Automatic Generation of Compiled Simulations through Program Specialization

Wing Yee Au
Daniel Weise*
Scott Seligman†
Computer Systems Laboratory
Stanford University
Stanford, CA 94305-2140

FUSE MEMO 91-4
This paper appeared in the Proceedings of the 28th Design Automation Conference, IEEE, June, 1991

Abstract

Using a program specializer, we automatically generated high-performance digital simulation algorithms from a simple interpreter-based simulator. By making simple changes in the simulator and the specializer, we generated four types of compiled simulations: the PC-set algorithm, an improvement on the PC-set algorithm, and two compiled versions of the BACKSIM algorithm. An analysis of our experiments indicate that large improvements to the PC-set method are possible, and that compiled simulation based on the pure BACKSIM algorithm should not be further investigated as the overhead of the algorithm is larger than the time it saves by pruning gate evaluations.

1 Introduction

As the size and complexity of digital circuits grows, so does need for logic simulation. Recent research has focused on *compiled simulation* because it provides substantially better performance than interpretive methods. In compiled simulation, a program is generated from the circuit to be simulated. Running this program, called a *compiled simulator*, then simulates the circuit. Compiled simulation executes much more rapidly than interpreted simulation because the overhead of tranversing and interpreting the data structure that describes the circuit is done only once, when the compiled simulator is created.

The simplest compiled simulators are based on the Oblivious and Levelized (LCC) [1] algorithms. In the oblivious algorithm, the compiled simulator evaluates all circuit elements once. Given an input change, the program is rerun until no circuit elements change. This method requires the greatest number of gate evaluations per input change. Nonetheless, it is very simple to implement, and incorporates a unit delay timing model. In the LCC method, a gate is only evaluated once all its inputs have changed. Therefore, only one run of the compiled simulator is needed to compute the final state of the circuit. The levelized method has the advantage of only requiring each gate to be evaluated once, but has the disadvantage of incorporating no timing model.

Other simulation algorithms have been investigated for compiled simulation. For example, Tortle_c uses an event-driven algorithm [2] to minimize gate evaluations and model timing, and the PC-set method [4] is a mixture of the oblivious and levelized methods. Some simulation algorithms, such as the demand-driven algorithm in BACKSIM [3], had not, prior to this work, been used as the basis for compiled simulation. We discuss these other algorithms in greater detail in the following sections.

Building compiled simulators for these simulation algorithms has not been an easy task. For example, event-driven simulators are more naturally implemented in an interpretive manner. Code generation for compiled simulations are not as simple to implement. Furthermore, the compiled simulation code usually needs some external event management data structure, or it has to use assembly code for event management and thus has machine dependencies. COSMOS and SLS [5, 6] compile the circuit into code which has to be scheduled by a centralized scheduler. Tortle_c [2] avoids a scheduler by using threaded code. LECSIM [7] avoids a scheduler by resorting to assembly code to do the scheduling. The BACKSIM algorithm depends on recursion and other data structures which makes a compiled simulator based on it difficult to implement.

In this paper, we present a new, automatic, method of generating compiled simulators using a general purpose *partial evaluator.* A partial evaluator converts a high-level program into a low-level program specialized for a particular application. In its application to generate compiled simulation, an interpreted simulator, which we call the *input or source simulator*, can be specialized for a particular circuit description to produce code that simulates the circuit—exactly the compiled simulator that we want. The algorithm underlying the compiled simulator is determined by the input simulator, so compiled simulators using different or more efficient simulation algorithms are easily generated by modifying the input simulator.

In addition, partial evaluation eliminates many inefficiences in the input simulator itself, simplifying the process of generating compiled simulators. That is, the programmer need not start with a completely tuned and super-efficient simulator: many of the algorithm independent optimizations performed by an expert programmer are automatically per-

*Supported in part by NSF Contract No. MIP-8902764, and in part by Defense Advanced Research Projects Agency Contract No. N00014-87-K-0828.
†Supported in part by NSF Contract No. MIP-8902764.

formed by the partial evaluator. The speed of the input simulator will affect the speed of specialization, but not the speed of the specialized program itself. This effect can inflate speedup numbers: by starting with a slower input simulator, one can make the speedups appear larger. This is true of our experiments, we estimate that our speedups (which range from 50 to 500 fold) are inflated two or three fold. Of course, the real measure of performance is the time taken to fully the compute a circuit's final state resulting from some input change, not speedup.

In our experiments, we generated four different types of compiled simulators. One corresponded to the PC-set method [4], and one was a variation of the PC-set method that yielded a 7-fold improvement on the original, on average. Two were compiled versions of the BACKSIM algorithm [3]. These experiments indicated that the overhead of the pure BACKSIM algorithm makes it infeasible as the basis for compiled simulators, although a hybrid version of the BACKSIM algorithm is best. We also attempted to specialize a true event-based data dependent simulator. Unfortunately, this exposed a weakness in the partial evaluator: the compiled simulator that it produced was too large to be usable, unlike our other experiments, where the code size was proportional to the number of gate evaluations. The technology of partial evaluation must be extended before we attempt this experiment again.

Section 2 describes the terminology and background of partial evaluation. Section 3 presents the algorithm of the input simulator we used for specialization. With this input simulator, we generated code similar to that produced by the PC-set method (Section 4) and then improved upon it (Section 5). Two compiled versions of the BACKSIM algorithm are presented in Section 6. Performance results are given in section 7. Conclusions and future work appear in section 8.

2 Partial Evaluation and Simulation

Partial evaluation converts a high-level program into a low-level program suited for a particular problem by specializing it with respect to some known data, or the structure of the data, that the program will see at runtime. The fundamental idea is this: when a program is to be run many times where some of the input data remains constant, it is more efficient to generate a specialized program that incorporates the constant inputs, and then run the specialized program on the inputs that change. We call the input data that is known the *static data*, and the part that is provided at runtime to the specialized program the *dynamic data*. The manipulation of the static part of the data is completely performed at partial evaluation time, and does not appear in the specialized program. The specialized program consists only of operations on the dynamic data, and these are the only computations executed at runtime. In general, the higher the ratio of static data to dynamic data, the higher is the performance gain, since more computation will be done at partial evaluation time rather that at runtime.

To generate a compiled simulator, the partial evaluator is given the simulator and the known inputs. Suppose that the simulator has the following functionality:

simulator(circuit-structure, circuit-state, inputs) $\rightarrow$ *outputs.*

The `inputs` consist of the pair <*input-format, input-values*> where input format can be further subdivided into a signal identifier and an application time. The `circuit-state` is a vector of register states. The circuit-structure and input-format are the static data, and the input-values and circuit-state are the dynamic data. Thus, partial evaluating the simulator on the static data gives a program that acts on the dynamic data:

PE(simulator, circuit-structure, input-format) $\rightarrow$ *specialized-program*

specialized-program(circuit-state, input-values) $\rightarrow$ *outputs*

The specialized program we generate is exactly code for simulating the circuit, ie., the compiled simulator we are seeking.

The partial evaluator performs as much computation as possible as it operates. By this mechanism it convert data structures into code. Partial evaluation does not change the input program's algorithm. Therefore, the algorithm of the input simulator determines the algorithm of the compiled simulator.

The partial evaluator we use in this work is called FUSE, a fully automatic partial evaluator developed at Stanford [11]. FUSE is implemented in the Scheme programming language [8], a dialet of LISP, and operates on Scheme programs. Hence all the simulation code presented below is in Scheme.

3 The Input Simulator

A circuit is represented by a data structure consisting of node and gate objects. Nodes have values that characterize the circuit state, while gates have logical functions that are performed on the input nodes of the gates. Flip-flops are treated as special gates with internal states. Nodes are numbered from zero to the total number of nodes; the number both identifies the node and indexes into a state vector holding the values of the nodes. Gates are similarly numbered. Flip-flops have additional numbering which indexes into a flip-flop state vector. These implementation details are hidden from the user by a circuit description language, largely borrowed from [8, 9], which allows a hierarchical description of the circuit in terms of blocks. Blocks of arbitrary width are defined using recursion.

Our simulator (Figure 1) uses a 2-state unit-delay logic model. The circuit state is represented by a state vector, indexed by node-id. A temporary state vector is also needed to hold the changes to the node values for each time point. The simulator uses three sets to maintain the scheduling information. Current-set holds the nodes that were changed during the last time point. Gate-set holds the gates that need to be evaluated at this time point, which are the fanouts of the nodes in current-set. Next-set holds the nodes that change by evaluating the gates in gate-set.

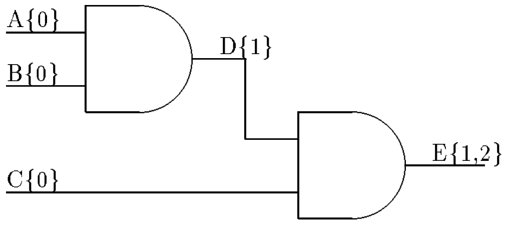
The inputs to the simulation is a list of <*signal-identifier, signal-value*> pairs for each application time, together with two vectors describing the initial state of the circuit (values at each node) and initial flip-flop states. Output requests may range from the value of a specific node at a specific time, to the complete history of the value of all nodes at all time points. A user interface is easily be built on top of these representations.

4 The PC-set Algorithm

The PC-set method [4] improves on the LCC method by including timing information via extra gate evaluations into a straightline compiled simulation. The simulations take longer, but they are more accurate. The PC-set method does a breadth-first prescan of the circuit to determine the

```
while inputs or current-set is not empty
 set inputs: for each input at this time point
               update state-vector and temp-vector
               add input node to current-set
 mark gate : for each node in current-set
               add fanout of node to gate-set
             clear current-set
 run gate  : for each gate in gate-set
               run gate using state-vector
               update temp-vector
               add output of gate to next-set
             clear gate-set
 update    : for each node in next-set
               copy temp-vector to state-vector
 swap current-set and next-set
 advance time
```

Figure 1: Algorithm of the input simulator.



```
char A_0, B_0, C_0, D_0, D_1, E_1, E_2;
  ...
D_0 = D_1;
D_1 = A_0 & B_0;
E_1 = D_0 & C_0;
E_2 = D_1 & C_0;
```

Figure 2: PC-sets of a simple circuit.

timepoints at which each gate may need evaluation, and produces a compiled simulator to evaluate the gates at those times. Our method of using a partial evaluator automatically produces the compiled simulator that the PC-set compiler produces.

We first describe the PC-set method in more detail. Given a digital circuit, the value of each node potentially changes only at certain times. The set of potential change times of node, which is called the PC-set, is easily calculated under a unit-delay timing model. Assuming the primary inputs of a circuit changes at time 0, these nodes have the PC-set of $\{0\}$. For the output of a certain gate, its PC-set is the union of the PC-sets of the inputs of the gate, with each element incremented by 1. Thus, the PC-set of a node is the length of all paths between that node and the primary inputs of the circuit. Figure 2, taken directly from [4], shows the PC-sets of the nodes of a simple circuit. From now on, we will refer to this circuit as the sample circuit.

Once the PC-sets of a circuit is known, it is relatively easy to generate code for the simulation. First, one variable is generated for each member of the PC-set of each node. In addition, for each node that uses an initial value, *ie.,* when the length of the shortest paths to the inputs of a gate are

```
(define (pcset_h state ffstate inputs)
  (let* ((A_0 inputs[1])
         (B_0 inputs[2])
         (C_0 inputs[3])
         (D_0 state[4]))
    (cons
      (list 0 (vector state[0] C_0 B_0 A_0 D_0))
      (let ((D_1 (logical-and2 A_0 B_0)))
        (list
          (list 1 (vector (logical-and2 C_0 D_0)
                          C_0 B_0 A_0 D_1))
          (list 2 (vector (logical-and2 C_0 D_1)
                          C_0 B_0 A_0 D_1)))))))
```

Figure 3: PC-set code for circuit in Figure 2 automatically produced by partial evaluation, returning history of all node values during simulation.

not the same, an additional variable is added. Therefore, for the sample circuit, we have the variables A_0, B_0, C_0, D_1, E_1, and E_2, and then we add D_0. Then simulation code is generated. A gate simulation is generated for each member of the PC-set of each non-input node, since these are the only times at which the nodes change their values. At the end of the simulation, the node-time variables yield a complete history of the circuit for an input vector. The code produced for the circuit is also shown in Figure 2. It is easily shown that the PC-set simulation is more efficient than an oblivious simulation. Running an oblivious simulation for 2 time units, the minimum time for the circuit to settle, needs 4 evaluations. Only 3 evaluations are needed in the PC-set simulation.

The computation behind the PC-set code is equivalent to our simulation algorithm (Figure 1). The PC-set code is just the straight-line code formed by unfolding the loop of the simulator, and specializing with respect to a particular circuit. Figure 3 shows the specialized code for simulating the sample circuit automatically generated by the partial evaluation. It is virtually equivalent to the code in Figure 2. In this experiment, the simulator (and therefore the compiled simulator) returned the value of each node during settling, that is, the complete history of the simulation itself. The only additional work this performs over the PC-set method is the creation of the histories; the gates evaluations are identical.

5 Improving on the PC-set Method: LPC

Like all simulation algorithms that compute from a circuit's inputs to its outputs, the PC-set method performs needless work; gates whose outputs do not effect the final output are computed. For example, the output of a combinational circuit may change many times before reaching steady state, but only its final value need be calculated; its history is irrelevant.

The difficulty in eliminating these unneeded computations is separating the needed computations from the unneeded ones. The history of a combinational circuit that clocks a register is relevant. The partial evaluator elegantly and automatically solves this problem through dead code elimination; the specialized program only includes those operations that may contribute to its output. No knowledge of circuits, gates, or timing is required. By changing the simulator to only report the final state of the circuit and the final values of the outputs, only those gate evaluations that may contribute to these values occur in the compiled simulator.

```
(define (pcset_o state ffstate inputs)
  (list
    (logical-and2 inputs[1]
                  (logical-and2 inputs[2]
                                inputs[3]))))
```

Figure 4: Code for circuit in Figure 2 automatically produced by partial evaluation, returning just the final value of the output

The complete PC-set code (of which there is a large amount, as mentioned in [4]) is generated only when the simulator returns the complete history of the circuit. Changing the simulator to only return the final values of the output nodes (a three line change) dramatically reduces the number of gate evaluations and the size of the produced code. An example is shown in Figure 4, which presents the compiled simulator for the sample circuit. It is much simpler than the previous compiled simulator. We have measured 4 fold to 14 fold fewer gate evalations using this improvement (Figure 10).

The style of code generated when the simulator only reports final values lies somewhere between the LCC method and the PC-set method, so we refer to it as the LPC-set method. Indeed, using this strategy, the compiled simulator for a combinational circuit is identical to the one produced via LCC. The difference occurs for sequential circuits, where timing matters: all transitions on the outputs of combinational circuits that generate clocks (ie, clock flipflops) are computed, whereas (in a well designed circuit) only the final values of combinational circuits that produce data are computed.

6 The BACKSIM Algorithm

There is yet another source of inefficiency in the improved PC-set method: gate evaluations occur that *may* contribute to the final result, but don't *actually* contribute to the final result due to data dependencies. For example, consider a 2-input or-gate whose first input has already been determined to be *true*. We may *cut off* the evaluation of its second input because this value is not needed. Gate evaluations would be saved if gates were only evaluated when provably needed. We say that a simulator has a *selective-trace* property if it uses data-dependencies to prune unneeded evaluations. (For example, event-driven simulators are selective-trace.)

BACKSIM, a selective-trace, variable delay simulator that only evaluates gates as they are needed, has been implemented [3]. Starting from the output request of the user, it works backward through the circuit, only evaluating those gates that are needed to determine the outputs. Because of fanout, it may request the value of a gate more than once. The gate (and all the gates it depends on) are only evaluated once regardless of the number of times its value is requested.

Because of cutoff, BACKSIM performs even fewer gate evaluations than the LPC-set method. Consider the sample circuit again. If the input C is known to be 1, the signal value at node D need not be requested. This early cutoff is useful when one input to a gate is inexpensive to evaluate, *e.g.,* a primary input, and the other input is expensive to compute.

BACKSIM's increased efficiency (w.r.t. gate evaluations) makes it a good candidate for specialization. However, there is unavoidable overhead in the method: remembering and testing whether a gate has been evaluated during the simulation. During an interpreted simulation, this overhead is relatively small, and BACKSIM has been shown to be more efficient than event-driven simulation. For compiled simulation, the relative overhead is much larger.

```
(define (demand state ffstate inputs)
  (list
    (if inputs[1]
        (if inputs[2]
            inputs[3]
            0)
        0)))
```

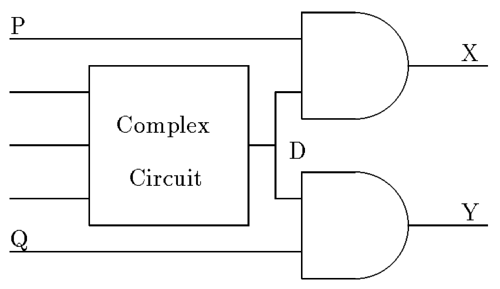
Figure 5: Demand-driven code produced by partial evaluation.

We made one very simple change to the experiment for the improved PC-set method to generate a compiled simulator based on the BACKSIM algorithm: the primitive boolean functions (such as `not`, `and`, etc.) were defined in terms of `if` expressions, instead of as primitive functions. When logical operations are expressed as functions all inputs must be known before the logical function is computed, there is no chance for cutoff. When logical operations are expressed as conditionals, cutoff will occur, as the inputs to a gate need now not be fully computed. For example, consider the rendering of an OR gate with inputs A and B: `if <expression computing A> then true else <expression computing B>`. The B expression is evaluated only when the A expression evaluates to false. The simulation program for the sample circuit when logical functions are expressed as conditionals (Figure 5) clearly shows the desired cutoff. Using conditionals rather than functions loses flexibility, and increases code size, limiting us to a 2-level signal model.

The hard part of a compiled simulator based on the BACKSIM algorithm is handling gates with fanout greater than one, as the expression computing the value of the gate would appear in the arms of many if-expressions, causing duplicate code and computation. To avoid this, the expression is textually isolated, and storage is reserved for both its value, and for remembering whether the value has been computed.

The partial evaluator automatically produces the code to this. When producing Scheme code, it uses Scheme's *delay* and *force* constructs. The delay operation is given an expression and returns a an object, called a *thunk* that can be *forced.* When the thunk is forced for the first time, the expression is evaluated and the value remembered. On each subsequent force, the remembered value is returned. Figure 6 shows the use of delay and force to avoid computing shared values unless absolutely needed. Node D is shared by the two and-gates. The value at D corresponds to the expression `shared-value`, which appear as a separate binding since it is shared by the last 2 lines in the code. Without the delay-force construct, `shared-value` will be evaluated every time the code is executed. This is obviously not necessary when P & Q are both 0. Early cutoff without unnecessary computation is achieved by the delay in the binding of `shared-value` and the `force` in the last 2 lines.

When we ran this experiment, we found that, when the compiled simulator was expressed in Scheme code, the costs of delays far outweighed their benefit. The compiled simulators ran (expressed in Scheme) 2 to 3 times slower than the LPC-set method. When expressed in C code, they ran roughly 10% slower (Figure 11) than the LPC-set method. To explain these results, we measured the number times each



```
(define (delay-force state ffstate inputs)
   (let (shared-value
          (delay <complicated expression>) )
     (list (if p (force shared-value) 0)
           (if q (force shared-value) 0))))
```

Figure 6: Circuit and code illustrating the use of *delay-force* to avoid computing shared values unless absolutely needed.

Circuit	0	1	2	3	4	5	6	7	8
C432	255	213	193	12	7	0	0	0	0
C499	25	164	182	121	43	8	6	1	0
C880	120	236	235	54	54	5	2	3	8
C1355	581	1243	534	60	9	4	3	0	0
C1908	149	572	420	74	29	3	5	0	0

Figure 7: Measuring number of times the value of each gate is requested during the BACKSIM algorithm. Accumulated sums for 50 random input vectors. Each column represents the number of gates requested the number of times specified in the column heading.

delayed expression was forced (Figure 7). We found that far fewer evaluations were saved than we had expected. Only for one circuit was the number of gates whose values were never needed as high as 1/3. The others were much less. Although we could lower the overhead of delayed expressions further, it could never be made low enough to make the pure BACKSIM strategy attractive for compiled simulation.

We then ran a second experiment that was a hybrid between the LPC-set and compiled BACKSIM algorithms: any delayed expression in the compiled BACKSIM method is simply precomputed. This tactic, achieved by setting a global switch in the partial evaluator, removes overhead at the expense of performing unneeded computations. This strategy, which we call *hybrid BACKSIM*, produced compiled simulators than ran about 5% faster than the those produced using the LPC-set strategy.

7 Experimental Results

We ran our compiled simulators on 5 of the circuits from ISCA85 [12], which are frequently used to benchmark logic simulators. All the circuits are combinational. Their characteristics are summarized in Figure 8. The simulation performance data are presented in Figure 9. The tests are performed on the input simulator (interpreted) in data-independence mode, in normal event-driven mode, and on specialized code produced with the PC-set algorithm and the demand-driven algorithm. In the LPC-set method only the final values of the circuit outputs are requested. Figures 3 and 4 constrast the difference between PC-set code and LPC-set code. Figure 10 compares the number of gate evaluations between the PC-set and LPC-set methods.

The compiled simulators are clearly much faster than the interpreted simulator they are based on. The measurements in the data-independent and LPC-set columns should be very consistent, since data-independence means that the same amount of computation is done on every run. Comparing columns 1 and 3 of Figure9, we are achieving 70 to 900 fold speedups. (Note, the additional speedup for the hybrid BACKSIM is larger here than in the C timings. We suspect this is due to compiler differences, and that the more conservative C timings are to be believed.)

Circuit	Inputs	Outputs	Levels	Occurrences
C432	36	7	18	202
C499	41	32	12	274
C880	60	26	25	468
C1355	41	32	25	618
C1908	33	25	41	937

Figure 8: Test case characteristics.

Circuit	Data-independent	Event-driven	LPC-set	hybrid BACKSIM
C432	2.5722	0.9200	0.0043	0.0033
C499	1.5298	0.9288	0.0225	0.0194
C880	5.0813	1.7558	0.0143	0.0118
C1355	7.2224	3.6437	0.0235	0.0177
C1908	17.3569	4.5343	0.0199	0.0149

Figure 9: Test results in seconds per vector. Columns 2 and 3 are interpreted simulations, columns 4 and 5 are compiled simulations. All tests are performed on an HP-9000-350 with 16-megabytes of memory. The simulation were run under version 7.1-alpha of the MIT Cscheme environment using the Liar compiler. Each circuit was run on 50 randomly generated vectors and then averaged. The time for reading inputs or printing outputs are not included.

8 Conclusions and Future Work

Partial evaluation proves to be very promising in generating compiled simulations. It provides high speedups by eliminating overhead, and optimizing the compiled simulator. In addition, we have great flexibility in controlling the algorithm and output requests of the compiled simulator by simply varying the input program and partial evaluation environment. The partial evaluator can thus generate efficient simulators that are tailored exactly to our needs, and all these are done automatically. This high flexibility comes

Circuit	Full PC-set	LPC-set	Ratio
C432	1713	226	7.58
C499	1032	238	4.37
C880	3147	396	7.95
C1355	5986	550	10.88
C1908	12691	871	14.57

Figure 10: Amount of work saved in requesting outputs only: number of gate evaluations needed.

Circuit	LPC-set	hybrid BACKSIM	BACKSIM
C432	2.4	2.2	2.5
C499	5.4	5.6	6.0
C880	5.7	4.9	X
C1355	8.3	7.8	9.0
C1908	7.7	7.4	7.9

Figure 11: Comparing the LPC-set, BACKSIM, and hybrid BACKSIM strategies. The compiled simulators were expressed in C code, timings are seconds per 5000 input vectors. The X indicates where the C compiler blew up.

about because the input simulator is written in a high-level language, yielding ease of construction, modification, and debugging. Partial evaluation provides the missing link between the two kinds of simulator implementations: the versatile interpreted method and the efficient compiled method.

Using the partial evaluator, we were able to implement and compare many different strategies for compiled simulations: PC-set, LPC-set, BACKSIM, and hybrid BACKSIM. We showed that LPC-set was superior to PC-set, that BACKSIM wasn't worth its overhead, and that the hybrid BACKSIM was best of all. Which one wins out is a function of compile time for a tuned compiler, the longer compile time required, the less useful the method. Because we are using a general program for generating the compiled simulations, we cannot estimate what compilation time would be when using a special purpose program for creating compiled simulators. (The partial evaluator we are using for our experiments runs very slowly, it's not fast enough to routinely generate compiled simulators.)

Future work along this line is to try to generate compiled simulators for other algorithms such as event-driven simulations. This involves some complication because we are then specializing a *data-dependent* program. The partial evaluator will have to deal with more side-effects, such as those on the state vector, at partial evaluation time. Although FUSE is capable of dealing with side-effects, and we did produce correct code for a data-dependent simulator, the code does not seem promising enough to deliver the high performance we are after.

Another point is the flexibility of the input structure that we allow in the partial evaluation. Only the input-values are allowed to vary in the residual code. The input structure must be completely static data. This arrangement of course produces the high performance we achieved in our experiments. But once again, we are trading efficiency for flexibility. To achieve this flexibility, in addition to producing less efficient code, the partial evaluator must handle more relaxed promises about the input data.

Finally, we could produce even higher performance compiled simulators by supplying actual input values at compilation time. For example, one could generate a compiled simulator that had the clocking strategy built into it. This would allow many more simplifications to occur at compile time and dramatically boost performance.

References

[1] M. Chiang and R. Palkovic, "LCC simulators speed development of synchronous hardware," Computer Design, Mar. 1, 1986, pp. 87-91.

[2] D. M. Lewis, "Hierarchical Compiled Event-Driven Logic Simulation," *Proceedings of ICCAD-89*, pp. 498-500.

[3] S. P. Smith, M. R. Mercer, B. Brock, "Demand Driven Simulation: BACKSIM," *Proceedings of the 24th Design Automation Conference*, 1987, pp. 181-87.

[4] P. M. Maurer, Z. Wang, "Techniques for Unit-delay Compiled Simulation," *Proceedings of the 27th Design Automation Conference*, 1990, pp. 480-84.

[5] R. E. Bryant, D. Beatty, K. Brace, K. Cho, T. Sheffler, "COSMOS: A Compiled Simulator for MOS Circuits," *Proceedings of the 24th Design Automation Conference*, 1987, pp. 9-16.

[6] Z. Barzilai, J. L. Carter, B. K. Rosen, J. D. Rutledge, "HSS–A High-Speed Simulator," *IEEE Trans. on Computer-Aided Design*, 6(4), July, 1987, pp. 601-16.

[7] Z. Wang, P. M. Maurer, "LECSIM: A Levelized Event Driven Compiled Logic Simulator," *Proceedings of the 27th Design Automation Conference*, 1990, pp. 491-96.

[8] H. Abelson, G. J. Sussman, *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge, MA, 1985.

[9] D. Weise, *Hierarchical Formal Multilevel Verification of Digital MOS/VLSI Circuits*, PhD. Thesis, Massachusetts Institute of Technology, Artificial Intelligence Laboratory Technical Report 978, Cambridge, MA, 1984.

[10] A. Berlin, D. Weise, "Compiling Scientific Code Using Partial Evaluation," *Computer,* IEEE, 23(12), pp. 25-37, 1990.

[11] D. Weise, "Graphs as an Intermediate Representation for Partial Evaluation," Stanford University, Computer System Laboratory Technical Report CSL-TR-90-421, Stanford, CA, 1990.

[12] Brglez, Franc, P. Pownall, R. Hum, "Accelerated ATPG and Fault Grading Via Testability Analysis," *Proceedings of the International Symposium on Circuit and Systems*, 1985.