

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 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 than 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)
→ outputs.
```

The inputs consist of the pair  $\langle \text{input-format}, \text{input-values} \rangle$  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) →
specialized-program
specialized-program(circuit-state, input-values) →
outputs
```

The specialized program we generate is exactly code for simulating the circuit, i.e., 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 dialect 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  $\langle \text{signal-identifier}, \text{signal-value} \rangle$  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