Optimizations for a Simulator Construction System Supporting Reusable Components

David A. Penry
Department of Computer Science
Princeton University
dpenry@cs.princeton.edu

David I. August
Department of Computer Science
Princeton University
august@cs.princeton.edu

ABSTRACT

Exploring a large portion of the microprocessor design space requires the rapid development of efficient simulators. While some systems support rapid model development through the structural composition of reusable concurrent components, the Liberty Simulation Environment (LSE) provides additional reuse-enhancing features. This paper evaluates the cost of these features and presents optimizations to reduce their impact. With these optimizations, an LSE model using *reusable* components outperforms a SystemC model using *custom* components by 6%.

Categories and Subject Descriptors

I.6.8 [**Simulation and Modeling**]: Types of Simulation—*Algorithms*; B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids—*Algorithms*

General Terms

Algorithms, Measurement, Performance

Keywords

Liberty Simulation Environment, synchronous-reactive

1. INTRODUCTION

Microprocessor architects use processor simulation models to predict important design characteristics such as performance and power. Ideally, they would like to generate and use a large number of models to explore significant portions of the microarchitectural design space. This can only be possible if development time for the models is short and simulation speed is high enough to allow simulation of significant samples of code.

High model reuse is essential to obtaining short development times. Writing new code to model each point in the design space will not be acceptable. Several modeling systems, such as EXPRESSION[8], Asim[5], HASE[3], and Ptolemy[7] have attempted to provide reuse by providing libraries of components. Hardware or system description languages such as VHDL, Verilog, or SystemC[9] also provide facilities to create libraries of components. Models are built by instantiating components and connecting them together.

Components in such systems are written as if they execute concurrently. To provide semantics to the concurrent execution, some *model of computation* must be implemented by the system. The most common such model of computation is the well-known Discrete-Event model, used by VHDL, Verilog, SystemC, and others. The choice of a specific model of computation has two significant effects:

- Some models of computation, such as those used by EXPRESSION and Asim, limit the amount of reuse possible by forcing manual partitioning of the components in some situations or by requiring code to be written to explicitly control when components execute.
- Different models of computation experience different amounts and kinds of simulation overhead. For example, a Discrete-Event model requires maintenance of an event queue while EXPRESSION's model of computation does not.

Characteristics of the components themselves can also affect reuse and simulation speed. Two characteristics necessary to achieve very high degrees of reuse[1] and having potential performance impact are:

- Components must conform to explicit, standard communication contracts so that they may be connected together freely. These contracts should separate timing from data computation and flow control from data flow. These standard contracts may increase the number of signals in the model relative to custom contracts and thus may slow simulation.
- Individual components must be medium-grained; they need not be primitive logic components such as AND-gates, but they should be smaller than a functional unit. For example, a monolithic cache unit is too coarse-grained for reuse; it does not allow modeling of differences in timing between different arrays or such things as insertion of virtual to physical translations at different points in the data path. It is better to compose the cache out of components such as controllers, arrays, and TLBs. Using medium-grain components

[1]A discussion of why these characteristics are necessary can be found in [10].

DAC 2003, June 2–6, 2003, Anaheim, California, USA.

instead of large-grain components implies that more components and more signals are required, and thus may slow simulation.

We call any reduction in simulation speed caused by the use of components with these characteristics the *reuse penalty*.

While all of the aforementioned systems improve reuse compared to previous approaches, all of these systems fail to achieve the highest degrees of reuse because of difficulties with their models of computation or lack of standard commmunication contracts. SystemC is perhaps the best of these systems; its Discrete-Event model of communication does not prevent reuse, and its object-oriented nature as described in [2] might support the addition of contracts, but this has not yet been done.

The Liberty Simulation Environment (LSE) [10] is a simulator construction framework which has been designed to support higher degrees of reuse than previously possible. Explicit, standard communication contracts are supported by the framework. It includes a medium-grained component library designed for microprocessor development. This library includes such modules as branch resolvers, cache arrays, cache controllers, register renamers, queues, arbiters, and pipelines. This library can be extended, as was done to create the Orion interconnect power modeling libraries[11]. The model of computation is a Heterogeneous Synchronous Reactive model; this model provides performance advantages over the Discrete-Event model while not limiting reuse.

It is common wisdom that reuse comes at a price. In this paper we illustrate that with proper choice of the model of computation and additional optimization techniques it is possible to eliminate the reuse penalty when compared with systems using the popular Discrete-Event model of computation. We begin by describing LSE in Section 2. In Section 3 we compare equivalent models in LSE and SystemC to show the performance difference between the Heterogenous Synchronous Reactive model of computation and the Discrete-Event model of computation. We then measure the reuse penalty in LSE. In Section 4 we describe optimization techniques used in LSE to reduce the reuse penalty. In Section 5 we evaluate these optimization techniques and show that these optimizations allow an LSE model built out of *reusable* components to exceed the performance of a SystemC model built out of *custom* speed-optimized components.

2. LIBERTY SIMULATION ENVIRONMENT

LSE is a framework for construction of microarchitectural simulation models. A model is structural and concurrent, consisting of a netlist of connected instances of components called *modules*. These modules execute with concurrent semantics and communicate with each other through ports. This familiar style of modeling is similar to Verilog, VHDL, and SystemC, but there are a number of important differences. These differences can be classified as specification, infrastructure, and model of computation differences.

2.1 Specification differences

The specification differences between LSE and other systems are not the focus of this paper and space constraints prevent discussing them in detail. They exist principally to support higher reusability. For more information, see [10]. It is necessary, however, to explain two facets of the specification.

First, a Liberty model description is a two-level description. The top level of the description describes the instantiation and parameterization of modules and the connections between them. Modules are described separately as code templates in a target language, currently C augmented with API calls. These templates make up the module library. This two-level description makes it possible for the modules to be treated as black boxes (thus arbitrary C code can be in the modules) while still allowing analysis and use of the connectivity of the system.

Second, the communication contract between modules involves three signals per connection: a data signal in the forward direction, an enable signal in the forward direction, and an acknowledge signal in the backward direction. Modules use these signals to provide flow control behavior. For example, an arbiter module will look at its input data signals to determine which have priority and then assert the acknowledge signals for those ports which have won arbitration. This standard flow control contract greatly increases reuse of modules but creates more signals to be manipulated, which may decrease performance. The flow control behavior can be changed at each port through override code called *control functions* in the top-level description.

2.2 Infrastructure differences

A Liberty description is never directly interpreted like Verilog or VHDL sometimes are, nor is it directly compiled like SystemC. Instead, the description is used to generate the code of the simulator. The module template code in the library is "woven" with simulator API implementation code, control functions, and various pieces of the description to produce the code for each module instance. The template is not fully parsed as C; the generation process resembles macro expansion. This generation process enables the simulator optimizations described in Section 4.

2.3 Model of computation differences

The underlying model of computation of LSE is also different. Verilog, VHDL, and SystemC all use a Discrete-Event (DE) model. LSE uses the Heterogeneous Synchronous Reactive (HSR) model proposed by Edwards in [4]. This model is an extension of the Synchronous Reactive model (used in languages such as Esterel[1]) for situations where the precise input-to-output data flow of blocks of code are unknown (such as when the block is being treated as a black box).

In the HSR model of computation, signals can take on an *unresolved* value which is "less than" all other values. No other values are comparable with each other. All code blocks in the system which manipulate signals must compute monotonic functions on the signal values; if input values increase (relative to the partial order), output values must increase or remain the same. Changes to signal values are seen instantly by all receiving blocks. Computation within a time step takes place by setting all signal values to *unresolved* and invoking blocks until the signal values converge. The schedule of invocations can be either static (determined at system build time) or dynamic (as a signal changes values, its receivers are scheduled, as in Discrete-Event). The HSR model of computation does not require repartitioning of blocks as some other models do because the availability of the *unresolved* value permits blocks to partially evaluate their outputs.

This model of computation offers several potential performance benefits over the Discrete-Event model when creating and using a medium-grain library:

- Per-signal overhead is less as there is no double-buffering of non-state signal values and tests for changes to values are simpler.
- Blocks can use the *unresolved* value as an indicator of input availability and prevent premature computation of outputs that need that input when the input is not yet available. They can also use the absence of an *unresolved* value as an indicator that an output signal has already been computed, preventing redundant recomputation of that output.
- Modules can update internal state as soon as all inputs producing the new state value are available and all outputs depending upon the old state value are generated. This can be *extremely* useful when writing modules which implement Mealy state machines.
- The order of execution of blocks can be determined optimally and then encoded statically into generated scheduler code. This removes the need to dynamically schedule any code and may also reduce the number of executions of a block. It also removes the need for an event queue.

Note that a block and a module are not the same thing; a block is the smallest unit of scheduling granularity, but a module can contain more than one block. In particular, control functions are separate blocks.

3. EFFECTS UPON SIMULATION SPEED

In this section we measure the effects of the model of computation and reuse upon simulation speed.

3.1 Experimental methodology

We measure the effects upon simulation speed caused by the model of computation and reuse by measuring the running time of three simulation models of the same simple 4-way out-of-order processor implementing the Alpha ISA having an instruction window of 16 instructions, a reorder buffer of 32 instructions, perfect caches, and perfect fetch. These three models are:

1. A SystemC model using custom modules designed for speed for this particular configuration for each stage (fetch, decode/rename/commit, execute) plus a module to contain the other modules.
2. An LSE model using custom modules designed for speed for this particular configuration for each stage, as in the first model. No code for a "containing" module is required in LSE.
3. An LSE model using modules from the reusable library.

SystemC is used for comparison because it uses a Discrete-Event model of computation and is gaining wide acceptance for architectural modeling. There are three ways in which the custom modules are more configuration-specific than the reusable modules:

Table 1: Model characteristics

Model	Instances	Signals	Non-edge Signals
Custom SystemC	4	71	32
Custom LSE	3	138	48
Reusable LSE	11	489	423

- Custom modules do as much work as possible at the clock edge rather than combinationally during the clock cycle. Generally, the only non-edge-sampled signals are flow control signals. Reusable modules often separate combinational logic and state elements into different modules. This leads to both data and flow control being non-edge-sampled. Edge-sampled logic is more efficient because it is executed at most once per cycle.
- Custom flow control contracts are made. For example, if information is always removed from a multiple-output buffer in FIFO order, only a count of how many items to remove needs to be used for flow control rather than individual remove signals.
- Custom modules do not have any support for options unneeded in this configuration.

Characteristics of the models are given in Table 1, including the number of signals which are not sampled solely at the clock edge.

All models were compiled using gcc 2.96 for x86 with flags `-g -O2`. Version 2.0.1 of SystemC was used; it was compiled with the same compiler using the default installation flags. Flags `-g -O3` were tried for the SystemC model, but the results for all experiments differed by less than 1% and are not reported.

The models were run on nine benchmarks chosen from the SPEC CPU2000 and MediaBench suites. The benchmarks were 181.mcf, 186.crafty, 197.parser, 253.perlbmk, 179.art, 183.equake, jpegdec, jpegenc, and g721enc. All these benchmarks were compiled for the Alpha ISA for OSF Unix with gcc 2.95.2 19991024 (release) using flags `-static -O2`. Runs took place on a dual-processor 2.4 GHz Xeon system with 2 GB of physical memory running Redhat 7.3 (Linux 2.4.18-5smp). The benchmarks were run to the earliest of completion or 50,000,000 fetched instructions. Cycle counts for each benchmark were verified to be equal for each model.

All reported mean cycles/sec are the unweighted mean cycles/sec achieved over five runs of each benchmark. Speedups reported are obtained by computing the speedup of each benchmark (over five runs) and taking the mean of the speedups. Individual benchmark results are not reported due to space limitations, but the standard deviations (σ) of the speedups on a per-benchmark basis are very small, indicating that the speedup results are independent of the benchmarks.

All reported build times are the average time in seconds to build the model from source code to an executable over five runs of the build process.

3.2 Results

Table 2 shows the cycles/sec, speedups, and build time obtained by the three models. None of the optimizations described in the next section are applied to the LSE models and the schedule of block execution is completely dynamic.

Table 2: Performance of models

Model	Mean cycles/s	Speedup	σ of speedup	Build time(s)
Custom SystemC	53722	–	–	49.06
Custom LSE	155111	2.88 vs. SystemC	0.12	15.4
Reusable LSE	40649	0.26 vs. Custom LSE	0.01	33.9

Note that the speedup of the custom LSE model is with respect to the SystemC model and the speedup of the reusable LSE model is with respect to the custom LSE model.

The custom LSE model outperforms the equivalent custom SystemC model by 188%. The greater performance of LSE stems from the reduction in overhead allowed by the Heterogeneous Synchronous Reactive model of computation, use of the *unresolved* signal value to prevent premature computation, the static instantiation of modules (which leads to increased constant propagation and dead-code elimination in the compiler), and differences in run-time overhead and compiler quality between C and C++. While an exact breakdown of how much improvement comes from each factor is not available, we believe the model-of-computation related components (the first two) to be most significant because compiler optimizations and differences rarely show this magnitude of performance effects. Note again that this improvement occurs even with modules which have been carefully written for speed in this particular configuration and in spite of the larger number of signals which the custom LSE model contains.

The reusable LSE model shows a large reuse penalty of 74% performance loss when compared with the custom LSE model. This stems principally from the increase in the number of signals that must be handled in the reusable model, which in turn is due to the standard flow control contracts. Note that the ratio of performance is approximately equal to the reciprocal of the ratio of number of signals. The large number of non-edge-sampled signals in the reusable LSE model does not play a significant role because the modules use the presence or absence of *unresolved* signal values to avoid premature or redundant signal value calculations when a module is executed more than once.

4. OPTIMIZATIONS

LSE introduces improvements upon the scheduling algorithm provided by Edwards[4]. It also makes optimizations to the generated code in order to mitigate the reuse penalty. These scheduling improvements and optimizations are described in this section.

Previous work in compiling and optimizing Synchronous Reactive languages has assumed that all the code is visible and parsed by the compiler, which is not true of a Heterogeneous Synchronous Reactive system. An overview of previous Synchronous Reactive work can be found in [6].

4.1 Dynamic sub-schedule embedding

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

Edwards' basic static scheduling algorithm is:

1. Build a directed graph where each signal is a node and an edge (u, v) implies that signal u's value is needed to compute signal v. This graph will *not* be acyclic in the presence of limited dependency information because the flow control signals form cycles.

2. Break the graph into strongly-connected components (SCCs). Topologically order the SCCs.

3. Partition each SCC of more than one signal into two sub-graphs called the head and the tail. Pick an arbitrary schedule for the head. Schedule the tail recursively. Add to the schedule a loop containing the tail's schedule followed by the head's schedule; this loop executes as many times as there are signals in the head. Follow the loop with the tail's schedule again.

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

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

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

4.2 Dependency information enhancement

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

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

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

4.3 Block coalescing

The static schedule indicates the order in which signals should be generated, but blocks generally can produce more than one output signal per invocation. Edwards' algorithm includes a post-processing step to merge or coalesce invocations of blocks and thus reduce the number of invocations. The algorithm moves signal generations to the earliest point in the schedule at which the same block must be invoked where the move does not violate dependency constraints.

LSE uses a slightly different algorithm. First, LSE unrolls all the loops in the schedule before performing coalescing; this removes special cases that Edwards describes and makes it easier to merge with the last invocation of a block in a tail. Second, LSE is able to move multiple signals when it coalesces. One deficiency in Edwards' algorithm is that if a signal has no earlier block invocations to merge with, it prevents anything that depends upon it from moving before it. We maintain a list of signals to move and add a signal to the list when it is preventing a later signal from moving; this makes it possible to re-topologically sort the schedule as needed and increases the number of merging opportunities.

4.4 Code specialization

As noted before, module instance code is generated from templates. This provides the ability to specialize the code generated by creating different implementations of the port API for each port of each module. The API implementations are specialized for:

- Constant control signals
- Static vs. dynamic scheduling of receiving code blocks
- Diversity of receiving blocks across port connections: if all connections to a port are to the same block, code to update variables and do scheduling (if any) is simpler.

Table 3: Performance of reusable models

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

5. EVALUATION OF OPTIMIZATIONS

In this section, the effectiveness of LSE's optimizations is evaluated by measuring the running time of simulators generated with and without different optimizations enabled. The LSE model using custom modules and the LSE model using reusable modules are used for these evaluations. The experimental setup and reporting is the same as in Section 3; the only difference is that different combinations of optimizations are tried.

Note that no meaningful comparison can be made with Edwards' original scheduling algorithm because of its exponential runtime. An attempt to schedule the custom LSE model using Edwards' original algorithm did not terminate after 18 hours of runtime. Such a large runtime would be impractical for design exploration. As a consequence, all static scheduling in these experiments is done using LSE's dynamic sub-schedule embedding. However, we do compare the different block coalescing algorithms.

5.1 Static scheduling and dependency information enhancement

Table 3 shows the performance of the *reusable* LSE model when information enhancement and scheduling are varied. The enhanced block coalescing algorithm is used. Results where the framework embedded a dynamic schedule in a static schedule are marked with a "*". The maximum build time was 36.21 seconds; this is about 7% greater than the build time without optimizations.

The role of information is seen to be very important; a 40% speedup ($\sigma = 0.03$) can be obtained by analyzing the control functions and knowing module input/output dependencies and using these for static scheduling. Even with dynamic scheduling there is still an 18% speedup ($\sigma = 0.02$) to be obtained from the control function analysis. The effects of analysis and module annotations are synergistic; only when both are present can a fully static schedule and the maximum speedup be obtained.

Control function analysis alone produces speedup for two reasons. First, when control functions are analyzed, any constants found can be used in code specialization. Second, analysis allows the scheduler to remove an extra invocation of all control functions which is required in dynamic scheduling to produce any constant outputs of the function.

Module annotations alone produce much less benefit; there are fewer code specialization opportunities and no extra invocations to remove.

Table 4: Performance of coalescing

Coalescing Algorithm	Mean cycles/second	Speedup vs. original	σ of speedup
None	20564	0.40	0.01
Original	55983	1.00	–
Enhanced	57046	1.02	0.01

When information enhancement and scheduling are varied for the *custom* LSE model, equivalent speedups are not observed. Individual results for optimizations are not given because of space limitations, but they are all very similar to each other: no combination of optimizations achieves even a 1% improvement. This is because the custom modules provide only port independence information (there are no module annotations) to the framework and there are no control functions to analyze; as a result, the schedule is always completely dynamic and few optimizations come into play. Two combinations of optimizations show a 1% average slowdown; in both cases the slowdown is due to a single outlier.

Finally, note that when all the information is used for static scheduling, the LSE model built from *reusable* components is 6% faster than the SystemC model built from *custom* components. The combination of these optimizations with the HSR model of computation has allowed the reuse penalty to be completely mitigated when compared with a conventional Discrete-Event model.

5.2 Block coalescing

Table 4 shows the effects of the block coalescing improvements for the reusable model with all information available and static scheduling. Skipping the coalescing step results in a 60% slowdown, so coalescing is very important. However, the improved coalescing algorithm gives only a marginal improvement over the original coalescing algorithm.

6. CONCLUSIONS

Architects wishing to explore the design space of microprocessors require models which are both reusable and sufficiently fast. A simulation framework supporting a library of medium-grain, structurally-composed, concurrent components with explicit communication contracts can meet the reuse requirements. The Liberty Simulation Environment is such a framework. However, the speed of simulation can be affected by both the model of computation and the reusability of the components.

We show that for a simple out-of-order processor a model built in the Liberty Simulation Environment, which uses a Heterogenous Synchronous Reactive model of computation, can outperform an equivalent SystemC model using the Discrete-Event model of computation by 188%. This performance advantage occurs principally because of characteristics of the HSR model of computation and despite both models using components specially written for speed in their respective models.

We also show that when using reusable rather than custom components for the processor model, LSE experiences a 74% reuse penalty, illustrating the potentially large cost of reuse.

By optimizing the generation of simulator code to further exploit the Heterogeneous Synchronous Reactive model of computation, a 40% speed improvement can be achieved relative to an unoptimized simulator for the processor model using reusable components. These optimizations significantly reduce the reuse penalty.

The optimizations and the low overhead of the Heterogeneous Synchronous Reactive model together enable the speed of an LSE model for this processor built using reusable components to exceed that of a SystemC model using custom components designed for speed by 6%.

7. ACKNOWLEDGEMENTS

We would like to thank Manish Vachharajani, Neil Vachharajani, and the anonymous reviewers for their insightful comments.

8. REFERENCES

[1] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[2] 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.

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

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

[5] 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.

[6] N. Halbwachs. Synchronous programming of reactive systems, a tutorial and commented bibliography. In *Tenth International Conference on Computer-Aided Verification*, June 1998.

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

[8] 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)*, pages 256–261, October 2001.

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

[10] 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*, November 2002.

[11] 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.