General Compiled Electrical Simulation

Daniel Weise
Stanford University
Computer Systems Laboratory
Center for Integrated Systems 207
Stanford, California 94305
(415) 725-3711

Scott Seligman
Stanford University
Computer Science Department
Margaret Jacks Hall
Stanford, California 94305
(415) 723-3088

1987

Abstract: This report describes the initial results of our research into *General Compiled Electrical Simulation*. We use advanced compiler techniques to speed up electrical level simulation such as the type performed by SPICE. Our system creates a *simulation program* by compiling together a simulator and the circuit to be simulated. Our approach results not only in substantial speedups, but also in simpler simulators. An added benefit is that simulation programs can be parallelized much more effectively than a simulator can be parallelized.

Submitted to the 1989 Design Automation Conference

This research was supported by a CIS Seed Grant and Darpa Contract N00014-87-K-0828.

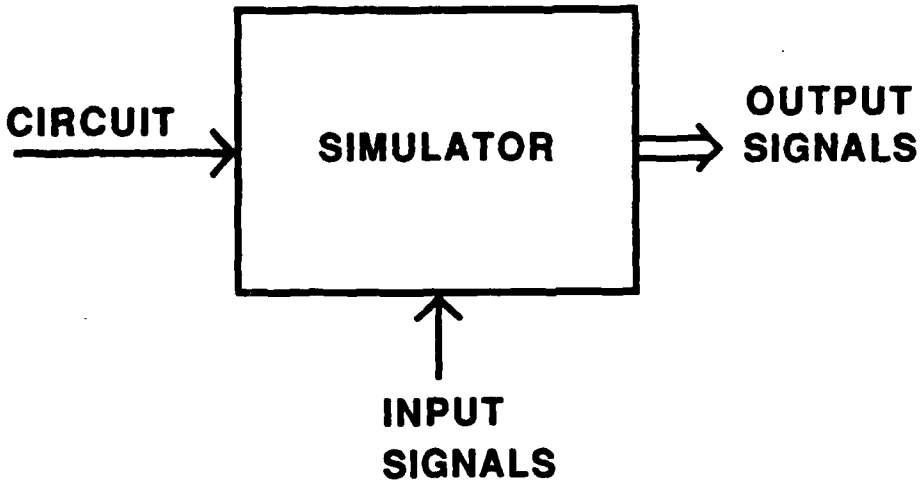
1 Introduction

Circuit simulation is vital to creating correctly functioning VLSI circuits. The advent of SPICE [Nagel] and other circuit simulators was a boon for circuit designers. Unfortunately, accurate electrical level simulation requires prohibitive amounts of computation for moderate or large circuits. Even when a given circuit takes an acceptable amount of time to simulate, it must be simulated many times before its design is finished, and the total simulation time can be very large. Most importantly, as circuit sizes increase, the size of circuits we wish to simulate grows as well. [White] recounts that at one major IC house more than 70% of an IBM 3090 is devoted to circuit simulation, and that at another house SPICE is run more than 10,000 times a month. Faster simulators will reduce the cost of designing functional silicon and improve the designs.

We are designing and building an interactive system for high speed electrical simulation of digital and analog circuits that is simple to use, programmable, much faster than any highly optimized simulator, capable of hierarchical and mixed mode simulation, and able to produce efficient code for parallel processors. We are employing four important ideas in the design of the system: general compiled simulation, embedded operation, object orientedness, and standard interfaces. Our ultimate goal is a software/hardware system costing around $15,000 that can sustain 100 MFLOPS on simulation problems.

This paper discusses the first of these ideas, *general compiled electrical simulation.* Standard circuit simulators accept a circuit and its input waveforms, and return the circuit's output waveforms. In general compiled electrical simulation (Figure 1), a simulator and a circuit are together compiled into a *simulation program* that, when run, has exactly the same behavior as running the simulator over the circuit, but runs many times faster. The simulation program's increased speed comes from compile-time unfolding of all constant data structures (such as the structure of the circuit itself) and from optimization of the unfolded code. Component values can be left symbolic during compilation so that recompilation isn't necessary when component values change. This feature increases the speedups for *What-If?* simulation and Monte Carlo analysis.

A simulation program consists of mostly straight line arithmetic code which uses few, if any, structured values. All opportunities for parallelism are explicit. A compiler can plan the spatial and temporal use of nearly every value, i.e., where the value is stored and when the value is computed. Therefore a simulation program will make very efficient use of heavily pipelined and parallel machines. A compiler will also be able to optimize register and data cache usage. Because the dynamic behavior of the simulation program is statically determined and all parallelism is explicit, economical special purpose hardware for executing the simulation program is feasible.



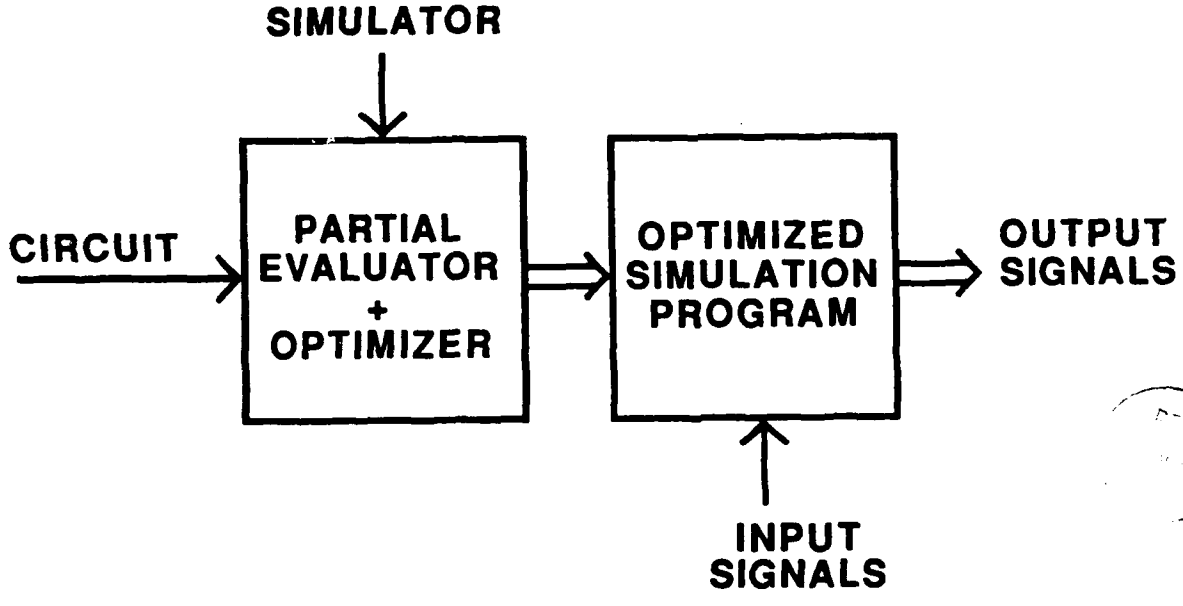


Figure 1: Normal Simulation and General Compiled Simulation. Normal simulation is depicted in the top figure. In normal simulation a simulator accepts a circuit and its input signals, and returns the circuit's output signals. In general compiled simulation, shown in the bottom figure, a *partial evaluator* accepts a simulator and a circuit, and produces a *simulation program* that, when applied to input signals, produces output signals.

A prototype of our system is implemented for linear circuits[1]. The simulation programs we produce run at least five times faster than [Spice3].

This paper has six sections. Background and related work are presented next, in Section 2. Section 3 describes simulation programs and the benefits of compiled simulation. Section 4 discusses general compiled simulation and presents our partial evaluator. Our results are given in Section 5. The last section presents a summary and outlines our future research.

2 Background and Related Work

The creation of simulation programs for faster simulation has been successfully implemented at the switch level, logic level, and behavioral level. Some work has begun at the analog level.

[Bryant] has written COSMOS, a program for compiled switch-level simulation. Not counting compilation time, COSMOS runs about ten times faster than its cousin MOSSIM. The speedups come from processing structure only once and from optimizing the resulting simulation program (actually, a system of boolean equations). At the logic level [Barzilai] describes a program called HSS that compiles logic expressions into System/370 assembler code. HSS can simulate around 240 million gate-patterns per second, a fairly respectable speed. [Hansen] has implemented a compiled simulation system called *Terse* for behavioral/logic simulation. *Terse* employs a rich set of symbolic input types to provide as much compile-time optimization as possible. Hansen doesn't quote specific speedups, but claims that *Terse* is instrumental in daily design work at MIPS.

Some compiled simulation was performed in early versions of SPICE [Nagel], where some of the matrix LU decomposition routines were turned into assembly code for each circuit. Considering that model evaluation was the CPU intensive part of his system, Nagel should have investigated compiled simulation for model evaluation. Unfortunately, he didn't have the technology for doing so. Another program that produced assembly code for matrix operations was [ASTAP]. More recently [Vladimirescu] reported on a more modern system that also compiles away the control portion of matrix solution.

[Lewis] has proposed compiled simulation and special purpose hardware for electrical simulation. He intends to build special purpose hardware that can perform simulations 500 times faster than uniprocessor systems. Our research differs from his in three major respects. First, we are interested in automatic general methods for compiled simulation whereas Lewis writes his simulation compilers by hand. Second, we are interested in designing

[1] Results and timings for non-linear circuits will be ready by the time final versions of papers are due for the inclusion in the proceedings.

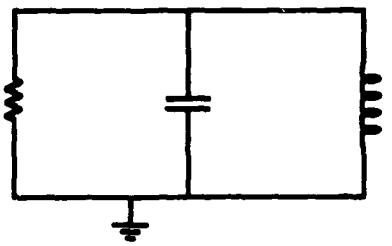


Figure 2: A simple RLC circuit

the cheapest, simplest hardware possible. Our goal is to sustain 100 MFLOPS for around $15,000. Third, we also want our compiler to produce efficient code for commercially available parallel processors. This paper addresses only the first difference.

3 Simulation Programs

Compiled simulation produces *simulation programs* that run much faster than an equivalent simulator simulating a circuit. We expect a tenfold speedup over a simply written simulator and a fivefold speedup over an optimized simulator. Two factors contribute to the speed of the simulation program: constant folding of the topology of the circuit, so that the topology is processed only at compile time, and optimizing the resulting program using techniques such as common subexpression elimination and dead code elimination. Constant folding the topology at compile time unfolds the loops that tranverse the structure of the circuit so that the circuit structure is embedded in the resulting program. This process yields virtually straight line code. For example, for a simple transient analysis simulator only two loops remain: the Newton-Raphson iteration for nonlinear analysis and the outer integration loop.

As an example of the power of compiled simulation we present a fragment of our transient analysis simulator (Figure 3) and the code produced for that fragment (Figure 4) when compiled against an RLC circuit (Figure 2). The simulator uses nodal analysis and trapezoidal integration. To generate the state at time $t + h$ from the state at time t, it creates a matrix M to be solved for node voltages at time $t + h$. From the node voltages at time $t + h$ and the state at time t it computes the branch currents at time $t + h$. The simulator computes matrix M by summing into it the current and conductance contributions of each component. The simulator is object oriented: each time-varying and reactive component carries its own function for computing its contribution to the matrix M. The methods must be retrieved and invoked at each time step.

Figure 4 depicts the simulation program that results from partially evaluating the func-

```
(define (next-state state)
  (let* ((matrix (+matrices
                  matrix
                  (create-integration-matrix circuit state h parameters)))
         (new-voltages (solve-matrix (trim-ground matrix)))
         (new-currents (compute-b-currents
                        circuit new-voltages state h parameters)))
    (create-new-state new-voltages new-currents (+ h (state-time state)))))

(define (create-integration-matrix circuit state h parameters)
  (let ((voltages (state-voltages state))
        (currents (state-currents state)))
    (let loop ((components (circuit-components circuit))
               (matrix (create-nxn+1-matrix (circuit-number-of-nodes circuit))))
      (if (null? components)
          matrix
          (loop (cdr components)
                ((2t-component-integration-method (car components))
                 matrix voltages currents h parameters))))))
```

Figure 3: Scheme Code Fragments for Transient Analysis. These two routines constitute the inner loop of transient analysis for linear circuits. The first function accepts a state at time t and returns the state at time $t + h$. It first calculates the node voltages by solving the matrix which is the sum of the "DC matrix" (the current and conductance contributions from resistors and time-invariant current sources) and the "integration matrix" (the current and conductance contributions from time varying and reactive elements). Once the voltages are computed the branch currents are computed. The function `create-integration-matrix` uses object oriented techniques to collect the conductance and current contributions of the reactive components to the admittance matrix. It retrieves the function for computing an element's contributions from the element and then invokes the function. We show these fragments to emphasize the amount of work the simulator must perform to compute the next state. Function retrieval and invocation occur for each component. A matrix is created and inverted for each time step. Figure 4 shows the power of compiled simulation by presenting the result of compiling this code for the simple RLC circuit of Figure 2. All intermediate structured values and control constructs completely vanish leaving only straightline arithmetic code.

```
(LAMBDA (STATE)
  (LET*
      ((TEMP1 (VREF STATE 1))
       (TEMP2 (VREF STATE 0))
       (TEMP3 (VREF TEMP1 1))
       (TEMP4 (VREF TEMP2 0))
       (TEMP5 (* TEMP4 .00005))
       (TEMP6 (+ TEMP5 TEMP3))
       (TEMP8 (VREF TEMP1 2))
       (TEMP9 (* TEMP4 .02))
       (TEMP10 (+ TEMP9 TEMP8))
       (TEMP12 (- TEMP10 TEMP6))
       (TEMP13 (VREF STATE 2))
       (TEMP14 (* TEMP12 49.6277915633))
       (TEMP15 (- TEMP14 TEMP4))
       (TEMP17 (* TEMP15 .02))
       (TEMP18 (- TEMP17 TEMP8))
       (TEMP19 (+ TEMP14 TEMP4))
       (TEMP21 (* TEMP19 .00005))
       (TEMP22 (+ TEMP21 TEMP3))
       (TEMP23 (* TEMP12 4.96277915633e-3))
       (TEMP24 (+ .1 TEMP13)))
    (VECTOR (VECTOR TEMP14) (VECTOR TEMP23 TEMP22 TEMP18) TEMP24)))
```

Figure 4: Compiled code for the transient analysis of the simple RLC circuit. This function accepts a state at time t and returns the state at time $t+h$. In this code h is 0.1 seconds. Not counting creating the next state, there are 20 instructions, of which 14 perform computation and 6 destructure the input state.
This code is optimal. Dead code elimination, constant folding, sign targeting, and arithmetic simplification are the major optimizations. For example, constant folding collapsed R, L, C, and H into constants such as .02 and 49.6278. Sign targeting eliminated TEMP7 and TEMP11. Arithmetic simplification eliminated TEMP16 and TEMP20.

tion `next-state` for the RLC circuit. The straight-lineness, compactness, and lack of structured values are the striking attributes of this code. No vestiges of the matrices produced or consumed during compilation, or of the control structures for doing so, or of the matrix inversion, or of the function retrievals and applications, appear in the final code. All address calculations vanish. There are many ramifications of the simplicity of the simulation program:

All uses can be planned. All values are explicit and can be explicitly planned. The compiler decides where in memory everything will reside. Heavily pipelined machines will be used very effectively. Also, state variables will be held in registers. Regular simulators cannot assign state variables to registers, which results in extra memory references and slower performance. (This drawback of normal simulators becomes worse and worse as processors get more and more registers.)

The program can be further optimized. Many opportunities for optimization arise because all values and their uses become explicit. Our system produces optimal code for the RLC circuit, an amazing result given our very inefficient prototype simulator. The most important optimizations are dead code elimination, sign targeting, arithmetic simplification, constant folding, and common subexpression elimination. It is virtually impossible for a human coding in assembler to create code as good as our system can create.

The program can be efficiently parallelized. As a corollary of being able to plan the use of all values, a compiler can also plan the efficient parallel use of the code for either tightly coupled or loosely coupled parallel systems. This ability is to be compared with the extremely hard task of producing good parallel code for a simulator itself, as we mentioned in the introduction. Because we parallelize the simulation program, not the simulator, we avoid many of the problems associated with attempting to parallelize simulators.

Component models are programmable with no overhead penalty. Today's simulators execute user-defined models less efficiently than they execute built-in models. The overhead comes from repeatedly looking up the model at simulation time. Compiled simulation performs the lookup once, at compile time, so that there is no cost difference between built-in and user-defined models.

Special purpose hardware. We can design extremely economical special purpose hardware for executing simulation programs. We anticipate that a machine that sustains

100 MFLOPs can be built for between $10,000 and $20,000. We believe we will keep five 20MFLOP ALUs busy with very little supporting hardware. Much of the hardware in existing "minisuper-computers" such as the Stellar and Ardent is support hardware. For example, in the Ardent, 30% of the gates are devoted to the scoreboard alone [Ardent]. In our proposed system we offload the function of such extra hardware into the compiler.

100 MFLOPS is a lot of power. If we pessimistically assume each timestep computation devotes 2000 floating point operations to each device, we can still comfortably simulate circuits which contain 50,000 devices.

4 General Compiled Simulation

In *general compiled simulation* we employ a program, called a *partial evaluator*, that accepts a simulator, a circuit, and a description of the data, and produces the simulation program. This is simpler and better than implementing a specific compiler for a specific simulator. General compiled simulation has many benefits:

Simulators become simpler to write. Because the circuit compiler does our optimizations for us, we need not waste our time trying to accelerate simulators. We will write coherent, simple, easily understood textbook style simulators. The importance of this should not be underestimated.

Component values can remain unspecified while compiling. When component values are unspecified during compilation the simulation program accepts the component values as parameters. A circuit isn't recompiled when these values change. This feature allows super-fast "*What-If?*" simulation. Also, Monte Carlo analysis and sensitivity analysis are speeded up dramatically. We see this super-amortization of the compilation time as a major strength of general compiled simulation.

Simulation programs can be recompiled for known component values. A Simulation program can be recompiled for given component values to constant fold those values into the program for even greater speedup. Therefore late binding of values need not have an effect on runtime speed. Note that once we have our partial evaluator, we get this added feature for free.

The Partial Evaluator

We feed the partial evaluator a function F, some real and some symbolic parameters, and we get out a new function G which accepts a parameter for each symbolic parameter to F. The function G returns a result as if we had called F on all its parameters at once.

The partial evaluator is an interpreter that handles symbolic values. Whenever the argument to a primitive function such as + is encountered, the partial evaluator adds an instruction to the program being created and returns a new symbolic value. Symbolic values carry information about the object they stand for. For example, a symbolic value can indicate it is a number, a string, or a vector or list of a certain length. The partial evaluator uses this information for several different tasks. For example, it performs compile-time type checking with this information. The partial evaluator also associates information with the symbolic values it creates. For example, when removing the head of a symbolic list that it knows contains five elements, it will create a symbolic list containing four elements. This methodology fully unrolls loops over lists whose lengths are known at compile time (such as a list containing the capacitors of a circuit).

For example, to create the compiled version of transient analysis upon circuit C with stepsize .1 seconds we type[2]

```
(define
  simulation-program
  (partially-evaluate transient-analysis c symbolic-value .1))
```

This call returns a simulation program that accepts component parameters and then performs a transient analysis.

The partial evaluator operates in three phases (Figure 5). The first phase builds a dataflow graph, the second phases optimizes the graph by applying graph transformations, the third phase allocates registers and produces code. For example, the first phase produces the dataflow graph shown in the top half of Figure 6, and the optimized version of this flowgraph appears in the bottom half. The code produced by the third pass was presented in Figure 4.

The first pass operates as described above, but symbolic values are augmented to be nodes of the dataflow graph. When a primitive such as + or * receives a symbolic value as an argument it creates a new node (symbolic value), sets up links in both directions between the incoming symbolic value and outgoing symbolic value, and then returns the new symbolic value.

[2]Of course, the system hides this ugliness from the user.

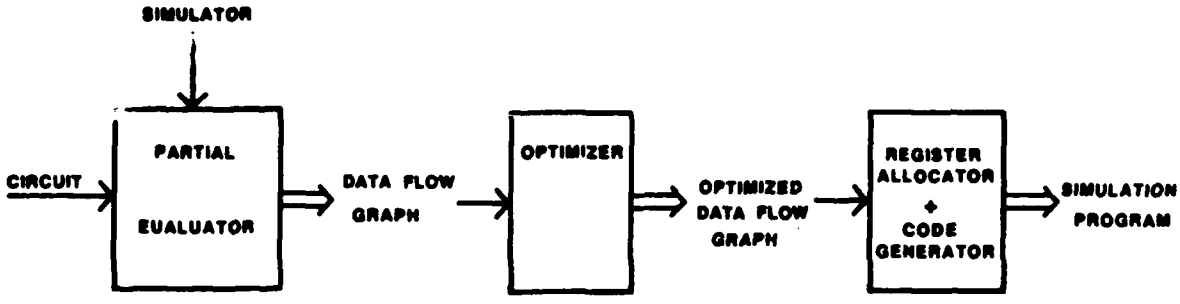


Figure 5: The partial evaluator. It has three phases. The first phase maps a simulator and a circuit into a dataflow graph. The second pass optimizes the dataflow graph. The third phase produces code. Fi gure 6 presents example outputs from the first two phases.

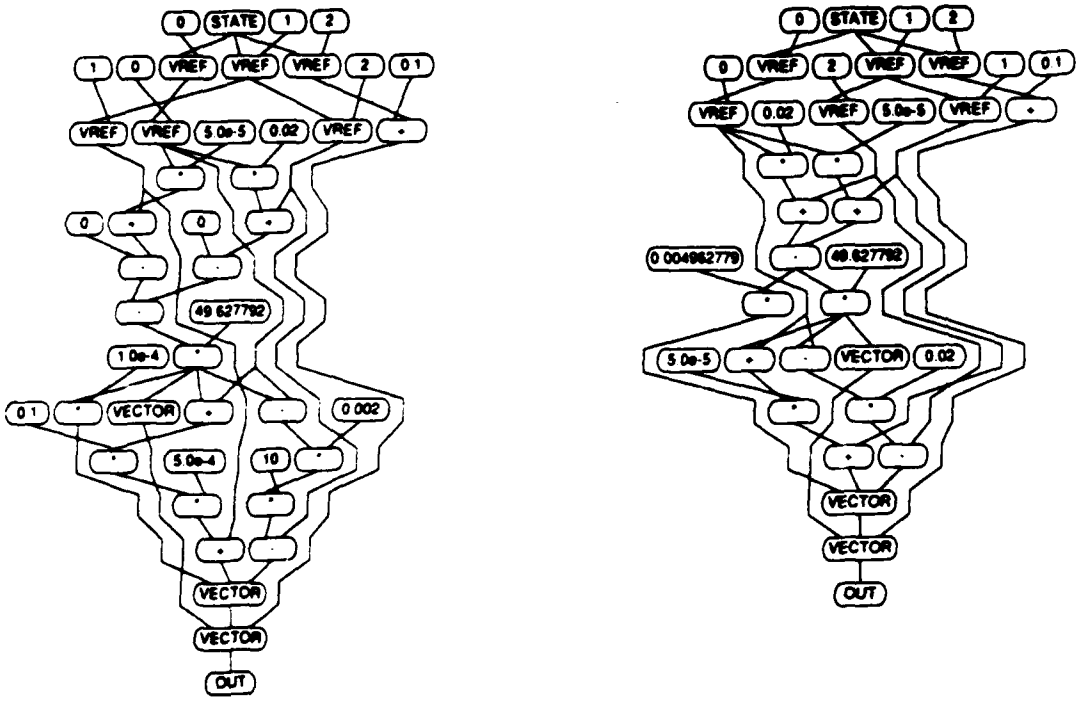


Figure 6: The dataflow graphs produced by the partial evaluator when compiling the RLC circuit. The graph on the left is the output of the first phase, the graph on the right is the output of the second phase.

#Components	Spice3	Simulation Program	Speedup
25	30s	2s	15
49	60s	5s	12
73	100s	7s	14
97	130s	10s	13
121	164s	12s	14

Figure 7: Speed of our system versus Spice3. All timings are reported in seconds. These experiments were performed on an HP9000/350 with 16MB. The circuits are all ladder networks. The circuits were simulated for 1000 timesteps. The numbers indicate that simulation programs run about 13 times faster than Spice3.

The second pass performs optimizations such as common subexpression elimination, dead code elimination, sign targeting, and constant folding. It implements these optimizations via graph transformations. Because it operates directly on the dataflow graph it can performs these optimizations very efficiently. Efficiency is very important: if the optimizer executes 5000 instructions to eliminate a given instruction, then, for the optimization to be worthwhile, the eliminated instruction would have had to been executed at least 5000 times.

The third pass performs register allocation and code generation. We are actively working on this part of the system. The code generator produces C code, which precludes any meaningful register allocation. We need to produce machine code to reap the full benefits of compiled simulation. We anticipate noticable improvements in our speedups once the code generator produces machine code. The most important issue for serial machines is better register allocation to minimize memory traffic.

These three passes are implemented in only 800 lines of code. Pass 1 is 400 lines, Pass 2 is 200 lines, and Pass 3 is 200 lines. These numbers will grow as we make the system run faster and as we move to more complex simulators. Nonetheless, we believe that outperforming Spice by at least a factor of five using only a 800 line compiler is a very interesting result.

5 Results

We have tested our system on circuits of up to 120 devices. The results are shown in Figure 7. To ensure that both simulators ran the same number of iterations we performed these experiments using a maximum step size much smaller than that needed for complete accuracy. Our system produced a simulation program written in C which was then compiled

with the same compiler used to compile Spice3. (A future version of our system will have an integral assembler.) For both simulators we counted the time to perform the simulation, not the time taken to read or write files.

Our results show our system running 13 times faster than Spice3. This speedup number is misleading and will change as our research progresses. First, we have not accounted for circuit compilation time in these figures. Because our partial evaluator is not coded for speed, and because it is running interpreted rather than compiled, circuit compilation time swamps simulation time. We have run experiments that indicate we can make the compilation time be equal to the time it takes to simulate ten timesteps.

Second, we aren't sure if our algorithms match Spice3's algorithms. In particular, Spice3 may be performing Newton-Raphson iterations. If so, it may be doing up to twice the necessary work. Once we start simulating non-linear devices the comparisons against Spice3 will be much fairer.

Third, because the partial evaluator emits C code rather than machine code, the speedups are not what they could be. We can achieve substantial reductions in memory accesses by using all available floating point registers. This is an important issue: floating point coprocessors are developing more and more registers.

For these reasons we have been conservatively stating that our system runs at least five times faster than Spice3.

There are two issues in scaling up our results to large non-linear circuits. The first issue is evaluating non-arithmetic functions such as exponentials and logarithms. These functions take longer to compute than arithmetic functions, so that as they become a larger fraction of the compute stream our speedups will decrease. We don't believe the decrease will be large. Offsetting this decrease is the time to solve the matrix, which dominates component evaluation time as circuit size grows.

The second issue is exponential blowup in compiled code size with respect to the number of nodes. Because the code for solving matrices is explicit, if there are A arithmetic operations in solving a matrix, then there will be at least A instructions in the simulation program. If we accept the experimental evidence that the complexity of matrix solution for electrical simulation is $N^{1.24}$ [Nagel] where N is the number of nodes, and pessimistically assume a five-fold space overhead factor, then, if one million instructions (4MB) were committed to solving the matrix, a circuit up to 88346 nodes could be handled. Since we don't anticipate simulating a circuit that large on a single processor, code blowup is not a problem.

6 Summary and Future Research

We have built a prototype general circuit compiler for linear circuits. It outperforms Spice3 by at least a factor of five. We are confident that we can extend the system handle Spice level simulation of large non-linear time-varying systems with equivalent speedups and no loss of accuracy.

Our novel contribution is the use of a partial evaluator to create simulation programs. This approach yields benefits on the compiler front, the simulator front, and the performance front. It wins on the compiler front because it has been very simple to create — a mere 800 lines of code produces exceptionally good results. It wins on the simulator front because we aren't tied down to a particular simulator. To speed up other types of simulations or to take advantage of special structures, such as those that arise in switched capacitive networks, we need only write a simulator and let the partial evaluator do the rest. It wins on the performance front because we are dramatically outperforming Spice3.

We have 6 tasks before us:

1. **Implementing simulation of non-linear devices.** This is straightforward and will yield more meaningful comparisons against Spice.

2. **Improving our code generation techniques.** The system outputs C code to be compiled, which incurs both an overhead time penalty and a performance penalty because C compilers are generally very stupid. We will gain performaince by having an integral assembler. We will lose portability, but simulation programs are so simple that porting the assembler will be a simple task.

3. **Improving the speed of the partial evaluator.** We need to get the time cost of compilation below the time it takes to simulate 10 timesteps.

4. **Producing code for parallel architectures.** We will investigate both tightly coupled and loosely coupled architectures.

5. **Designing economical hardware.** This hardware will execute our simulation programs at a sustained rate of 100 million floating operations per second. We want to keep five 20MH floating point chips busy with as little supporting hardware as possible.

6. **Generalizing our techniques to other scientific computations.** We believe our approach will work superbly whenever the structure of the data can be compiled away. We will have source code whose clarity and elegance is matched only by the speed of

the compiled system. Once we have Spice under our belt we will investigate using our techniques for the standard benchmarks of parallel scientific systems.

References and Bibliography

[Ardent] Lecture by Glen Miranker of Ardent at the EE380 seminar.

[ASTAP] W. T. Weeks, A. J. Jimenez, G. W. Mahoney, D. Mehta, H. Qassemzadeh, and T. R. Scott, "Algorithms for ASTAP - A Network Analysis Program," *IEEE Trans. on Circuit Theory,* Vol. CT-20, No. 6, November 1973, pp. 628-634.

[Barzilai] Z. Barzilai, J. L. Carter, B. K. rosen, and J. D. Rutledge, "HSS - A High-Speed Simulator," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* 6-4, July 1987, pp. 601-617.

[Bryant] Randal Bryant, Derek Beatty, Karl Brace, Kyeongsoon Cho, Thomas Sheffler, "COSMOS: A Compiled Simulator for MOS Circuits," in *Proceedings of the 24th Design Automation Conference,* June 1987, Miami Beach, Florida, pps 9-16.

[Lewis] David Lewis, "A Programmable Hardware Accelerator for Compiled Electrical Level Simulation," in *Proceedings of the 25th Design Automation Conference,* June 1988, Anaheim California, pps, 172-177.

[Hansen] Craig Hansen, "Hardware Logic Simulation by Compilation," *Proceedings of the 25th Design Automation Conference,* June 1988, Anaheim California, pps, 712-715.

[Nagel] Laurence Nagel, *SPICE2: A Computer Program to Simulate Semiconductor Circuits,* Electronics Research Laboratory *Report No. ERL-M520,* University of California, Berkeley, May 1975.

[Spice3] need this reference.

[Vladimirescu] Andrei Vladimirescu *et. al.*, "A Vector Hardware Accelerator with Circuit Simulation Emphasis," in *Proceedings of the 24th Design Automation Conference,* June 1987, Miami Beach, Florida, pps 89-94.

[White] Jacob K. White, *The Multirate Integration Properties of Waveform Relaxation, with Applications to Circuit Simulation and parallel Computation,* Ph.D. Dissertation, University of California, Berkeley, Electronics Research laboratory, November 18, 1985.