design and simulation environment. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(4):431–446, October 1998.

[13] R. Desikan, D. Burger, and S. W. Keckler. Measuring experimental error in microprocessor simulation. *Proceedings of the 28th International Symposium on Computer Architecture (ISCA)*, pages 266–277, July 2001.

[14] F. Doucet, M. Otsuka, S. Shukla, and R. Gupta. An environment for dynamic component composition for efficient co-design. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, 2002.

[15] F. Doucet, S. Shukla, and R. Gupta. Typing abstractions and management in a component framework. In *Proceedings of Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 115–122, January 2003.

[16] S. A. Edwards. *The Specification and Execution of Heterogeneous Synchronous Reactive Systems*. PhD thesis, University of California, Berkeley, 1997.

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

[18] A. Fauth, J. V. Praet, and M. Freericks. Describing instruction set processors using nML. In *Proceedings of the European Design and Test Conference (DATE)*, pages 503–507, March 1995.

[19] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W H Freeman & Co, New York, NY, 1979.

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

[21] J. C. Gyllenhaal. *An Efficient Framework For Performing Execution-Constraint-Sensitive Transformations That Increase Instruction-Level Parallelism*. PhD thesis, University of Illinois, Urbana, IL, 1997.

[22] G. Hadjiyiannis, S. Hanono, and S. Devadas. ISDL: An instruction set description language for retargetability. In *34th Design Automation Conference (DAC)*, pages 299–302, June 1997.

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

[24] R. Harper. *Programming Languages: Theory and Practice*. Draft, 2002.

[25] P. Henderson and J. James H. Morris. A lazy evaluator. In *Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages (POPL)*, pages 95–103, January 1976.

[26] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Francisco, CA, 1996.

[27] The IMPACT compiler. Web site: http://www.crhc.uiuc.edu/IMPACT, June 2004.

[28] J. W. Janneck, E. A. Lee, J. Liu, X. Liu, S. Neuendorffer, S. Sachs, and Y. Xiong. Disciplining heterogeneity – the Ptolemy approach. In *ACM SIGPLAN 2001 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, June 2001.

[29] R. B. Jones and D. L. Dill. Efficient validity checking for processor verification. In *IEEE International Conference on Computer-Aided Design (ICCAD)*, pages 2–6, 1995.

[30] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference for Object-Oriented Programming (ECOOP)*, pages 220–242, June 1997.

[31] R. Leupers and P. Marwedel. Retargetable code generation based on structural processor descriptions. *Design Automation for Embedded Systems*, 3(1):1–36, January 1998.

[32] Liberty Research Group. Web site: http://www.liberty-research.org/Software/LSE, August 2004.

[33] J. E. McCormick, Jr. Supporting predicated execution: Techniques and tradeoffs. Master's thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1996.

[34] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, MA, 1997.

[35] B. L. Noble and R. D. Chamberlain. Analytic performance model for speculative, synchornous, discrete-event simulation. In *Proceedings of the 14th Workshop on Parallel and Deistributed Simulations (PADS)*, May 2000.

[36] M. Odersky, P. Wadler, and M. Wehr. A second look at overloading. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pages 135–146, June 1995.

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

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

[39] M. S. Paterson and M. N. Wegman. Linear unification. In *Proceedings of the eighth annual ACM symposium on Theory of computing*, pages 181–186. ACM Press, May 1976.

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

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

[42] D. G. Pérez, G. Mouchard, and O. Temam. MicroLib: A case for the quantitative comparison of micro-architecture mechanisms. In *Proceedings of the 3rd Annual Workshop on Duplicating, Deconstructing, and Debunking*, June 2004.

[43] A. Pneuli, Y. Rodeh, and O. Strichman. Finite instantiations in equivalence logic with uninterpreted functions. Technical report, The Weizmann Institute of Science, Rehovot, Israel, 2001.

[44] W. Putzke-Röming, M. Radetzki, and W. Nebel. Objective VHDL: Hardware reuse by means of object oriented modeling. In *Proceedings of the 1st Workshop on Reuse Techniques for VLSI Design*, September 1997.

[45] W. Qin, S. Rajagopalan, and S. Malik. Flexible and formal modeling of microprocessors with application to retargetable simulation. In *IEEE/ACM Design Automation and Test in Europe (DATE)*, pages 556–561, March 2003.

[46] S. Rajagopalan, M. Vachharajani, and S. Malik. Handling irregular ILP within conventional VLIW schedulers using artificial resource constraints. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 157–164, November 2000.

[47] N. Ramsey and J. Davidson. Machine descriptions to build tools for embedded systems. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES), Lecture Notes in Computer Science*, volume 1474, pages 172–188, June 1998.

[48] H. Seidl. Haskell overloading is dexptime-complete. *Information Processing Letters*, 52(2):57–60, 1994.

[49] G. Smith. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming*, 23(2-3):197–226, 1994.

[50] D. J. Sorin, V. S. Pai, S. V. Adve, M. K. Vernon, and D. A. Wood. Analytic evaluation of shared-memory systems with ILP processors. In *Proceedings of 25th Annual International Symposium on Computer Architecture (ISCA)*, pages 380–391, April 1998.

[51] Y. N. Srikant and P. Shankar. *The Compiler Design Handbook: Optimizations and Machine Code Generation*. CRC Press, New York, NY, 2003.

[52] S. Swamy, A. Molin, and B. Convot. OO-VHDL: Object-oriented extensions to VHDL. *IEEE Computer*, 28(10):18–26, October 1995.

[53] D. E. Thomas and P. R. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, Norwell, MA, 1998.

[54] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11:25–33, January 1967.

[55] M. Vachharajani, N. Vachharajani, and D. I. August. The Liberty Structural Specification Language: A high-level modeling language for component reuse. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI)*, pages 195–206, June 2004.

[56] M. Vachharajani, N. Vachharajani, S. Malik, and D. I. August. Facilitating reuse in hardware models with enhanced type inference. In *Proceedings of the 2004 Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, September 2004.

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

[58] P. H. Wang, H. Wang, R.-M. Kling, K. Ramakrishnan, and J. P. Shen. Register renaming and scheduling for dynamic execution of predicated code. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 15–25, January 2001.

[59] P. Willmann, M. Brogioli, and V. Pai. Spinach: A Liberty-based simulator for programmable network interface architectures. In *Proceedings of the SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 20–29, June 2004.

[60] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hose. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, pages 84–97, June 2003.

[61] Y. Xiong. *An Extensible Type System for Component Based Design*. PhD thesis, Electrical Engineering and Computer Sciences, University of California Berkeley, 2002.

[62] Y. Xiong and E. A. Lee. An extensible type system for component-based design. In *Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 20–37, March 2000.

[63] S. A. Ziegler. Aggressive hardware support for predicated execution in out-of-order execution superscalar processors. Master's thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1993.

[64] G. Zimmerman. The MIMOLA design system: a computer aided processor design method. In *Proceedings of the 16th Annual Design Automation Conference (DAC)*, pages 53–58, 1979.