Microarchitectural Exploration with Liberty

Manish Vachharajani Neil Vachharajani David A. Penry Jason A. Blome David I. August
Departments of Computer Science and Electrical Engineering
Princeton University
{manishv, nvachhar, dpenry, blome, august}@cs.princeton.edu

Abstract

To find the best designs, architects must rapidly simulate many design alternatives and have confidence in the results. Unfortunately, the most prevalent simulator construction methodology, hand-writing monolithic simulators in sequential programming languages, yields simulators that are hard to retarget, limiting the number of designs explored, and hard to understand, instilling little confidence in the model. Simulator construction tools have been developed to address these problems, but analysis reveals that they do not address the root cause, the error-prone mapping between the concurrent, structural hardware domain and the sequential, functional software domain. This paper presents an analysis of these problems and their solution, the Liberty Simulation Environment (LSE). LSE automatically constructs a simulator from a machine description that closely resembles the hardware, ensuring fidelity in the model. Furthermore, through a strict but general component communication contract, LSE enables the creation of highly reusable component libraries, easing the task of rapidly exploring ever more exotic designs.

1. Introduction

Design-space exploration is an important technique used by microarchitects for both product design and research. Microarchitects designing a product must carefully evaluate the design-space surrounding a proposed design in order to realize a well-performing microarchitecture [1, 2]. Microarchitects researching novel techniques must evaluate them in multiple design contexts to understand their properties. In each case, architects must rapidly consider a wide range of options while maintaining confidence in their conclusions.

Most commonly, architects use simulators to evaluate a design's effectiveness. Unfortunately, the typical simulator construction methodology, hand-writing a simulator in a sequential language such as C or C++, is a poor choice for this design-space exploration. While writing a simulator the architect must map the microarchitecture, which is inherently structural and concurrent, to a sequential programming language with functional composition. This mapping process is laborious and error-prone and can often lead to simulators with problems: they can be *opaque* (obscuring the hardware being modeled), hard to reuse, difficult to modify, and inaccurate.

The community has created a variety of tools, from architecture description languages (ADLs) to simulation construction frameworks, aimed at making simulators easier to develop and reuse. Unfortunately, none of these tools solve the problem. Most fail because they do not address the root cause, the mapping between the concurrent, structural hardware domain and the sequential, functional software domain.

In this paper, we present the Liberty Simulation Environment (LSE), a simulator construction system designed from the ground up to shift the onus of microarchitecture mapping from architects, who can err, to a flexible automatic system. LSE descriptions are *transparent* (resemble the hardware being modeled), easy to reuse, easy to modify, and less likely to contain errors. LSE also establishes a contract between components to permit component level reuse, enabling the creation of highly reusable component libraries. Since LSE allows rapid construction of accurate simulators, it is an ideal tool for design-space exploration in both product development and research.

The rest of this paper is organized as follows. Section 2 explores in more detail why the manual mapping of microarchitectures to sequential programming languages causes errors in simulation. Section 3 explores the shortcomings of existing simulation tools that have made some progress alleviating the problems with sequential simulators. Section 4 presents the Liberty Simulation Environment (LSE). Section 5 describes our experience using Liberty and presents evidence that LSE achieves its goals. Section 6 addresses previous work in peripherally related areas.

2. The Sequential Mapping Problem

Writing and maintaining a microarchitectural simulator requires architects to simultaneously manage the design of complex hardware and manage the creation of a complex software system. In hardware design, architects use *modularity* and *encapsulation* to develop and test the system iteratively and systematically. To keep the simulator consistent throughout the design process with the hardware architects intend to build, the simulator and microarchitecture need to share a modularization strategy. This allows the simulator

to easily track design changes in the hardware. Unfortunately, the sequential execution and functional composition semantics of common sequential programming languages make this impossible.

When designing hardware, architects modularize the system's functionality into separate communicating hardware components. The architects agree on the inputs and outputs of each component and formalize them into a *communication interface*. This interface encapsulates the functionality of each component and exposes only the communication interface itself to the rest of the system. This encapsulation process can continue hierarchically since each component can internally divide the functionality into subcomponents. Changes can be made to parts of the system without affecting other parts of the system as long as the interfaces remain intact. This type of encapsulation and interconnection is called *structural composition*.

When hand coding a sequential simulator, architects must *map* this hardware design on to a programming language which modularizes functionality into functions. Architects compose or connect two components by invoking one function from within the body of the other. In this style of composition, called *functional composition*, the function's prototype, its argument list and return value, forms a piece of the component's communication interface with the rest of the system. Unfortunately, the remainder of the communication interface is determined *implicitly* by the functions it invokes. Arguments passed to a called function are *implicit* outputs of the calling component, and return values are *implicit* inputs to the calling component. Additionally, each function invocation not only augments the communication interface with additional inputs and outputs, but it also defines the recipients of the information. Since it is impossible to compose two components without function invocation, a component may not specify its communication interface independent of its functionality. Even worse, since function invocation may be guarded by control flow statements, a component's communication interface may even depend on *run-time* values.

To separate the interface definition from the specification of connectivity, simulator authors often use global state rather than function invocation to define communication paths. A function uses certain global variables as inputs and others as outputs. While this technique allows a component to anonymously communicate with other components, connectivity is still implicitly defined by functionality. Two components are connected if one writes to a piece of global state and the other reads that state before it is overwritten. Updates to this global state must be carefully timed to ensure consumers read the correct data. The exact order in which components are invoked determines which values consumers actually see. Component granularity also becomes important when using this communication style, since coarse grained components can cause loops in invocation dependences. For example, component A may need to be invoked before component B, but component B may need to be invoked before component A. In order to break such loops, it is often necessary to repartition the computation.

In general, as the number of components increases and the global state grows, manually managing the simulator's invocation order becomes prohibitively difficult. Thus, in order to regain control of the system, it is common to use a combination of global state as well as function invocation to communicate information through the system. Unfortunately, simulators written in this style inherit the shortcomings of *both* systems.

To see how the problems discussed above occur in practice, consider a simulator written in a sequential language that models a typical five stage superscalar processor pipeline. Figure 1a shows a typical main simulation loop for such a machine. 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 will begin computation at the back of the pipe and move toward the front so that later pipeline stages use state from previous cycles and earlier stages are aware of the availability of resources in later stages.

We now focus on the fetch and issue stages of the pipeline. Figure 1b and 1c show the computation performed in the fetch and issue stages respectively. From the pseudo-code, we see that as an instruction is sent to its functional unit, it is simultaneously removed from the instruction window (lines 8 and 13 in Figure 1c). Thus, when the code for the fetch stage is executed, it will see the newly created space in the instruction window.

Now, assume that the architects would like to change the behavior of the pipeline so freed slots in the instruction window are not available until the cycle after the instruction was issued. The hardware change corresponding to this architectural change may simply amount to removing any logic dealing with dequeuing from the computation of the `slots-available` signal. Such a change may be necessary, if, for example, the dequeue signals arrive too late in the cycle. Figures 1d, 1e, and 1f show the necessary changes to the simulator main loop, fetch logic, and issue logic, respectively.

Notice that a very small hardware change required a significant change to the sequential simulation code. These differences are indicated by the bars to the right of the line numbers in Figure 1. Specifically, the change to the microarchitecture required the architects to partition the issue logic, intermingling the code to remove the instruction from the instruction window with the code that scheduled the ex-

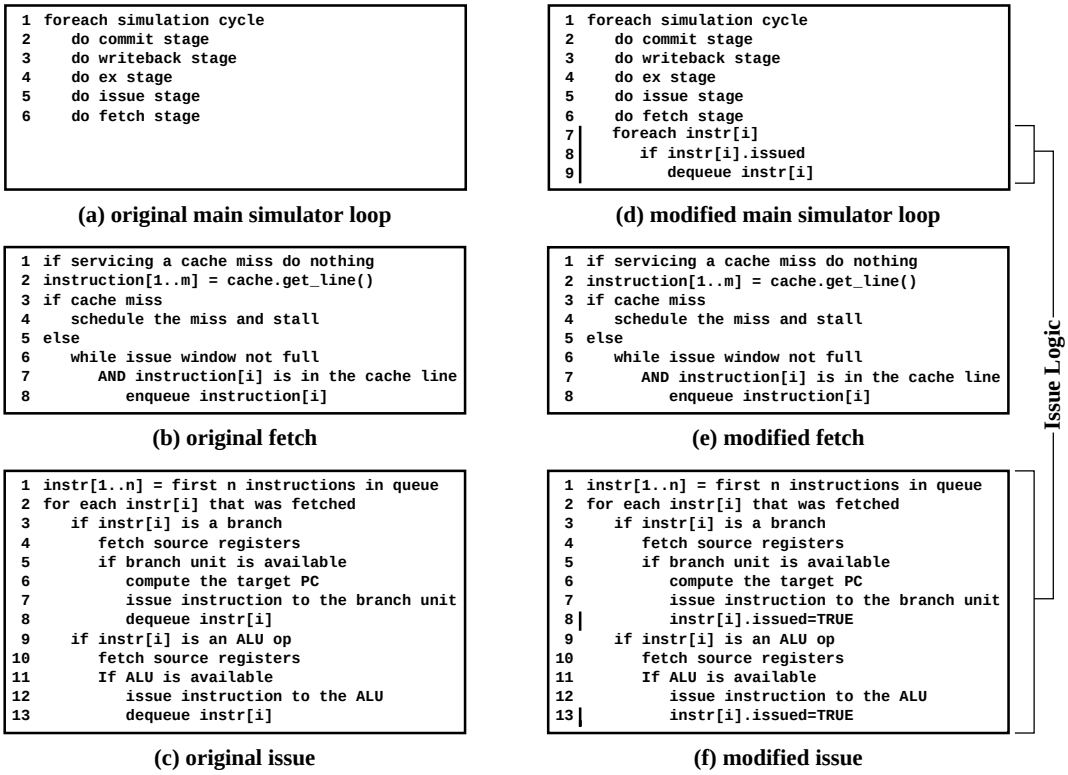


Figure 1: Sequential simulator code.

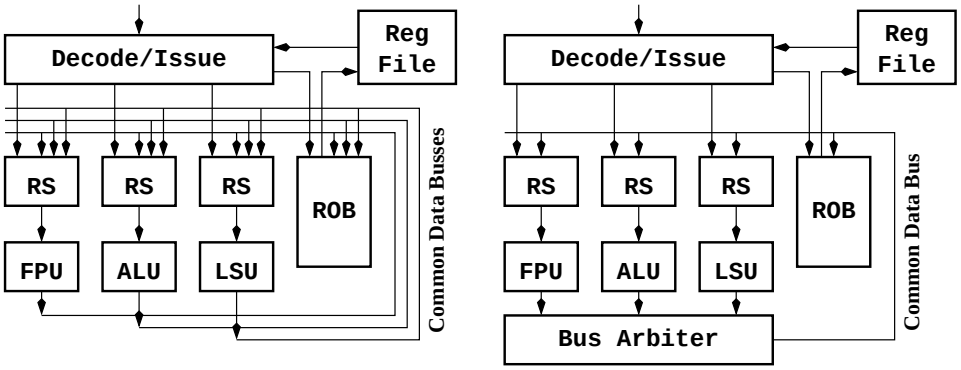
ecution of the pipeline stages (lines 7-9 in Figure 1d). This partitioning is necessary to ensure execution occurs in the proper order. Notice also that additional simulator state is needed to manage the issued status for each issue window slot. Just as changes to the way the instruction window was used required repartitioning of the issue logic, this new state, if used by other components, may also force repartitioning. While this small example may not seem overwhelming, this kind of partitioning is present throughout the code for a sequential simulator. Introducing parameters to control this type of behavior can quickly cause the code to become even more confusing and unmanageable.

The above argument and example illustrate the difficulties faced when mapping designs of concurrent, structurally composed hardware to sequential, functionally composed software. Since the component's communication interface is implicitly augmented by its functionality, systems built with functional composition fail to achieve the desired hardware-style encapsulation and separation of concerns. Thus, two similar microarchitectures may have radically different software implementations. Fixing misinterpretations of or modeling changes in the hardware description requires the architects to repartition components, thus remapping the hardware design to a software implementation. This *mapping problem* not only limits component reusability, but, as the next example will illustrate, it is also responsible for inaccuracies in architecture modeling.

Consider a machine which uses Tomasulo's dynamic scheduling algorithm and a simulator which is written in the same style as in the previous example. Figure 2 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. One need only modify the loop termination condition to cause the loop to exit if the writeback bandwidth has been exceeded. Figure 2d shows this modified code. Careful inspection, however, reveals that this one line code modification has had unexpected results. The communication of functional units to writeback buses is determined by the execution of the while loop (line 1 in Figure 2d). Thus, as shown in Figure 2c, run-time state has introduced a new hardware block, a bus arbiter, that is not *explicitly* modeled by the code. This one line code change misuses the execution semantics of sequential simulators to imply a bus arbiter.

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



(a) Original machine

(c) Limited write−back machine

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

(b) Pseudo−code for original machine

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

(d) Pseudo−code for limited write−back machine

Figure 2: The structure and pseudo-code for two Tomasulo-style machines.

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 introduce simulation errors.

This example demonstrates that manually mapping a concurrent, structural microarchitecture to a sequential program is extremely error prone. The resulting simulator is opaque with respect to the hardware it models, thus making it difficult to modify correctly. These modification errors may lead to inaccurate simulation. Others have noted there is a problem with the accuracy of simulators, but disagree on the source of the problems [2, 3, 4]. We contend that the mapping problem is the fundamental cause of inaccuracies in sequential simulators.

Solving the mapping problem clearly requires simulators to be designed using a methodology which, like hardware, is concurrent and structural. While highly disciplined object-oriented programming can allow full structural composition, this approach is still insufficient for the reasons described in the next section.

3. Concurrent and Structural Approaches

Several promising simulator specification systems have emerged that use structural composition to ease development and improve accuracy of simulators. However, despite this transition to structural modeling, most existing simulator construction techniques inherit some portion of the mapping problem, thus limiting their component reuse. This section demonstrates shortcomings in several of these replacements for hand-coded sequential simulators.

The Asim [1] performance modeling framework is a system targeted at solving the problems associated with mismodeled and approximated timing. To address these problems, Asim uses both concurrency and structural composition to model all intercycle communication. This structural composition of intercycle communication makes cycle-to-cycle communication explicit and reduces the system's dependence on global variables to communicate state.

Asim still models intracycle communication using sequential code that is composed functionally and thus suffers from the mapping problem. Communication changes which affect intracycle timing will require the same manual timing and repartitioning that plagued sequential simulators. Furthermore, a change that takes intercycle communication and turns it into intracycle communication (e.g. removing the only register on a communication path) may require rewriting some Asim modules since the two composition methodologies, structural and functional, are incongruent.

The EXPRESSION [5] architecture description lan-

guage (ADL) and its associated tools also attempt to alleviate the problems associated with hand-coded sequential simulators. Much like Asim, the EXPRESSION ADL employs both functional and structural composition. However, EXPRESSION allows both intercycle and intracycle communication to be modeled structurally. The functional composition is used to hierarchically build coarse grained components from finer grained ones, while the structural composition is used for communication between the coarse grained components.

From the available documentation, it appears that EXPRESSION limits the way in which components can be composed structurally. Components that are composed structurally cannot be invoked more than once per cycle. This means that *all* of a component's inputs must be available before it can be invoked. This restriction causes partitioning. If a single EXPRESSION component generates multiple outputs and one of these outputs is used to generate, in the same cycle, one of the component's inputs, then all the component's inputs will *never* be available simultaneously, thus forcing the component to be partitioned. This partitioning will make the simulator harder to understand and modify, just as it did for sequential simulators; EXPRESSION suffers from the mapping problem.

Other systems which enforce similar types of composition restrictions on their components will suffer the same limitations that Asim and EXPRESSION suffer. Furthermore, any system possessing multiple composition methodologies will, in general, have components which cannot be fully reusable. If the component is designed for one particular composition methodology, its use will be limited to situations where that particular composition methodology is appropriate.

SystemC [6] does not suffer from any of the problems mentioned above. It provides architects with a concurrent environment that supports structural composition for both intercycle and intracycle communication, and thus greatly simplifies the simulator construction process. An architect describes a microarchitecture by writing a collection of C++ classes that model the various components of the architecture, using SystemC data types and libraries to specify the communication interface. When using SystemC, architects need not worry about specifying the specific invocation order of components; instead they must only specify the timing of the system's communication. Since all composition can be specified structurally, a SystemC component's reusability is not limited by the composition methodology of the language. As a result, SystemC represents a significant advance over hand-coding sequential simulators.

Unfortunately, the system's generality is also its most serious limitation. SystemC uses a unified syntax for specifying module functionality and module connectivity. Thus it is possible to mix functionality description with structure description. This intermingling often obscures what the simulator is modeling. Additionally, SystemC makes no deliberate effort to define a universal mechanism to specify control flow between a pair of communicating components. Modules explicitly specify ports for control signals that they will use to determine whether or not to send data on the actual communication ports. Since these control interfaces may be designed in an ad-hoc fashion, components in SystemC are often unable to communicate with each other unless they were explicitly designed to communicate in the first place. However, due to its generality and concurrent semantics, SystemC is well positioned to be a simulation kernel for other higher-level architectural simulation systems.

From the above discussion, it is clear that a simulation system that avoids the mapping problem must be concurrent and support structural composition. Furthermore, the structural composition must not place any restrictions on when the composed modules will be invoked. In order to allow easy development of reusable modules, the communication interfaces must be carefully designed to allow two components, developed independently, to communicate even if such communication is unforeseen. Such communication must inherently support flow control so that architects may specify not only what data is to be sent, but when it is to be sent. These features will allow for the construction of widely applicable architectural component libraries since the system's communication semantics do not stand in the way of reuse. The Liberty Simulation Environment is a deliberate effort to *simultaneously* address all these issues, and it is the *first* simulation system to successfully achieve these goals.

4. The Liberty Simulation Environment

The Liberty Simulation Environment (LSE) is a simulator constructor that transforms a transparent machine description into an executable simulator. LSE's specification language and semantics are designed so that users of the system can design components that are interoperable even if they weren't designed to interoperate with each other. LSE leverages this capability to provide a user-extensible component library for microarchitecture simulation.

4.1. LSE Machine Description

LSE descriptions are built by instantiating predefined or user defined parameterizable components called *modules*. Each instance roughly corresponds to a hardware block in the complete microarchitecture. Some instances are used to model communication and dataflow between other instances, and others are used to model larger blocks such as caches and branch predictors.

Module instances communicate by sending values along connections made between the modules' *ports*. For example, LSE's branch predictor module has a port named `predicted_dir` on which the branch predictor outputs the

```
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 3: An LSE specification of the writeback stage of the machine shown in Figure 2, with an additional connection from the ALU to the LSU.

direction that it predicts a branch will take. It also has a port called `predict` to which other modules may send instruction addresses to request predictions. In general, each module specifies what inputs it requires and what outputs it generates by defining its port signature.

A user builds a machine description by instantiating modules, customizing their behavior by specifying parameters, and finally connecting them together by specifying the interconnection among the ports. Figure 3 shows a sample LSE description of the writeback stage of the machine shown in Figure 2c 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. Since connections in LSE are always point-to-point, we need two `tee` modules to handle the common data bus net and the ALU output net because these nets have fan-out. These tees are instantiated on line 6 and 7. Lines 17-30 connect the various ports on the modules to the appropriate other modules.

In addition to supporting conventional parameter types like integers and strings, LSE also supports *algorithmic parameters*. This feature is used in the above example to set the arbitration strategy that the `bus_arbiter` 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. This is shown on Lines 9-14.

The above example uses one additional feature related

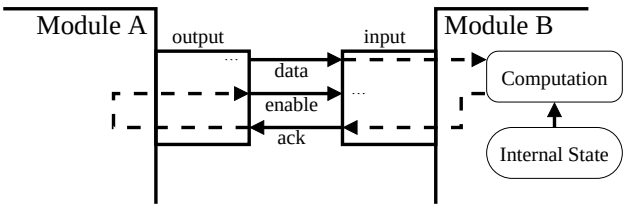


Figure 4: Connection with standard control flow semantics.

to ports and connections in LSE that allows modules to be more reusable. Notice that the description makes three connections to the arbiter module's `in` port. Each connection creates a separate instance of the port called a *port instance*. Within an architecture description, each instance of port can be treated independently by indexing the port like an array. Since the number of port instances is determined by the number of connections made to a port, modules are not restricted to having fixed numbers of inputs and outputs. Multiple port instances can be used, for example, by an arbiter module to automatically vary the number of inputs competing for a bus. Port instances can also be used to create fan-in/out. On the `tee` module, for example, port instances are used to fan-out an input to any number of multiple destinations. Without port instances, architects would be force to build a customized module each time an input or output width changed.

Notice that the LSE description directly corresponds to the structure of the hardware in terms of hardware blocks and interconnections. Since this description clearly exposes the structure of the machine, the LSE description can be visualized and manipulated in a graphical tool. This explicit structure also allows configurations to be analyzed so that the generated simulators may be optimized [7]. Furthermore, this description is easily constructed using only components from the LSE module library.

4.2. The LSE Module Contract

LSE is designed so that modules are interoperable, even when used in unanticipated ways. However, as we saw in Section 3, achieving this requires the definition of general flow control interfaces. LSE formalizes these interfaces into a contract that all components must obey. The LSE component communication contract draws inspiration from handshaking protocols commonly found on buses. Just as bus handshakes allow two components developed in isolation to interoperate, so too does the LSE handshake.

Each connection in an LSE description actually specifies three connections: a `DATA` and an `ENABLE` signal in the forward direction and an `ACK` signal in the reverse direction. Figure 4 shows these three wires. The receiving module should read its data from the `DATA` signal, use the data to update internal state if the `ENABLE` signal is high, and send an acknowledgment with an `ACK` signal if it is

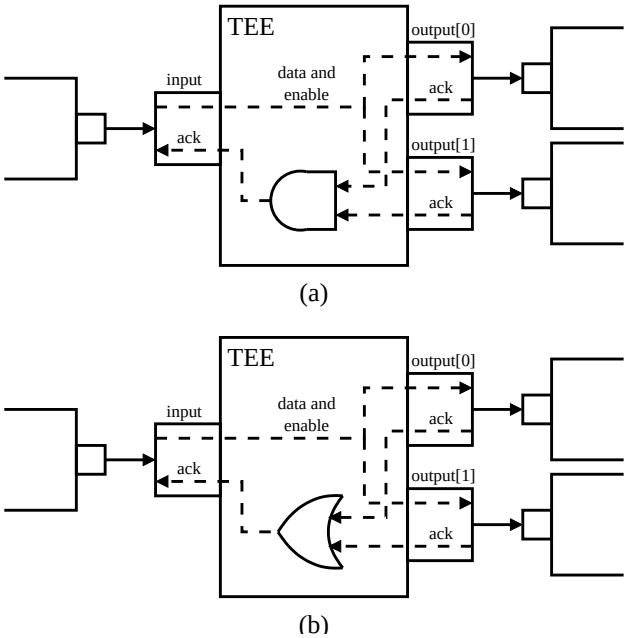
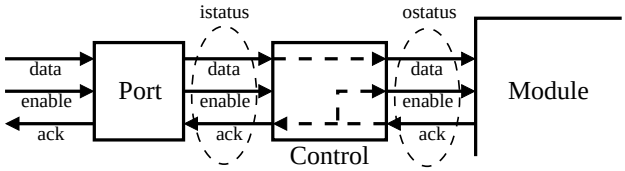


Figure 5: The `tee` module with default (a) and modified (b) control semantics.

able to accept the sent data.

In a typical communication, as is shown in Figure 4, the sending module (Module A in the figure) will send data on its output port. The receiving module (Module B in the figure) will determine whether or not it can accept the data and generate the corresponding ACK message. If the receiving module indicated that the data was accepted, then the sending module will enable the data for use by the receiver by raising the ENABLE signal. Otherwise the sending module will indicate that the data should not be used by lowering the ENABLE signal. Other styles of control (e.g. sending ACK before receiving DATA) can be used, and modules using different styles of control *can* interoperate as long as no intracycle *computational* loops are produced.

The 3-way handshake described above can also be used to coordinate communication between more than just a pair of modules. The `tee` module, for example, uses the handshake to coordinate the acknowledge signal of all the downstream recipients. The `tee` module forwards its incoming DATA and ENABLE signal to all its output ports, as shown in Figure 5a. 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 see an affirmative ACK signal if *all* modules down stream of the `tee` can accept the data. By modifying a parameter, we can also specify alternate behavior for the `tee` module. This behavior is shown in Figure 5b. In this case, the `tee` still fans out DATA and ENABLE signals to all connected modules, but it computes




```
LSU.store_operand =
        <<< return SIM_port_status_data(istatus) |
                   SIM_port_status_ack2enable(ostatus) |
                   SIM_port_status_ack(ostatus);
        >>>;
```

Figure 6: Control function overrides standard control.

the logical OR of the incoming ACK signals to generate the outgoing ACK signal. With these new semantics, the sending module will see an affirmative ACK if *any* module downstream can accept the data. Thus, using the 3-way handshake, we are able to implement useful control semantics for fan-out without difficulty.

The default semantics for the three signals associated with a port may not always meet the needs of a particular configuration. While it would be possible to remedy the situation by authoring a new module to change the control semantics, this would be tedious and dramatically limit reuse. To allow modification of control in the configuration, each port in LSE defines a *control point*. In the configuration, these control points can optionally be filled with a *control function* that modifies the behavior of the signals on the corresponding ports. While this behavior could be replicated by inserting an additional module, control points and control functions are provided for convenience.

To understand how control functions can be used in practice, recall the example from Figure 3. Assume the architects 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 6 fills the control point on the LSU's `store_operand` port. One may think of the control function as a filter situated between a module instance and the instance's 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). This particular control function passes the incoming DATA and ACK signals straight through without modification by using the SIM_port_status_data and the SIM_port_status_ack API calls, thus preserving the signals' original behavior. It moves the incoming ACK signal to the outgoing ENABLE wire by using the SIM_port_status_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.

The second obstacle to component reuse in existing systems was the limitation that modules be invoked only once per cycle. To avoid this problem, an LSE module must be invokable multiple times per cycle. This ensures that unnecessary composition constraints on module I/O will not prevent reuse by forcing users to partition modules.

When modules obey this module contract, they can interoperate with other LSE modules. Through good use of parameters, both algorithmic and non-algorithmic, a useful component library can be built.

4.3. LSE Component Library

In addition to providing users with a powerful tool with which to build microarchitectural simulators, LSE also provides a rich library of *flexible* predefined modules. These modules can be hierarchically combined to build more complex components or used directly in a microarchitecture description. This section will briefly introduce several of the modules in the library, explain their functionality, and highlight their configurability and reusability.

Many of the modules in the standard LSE library represent basic communication elements like tees and routers or basic memory elements like flops or queues. These modules form a core set of modules useful for controlling where signals go and controlling how long it takes for them to propagate.

The most primitive storage element in the default module library is the flop module. The flop module behaves like a register; it has a single input port and a single output port. Any data that is received on its input port is stored in the flop, provided the flop is not already storing another piece of data. The flop empties its contents when another module accepts the value by asserting the ACK signal on the output port.

Two additional memory elements provided by the library are a FIFO pipeline and a fixed size (configurable via a parameter) multiple input, multiple output (MIMO) queue. In addition to supporting multiple inputs and outputs, the MIMO queue supports an algorithmic parameter to prioritize the elements of the queue. This priority, which may be updated each cycle, is used to select which elements the queue will present as its output next cycle.

The library also contains tee, arbiter, demux, and filter modules. These modules can be used to route signals. Other than the tee, each of these modules possesses an algorithmic parameter to define its behavior. This parameter controls the arbitration logic in the arbiter module, the demultiplexing logic in the demux module, and the filtration logic in the filter module.

These modules, while simple, prove to be extremely versatile. When composed in various ways, the modules are actually able to build more complicated parts of a microarchitecture. Using queues and filters, we were able to model hardware blocks as complicated as a machine's wakeup logic. Thus, by using LSE's ability to create new modules by hierarchically composing existing modules, architects may easily extend the existing library.

In addition to these simple modules, the LSE module library contains more complex modules like branch predictors, caches, and instruction fetch units. The LSE branch predictor is actually a general two level predictor module. It can be used for branch prediction, but also for things like way prediction, predicate prediction, and cache line prediction. The number of predictors and the finite state machine controlling predictions are configurable via parameters. The hardware structure the module models is determined by instantiation parameters and its connectivity in the machine.

The decider module models the fetch logic of a machine, by combining incoming predictions and resolutions with the stored program counter to generate the next set of fetch addresses. It is common to connect the output of this module to the input of a cache controller in order to fully simulate the fetch stage of a pipelined machine.

In place of a cache module, the LSE library contains a collection of three modules: the cache_controller, the tag_comparator, and the cache_array. The cache_controller coordinates the cache_array and tag_comparator modules to handle external memory requests. It also connects this layer in the memory hierarchy to the adjacent levels. These modules can be composed hierarchically to build all combinations of physically tagged, virtually tagged, physically indexed, and virtually indexed caches. In fact, the modules are generic enough to be composed into a cache supporting way prediction by replacing the tag_comparator with the predictor module.

The Liberty system supports two methods of extending its module library. The first, as was already seen, is to hierarchically compose modules from existing ones. If the desired functionality cannot be achieved in this way, users can write new modules without much difficulty. LSE is even flexible enough to model systems not directly related to microprocessor cores. For example, LSE forms the basis of the Orion interconnection network simulator [8].

4.4. Iterative Refinement

With the LSE module contract, it is possible to create modules that are interoperable, and thus reusable. How-

ever, if library components were to require that every port be connected, users would be forced to specify an entire machine in LSE before any simulation was possible.

This iterative refinement is achieved in LSE by assigning default semantics to unconnected ports. In general, a module must behave in a reasonable fashion if some of its ports are left unconnected. For example, if the tag bits port on the `cache_controller` module is unconnected, the cache can assume that the tag bits come from the same address used for row lookup thus simulating a physically indexed, physically tagged or virtually indexed, virtually tagged cache. Alternatively, if the tag bits port is connected (e.g. to the output of the TLB), then the cache module will simulate a virtually indexed, physically tagged cache. Thus, during design of a system, if the details of the memory hierarchy are unimportant, default reasonable behavior can be obtained with minimal specification. As more detail and accuracy are necessary, additional connections can be made. In this way, complete architecture descriptions can be developed incrementally. Section 5 shows how this feature of LSE was used to build up a specification of an out-of-order machine model equivalent to `sim-outorder` from SimpleScalar [9].

4.5. Data Collection

Code for data collection is often intertwined with simulator code, obscuring the hardware being modeled. In order to facilitate data collection that is orthogonal to the simulator specification, LSE introduces two concepts: events and data collectors. Each time something "interesting" occurs in a module, the module will emit an event. The event notification will be tagged with the module that produced it, the time it was produced, and any associated data that the module wishes to emit. Data collectors, which are specified independently of the described architecture, get notified when events occur and aggregate the data contained in the event. Since certain collectors may be interested in only certain events, a mechanism is provided for collectors to filter the events they receive. These events and data collectors are similar to *aspects* in aspect-oriented programming [10].

5. Evaluation of LSE

The motivation to develop the Liberty Simulation Environment came from our own experience using sequential hand-coded simulators. Our experiences while trying to modify microarchitecture behavior and verify simulation results provided the impetus to search for a more appropriate technique to easily allow microarchitectural design-space exploration. Unfortunately, it is difficult to carry out objective experiments that demonstrate the quality of LSE as a simulation system since there exist no good metrics which can truly capture the differences in flexibility or ease of specification between two systems. While the previous sections of the paper have provided arguments explaining why the features of LSE promote clarity of specification, flexibility, and potential for reuse, this section will provide anecdotal evidence that illustrates these points. When possible, we introduce metrics to gauge the feature under test. Due to its popularity, availability, and position as one of the best freely available sequential simulators, we will periodically use SimpleScalar [9] as a baseline for comparison.

5.1. Building a Complete Model

One of the most important features of LSE is the ability to incrementally specify a microarchitecture starting from a simple model and refining the design until the full system is specified. In order to evaluate LSE's ability to support iterative refinement as described in Section4.4, we iteratively described an out-of-order machine model which, once finished, was a cycle accurate match of SimpleScalar's `sim-outorder`.

The starting point of the description and the first completely working simulator consisted of a single instance of the `decider` module. When this module is used by itself, the machine simulates an entire program in a single cycle since there are no connections or control to limit the fetch bandwidth.

The next step in the refinement process was to hook up a rudimentary pipeline to the fetch stage of our machine. This modification involved the instantiation of several more modules, primarily queues and tees. This new configuration allowed us to simulate a machine with coarse-grained pipeline timing. However, this configuration did not specify the logic to avoid data hazards. Thus, to make our machine more realistic, we further refined it by adding a module to handle register renaming and connecting the write-back ports of the pipeline to the machine's reorder buffer. It is worth noting that the machine's reorder buffer was modeled with a queue and a filter. Thus by only instantiating and connecting modules from the standard library, we were able to specify a complete out-of-order machine.

By connecting a branch predictor, an LSU, caches, and resolution logic to our existing model, we were able to transform the machine into a speculative out-of-order machine comparable to SimpleScalar's `sim-outorder`.

In order to verify the design, LSE's event and data gathering mechanism was used to produce a pipeline trace that could be visualized using tools provided by SimpleScalar. Once the model was completed, the LSE generated pipeline traces matched those produced by the default configuration of SimpleScalar exactly. Thus, using iterative refinement, we were able to construct a simulator which perfectly matched `sim-outorder`'s functionality and timing in only four weeks. It is important to note that during this four week period, much of the time was spent reading and analyzing SimpleScalar's sequential code to understand the structure of the machine it modeled.

5.2. Design Space Exploration

To evaluate the Liberty Simulation Environment's ability to perform design space exploration, we conducted several experiments to gauge how easy it was to modify an existing machine model. Several microarchitecture variants were configured in both the sequential and structural domain using SimpleScalar 3.0 and the architecture described in Section 5.1 as starting points. Modifications were made to both systems by an individual new to both Liberty *and* SimpleScalar. Table 1 summarizes the various configured architectural variants.

In order to evaluate LSE's flexibility, three metrics were used. The first metric, the diff/wc metric, measured how much a specification changed 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 modules affected in LSE-configuration changes 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 the particular modification took. The results of the experiment are summarized in Table 2. Note that the `splitda` modification could not be completed in SimpleScalar in under five hours and was abandoned.

Across the board, it took less time and fewer modifications to make these changes to the LSE specification as compared to the hand-coded sequential C simulator. Furthermore, the changes were more local in the LSE specification than they were in SimpleScalar. Despite the increased flexibility over SimpleScalar, the first three configurations shown in the chart require fairly significant distributed changes.

These three configurations correspond to different design choices in handling branch resolution. SimpleScalar recovers from a mispredicted branch when in the write-back stage. However, branches are marked as mispredicted in the dispatch stage, before this mispredict could have been detected in hardware. If there are two mispredicted branches in flight simultaneously and the second branch resolves before the first, SimpleScalar uses the fact that the first branch is marked mispredicted to suppress any mispredict recovery for the second branch. Thus SimpleScalar and our base configuration make an approximation rather than making a design decision. While modeling this approximation in Liberty was manageable, it added complexity that would not have been present had real hardware been modeled. Thus if we reevaluate the flexibility metrics, using the `mispred_imm` design as the baseline, we can obtain a more practical measurement of the changes required to explore different branch resolution designs. Table 3 shows the flexibility metrics using `mispred_imm` as the reference. As the table clearly indicates, modifying a configuration that is *actually* implementable in hardware leads to smaller, more local changes in specification.

The results from this experiment clearly demonstrate that using structural composition is advantageous when modeling microarchitectures. The sequential composition used in SimpleScalar proved to be a significant obstacle while trying to implement the described modifications. Since global variables were frequently used to communicate information, many changes which affected the connectivity of the machine required careful inspection *throughout* the code to find what variable references needed to be changed to new variable references. Furthermore, any change to the way state was updated required a thorough examination of the rest of the code to find all consumers of the data and ensure that each one was appropriately affected by the change.

The LSE configuration, however, proved to have a natural correspondence to the changes in the architecture. Simple reconfigurations of communication corresponded to small quick modifications to the description. Large changes took more time to model, but much of the time was spent actually thinking about how one would implement the change in hardware.

After changes were made to both systems, the resulting simulation pipetraces often differed. After careful examination of the changes the errors were found to be in the modifications to SimpleScalar in all cases. Any issues with the LSE configuration were obvious during configuration and promptly resolved since the configuration directly resembled the desired hardware. In fact, since changes were made to the LSE configuration and then to SimpleScalar, the insight gained during structural modification often eased the change in SimpleScalar. However, despite this advantage, the changes to the LSE configuration were more accurate and took less time.

5.3. Reuse

To measure the reusability of modules in LSE's module library, we compared the overlap of modules used in the SimpleScalar clone configuration and another machine configuration. This other configuration models an in-order machine that executes the IA-64 [11] instruction set architecture and was developed to support other research. The SimpleScalar clone configuration created instances of 16 distinct modules. The IA-64 in-order machine configuration created instances of 18 distinct modules. Between the two configurations, 11 modules were reused. The majority of modules that were specific to each configuration occurred in the memory subsystem since, for the SimpleScalar configuration, we used wrapped versions of SimpleScalar's cache rather than those found in the module library. Similarly, we wrapped SimpleScalar's branch

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

Table 1: Descriptions of the modeled microarchitectural variants.

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

Table 2: Time spent and code changed for modifications from the baseline configuration.

predictor, thus leading to additional configuration specific modules.

6. Related Work

A number of previous research efforts have attempted to simplify the architecture modeling process by creating domain-specific languages called Architecture (or Machine) Description Languages (ADLs) [12, 13, 14, 15]. The earliest ADLs only modeled instruction set architecture, and not microarchitecture, and are thus not important in this discussion. Later ADLs, such as LISA [13], RADL [14] and UPFAST [12], do model microarchitecture, and partially succeed in easing the task of obtaining simulators for simple and occasionally complex microarchitectures; the literature reports short model development times[12, 13]. However, they do not generally mitigate the transparency limitations of hand-coded simulators because they either require the user of the system to manage the order of the computation or limit concurrency to pipeline-like structures.

UPFAST [12] provides a way to partition the computation in the microarchitecture via a mechanism called TAPs. Each TAP is a sequence of code that is assigned a number by the user. The TAPs are then run in the order specified by this number. TAPs with the same number are run in an arbitrary order with respect to each other. TAPs give some concurrency, but each TAP may only be executed once, meaning that UPFAST suffers from the mapping problem for the reasons discussed in Section 3.

LISA [13] can be a reasonable system for specifying simple in-order pipelines, but, as the developers themselves state, LISA cannot model out-of-order processors and certain complex timing behavior that some microarchitectures contain. Furthermore, execution behavior in LISA is specified on an instruction by instruction basis and this description is intimately tied to the pipeline description. Thus LISA descriptions will see little reuse if the pipeline timing changes drastically.

RADL [14], though similar to LISA, can model more complex pipelines since RADL makes control signals explicit. Unfortunately, all timing is still tied to the main pipeline, making hardware that is independent of the main pipeline difficult to specify and drastically limiting reuse if the pipeline timing changes. As a result of these shortcomings, it is unclear whether RADL can model out-of-order processors. Even if it could, reuse, modularity, and ease of specification would then become primary concerns.

LSE does not have the shortcomings of the described ADLs. LSE has a concurrent specification language and thus makes it easy to specify components whose timing is only loosely coupled to the execution pipeline. Furthermore, since all modules in LSE obey a fixed contract, these concurrent components can be reused despite drastic changes in pipeline timing.

LSE is not the first system to realize that concurrency is an important programming and system specification feature. Among efforts not covered in Section 3, Ptolemy stands out as a popular tool [16]. However, Ptolemy is a general framework for exploring concurrency and does not eclipse any contributions made by LSE. LSE and Ptolemy are not directly comparable since Ptolemy acts as a general purpose concurrent programming system rather than a system for modeling microarchitecture.

Finally, a seemingly related project, MILAN [17], is, in fact, orthogonal to LSE. MILAN focuses on "vertical" integration of models, leaving "horizontal" integration as an open problem. LSE is a simulator constructor that addresses "horizontal" integration.

7. Conclusion

By solving the mapping problem, the Liberty Simulation Environment provides microarchitects with a means to radically improve their simulator design methodology. By

	LSE		SimpleScalar	
Config	diff/wc	Modules Affected	diff/wc	Functions Affected
mispred_old	19	1	53	4
mispred_com	31	2	334	5

Table 3: Code changed for modifications from the `mispred_imm` configuration.

adopting LSE, architects can have confidence in modeling results and explore ever more exotic designs through unrestricted component reuse. By moving to a methodology that makes modeling exotic designs easy and accurate, architects can avoid simply exploring incremental improvements to microarchitecture.

Additionally, by adopting a design-neutral, unrestricted open-source simulation framework, such as LSE, the community can improve the quality of best-known techniques. Transparency of specification and interoperability of components allow researchers to easily collaborate, exchange ideas, understand novel techniques, and evaluate the work of others in a variety of contexts, facilitating independent verification of research.

Acknowledgments

We thank Sharad Malik, John Sias, and Kees Vissers for their suggestions during the development of LSE and this paper. We also thank the anonymous reviewers for their insightful comments. This work was supported by National Science Foundation grants CCR-0082630 and CCR-0133712, a grant from the DARPA/MARCO Gigascale Silicon Research Center, and donations from Intel.

References

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

[2] R. Desikan, D. Burger, and S. W. Keckler, "Measuring experimental error in microprocessor simulation," *Proceedinngs of the 28th International Symposium on Computer Architecture*, July 2001.

[3] 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.

[4] 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)*, pp. 49–58, November 2000.

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

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

[7] D. A. Penry and D. I. August, "Optimizations for a simulator construction system supporting reusable components," Tech. Rep. Liberty-02-03, Liberty Research Group, Princeton University, September 2002.

[8] H.-S. Wang, X.-P. Zhu, L.-S. Peh, and S. Malik, "Orion: A power-performance simulator for interconnection networks," in *Proceedings of 35th Annual International Symposium on Microarchitecture*, November 2002.

[9] D. Burger and T. M. Austin, "The SimpleScalar tool set version 2.0," Tech. Rep. 97-1342, Department of Computer Science, University of Wisconsin-Madison, June 1997.

[10] 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*, pp. 220–242, 1997.

[11] Intel Corporation, *IA-64 Application Developer's Architecture Guide*, May 1999.

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

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

[14] C. Siska, "A processor description language supporting retargetable multi-pipeline dsp program development tools," in *Proc. of the 11th International Symposium on System Synthesis (ISSS)*, Dec. 1998.

[15] 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.

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

[17] A. Agrawal, A. Bakshi, J. Davis, *et al.*, "MILAN: A model based integrated simulation framework for design of embedded systems," in *Proceedings of the Workshop on Languages, Compilers, and Tools for Embedded Systems*, (Snowbird, Utah), June 2001.