A Programmable Hardware Accelerator for Compiled Electrical Simulation

David M. Lewis

University of Toronto, Dept. EE, 35 St. George St., Toronto, Canada

ABSTRACT

This paper describes a high performance hardware accelerator for electrical simulation, with a speedup of over 500 for a uniprocessor. The processor addresses a variety of problems ranging from timing simulation to circuit simulation. The accelerator combines special purpose units, such as a high speed device evaluator, with fully programmable general purpose processors. The specialized processors offer extremely high speed for performance critical parts of the simulation. The general purpose processors are optimized for compiled electrical simulation, and use a very long instruction word (VLIW) architecture. The network solution is compiled into VLIW code.

This paper concentrates on those features of the machine that are designed for circuit simulation algorithms, such as SPICE. A simplified example is used to expose the hardware and software techniques used to attack the problem, and estimate the performance improvement due to each technique.

INTRODUCTION

The electrical simulation of large VLSI designs is a computationally intensive task that is difficult to do quickly. While high performance has been obtained in logic simulation by the construction of specialized hardware, the same approach has not been as successful in electrical simulation. The complexity and variety of the algorithms for various problems have made it difficult to design specialized hardware to accelerate the computations. A wide variety of software techniques have been developed for circuit and timing simulation on conventional computers, but hardware implementations of these techniques are not necessarily efficient. For this reason, most work in accelerating electrical simulation has used standard general purpose multiprocessors (eg., [1] [2]). Although substantial speedups can be obtained, the implementations that have been published tend to have substantially degraded performance with increasing numbers of processors, with severe degradation as more than a dozen or so processors are used. This reduces the cost advantage of a large collection of processors. Although further improvements can be expected in algorithm design, exploiting parallelism, and in the raw performance of processors, multiprocessor systems are likely to remain expensive and substantially slower than will be needed for large designs.

Another approach to accelerating electrical simulation has been the use of vector processors [3]. Performance is limited in these machines by the need to gather the network voltages before evaluating the device models. The memory references needed to gather the voltages are scattered through memory in a manner that reflects the network structure, limiting the performance. Vector machines are also poor at the conditional code that occurs in device models. As a result, the performance improvement is only a few percent of that suggested by the maximum performance of these architectures.

We believe that hardware support is necessary for fast electrical simulation. Furthermore, we believe that the hardware should support a range of applications. Previous hardware accelerators have tended to be large and expensive machines that only solve a single problem. Electrical simulation is much more complex than logic simulation, so an accelerator for it is likely to be costly. Electrical simulation also spans a range of problems, so it would be advantageous if a single processor could operate effectively across this range, amortizing its cost. The same machine should be able to perform circuit simulations for performance critical circuits such as a RAM, or timing simulations for complex microprocessors. The ability to implement a range of algorithms also allows bugs to be fixed and improved algorithms to be implemented as they are invented.

Awsim-3 is a hardware accelerator that addresses these problems by providing the capability to accelerate a range of electrical simulation algorithms. It has been specifically designed to support traditional direct circuit simulation, exemplified by SPICE [11], as well as its more modern variants incorporating multirate integration and latency, such as SAMSON [13]. Awsim-3 is also aimed at relaxation-based simulation algorithms, such as iterated timing analysis [12], and modern timing simulation algorithms such as MOTIS-3 [10] and CINNAMON [14].

Awsim-3 offers a significant speedup on these problems by identifying those parts of electrical simulation that are common to all of these algorithms and providing hardware support for them. The hardware support consists of both high speed hardwired processors and slower programmable processors. Operations that are common to all simulators are also time consuming, typically taking 80% to 90% of the processor time. Awsim-3 attacks this problem first by providing a hardwired processor to perform these operations. The hardwired processor is about 3000 times faster than conventional software.

The remaining 10% to 20% of processor time is spent doing different operations in different simulators. Awsim-3 identifies the common aspects of simulation algorithms, and offers both a software and hardware solution for accelerating these operations. Because a larger class of operations need to be performed, the speedup obtained is not so large. The remaining 10% to 20% of the processor time is typically accelerated by a factor of 50-300. Together, these techniques offer a typical performance improvement of 500-1500.

This paper uses an example to show how each of the techniques is used in turn to attack the problem of simulation and reduce the time taken to simulate a circuit. It also shows how an analytical perfor-

mance model can be used to estimate the performance of the machine for a given simulation algorithm. A complete description of the architecture and the support provided for all simulation algorithms would be too long, so this paper concentrates on a single algorithm and those features which support it. This algorithm is direct circuit simulation. Future papers will describe other architectural details and algorithms.

ARCHITECTURE OVERVIEW

In order to expose the strategy taken by Awsim-3 for accelerating electrical simulation, a simple example will be used. This example is considerably simpler than a real simulation algorithm, but illustrates the principles involved. The code in figure 1 is a simplified version of the code that would be used to generate the Jacobian matrix during direct circuit simulation. The vectors **ng**, **ns**, **nd**, and **nb** contain one entry per device in the circuit. Each word in each vectors contains the index of the terminal connected to a specific device's gate, source, drain, and bulk node respectively. The **np** vector contains parameters describing the characteristics of each device. The code in figure 1 gathers the voltages for each device, evaluates a device model function **f**, and loads the results into the matrix **A**.

In a typical simulator, from 80% to 90% of the processor time would be spent in evaluating the function **f**. Awsim-3's first attack on the problem is therefore to provide a fast device model evaluator. Device model evaluation is a common and time consuming operation in all electrical simulators, and is sufficiently common to merit a fully hardwired implementation. Awsim-3's model evaluator is capable of evaluating a four terminal device model function and its partial derivatives in 200ns. A device with one terminal connected to the supply (the bulk in FETs) can have three terminal functions and partial derivatives evaluated in 600ns, about a factor of 3000 faster than a typical software model. The architecture of this processor will be described in a separate paper.

```
for (i ← 1 to N)
        vg ← v [ ng [i]]
        vs ← v [ ns [i]]
        vd ← v [ nd [i]]
        vb ← v [ nb [i]]
        p ← np [i]
        x ← f (vg, vs, vd, vb, p)
        ai ← axi [i]
        aj ← axj [i]
        a [ai] [aj] ← a [ai] [aj] + x
end
```

Figure 1. Simulation Example

Device model evaluation is only one example of a function that can be accelerated by a hardwired implementation. To allow the possibility of adding other specialized processors later, the architecture of Awsim-3 is provides a mix of general purpose processors (GPs) and special purpose processors (SPs). The SPs provide completely hardwired support of some function, while the GPs perform the rest of the algorithm.

The partitioning of functionality between GPs and SPs is based upon several criteria. These include the amount of time required by a software implementation, the number of simulation algorithms which can use the function, and the performance offered by a hardwired implementation. A function should be performed by a SP if it is slow, used often, common to all algorithms, and has a cheap but fast hardware implementation.

The hardware device model evaluator meets all of these criteria, since all simulators can use it, and it offers a high speedup. For the purposes of this paper, it can be considered to be a black box that accepts four terminal voltages and a device parameter, and produces the value of a function and its partial derivatives. The device model evaluator requires 34 clock cycles to perform an evaluation, but can initiate one every two clock cycles. It can handle overlapped requests as short as a one evaluation per request.

To date, the device model evaluator is the only definite choice for an SP, although an event queue manager is a likely candidate for event driven simulation algorithms.

Having addressed the device evaluation problem, the next bottleneck remains. The remaining overhead still accounts for 10%-20% of the simulation time, limiting performance improvement to 5-10×. The attack on this bottleneck uses software and hardware techniques to improve performance. Software transformations are used to put the problem in a form that makes specialized hardware advantageous.

The software technique used to address bottleneck this is compiled simulation. Compiled simulation is not new: [7] provides a recent example. This is a purely software technique that optimizes the code in figure 1 by taking advantage of the fact that most of the operations in ths code can be statically determined. The value of **N** and the vectors **ng**, **ns**, **nd**, and **nb** are all determined by the network being simulated, and will not change during the simulation. This allows the number of iterations and the value of each of **ng[i]**, **ns[i]**, **nd[i]**, and **nb[i]** to be determined during each iteration through the loop. Assuming the values shown in figure 2(a) for the vectors, the loop can be unrolled, and the code of figure 2(b) results. Note that **vg**, etc., are now arrays. Since all of the indices into the vectors are constants, absolute addresses can be used for all memory references.

```
N = 7
ng = {1, 0, 7, 4, 2, 2, 6}
ns = {2, 3, 0, 4, 1, 1, 5}
nd = {0, 3, 1, 2, 1, 4, 5}
nb = {7, 8, 4, 1, 1, 2, 3}
p = {1.5, 1.0, 1.0, 1.0, 2.5, 2.5, 1.5}
ai = {0, 0, 1, 1, 2, 2, 3}
aj = {0, 1, 0, 1, 2, 3, 2}
```

(a) Example Values

```
vg [0] ← v [1]; vs [0] ← v [2]; vd [0] ← v [0]; vb [0] ← v [7]; p [0] ← 1.5;
vg [1] ← v [0]; vs [1] ← v [3]; vd [1] ← v [3]; vb [1] ← v [8]; p [1] ← 1.0;
vg [2] ← v [7]; vs [2] ← v [0]; vd [2] ← v [1]; vb [2] ← v [4]; p [2] ← 1.0;
vg [3] ← v [4]; vs [3] ← v [4]; vd [3] ← v [2]; vb [3] ← v [1]; p [3] ← 1.0;
vg [4] ← v [2]; vs [4] ← v [1]; vd [4] ← v [1]; vb [4] ← v [1]; p [4] ← 2.5;
vg [5] ← v [2]; vs [5] ← v [1]; vd [5] ← v [4]; vb [5] ← v [2]; p [5] ← 2.5;
vg [6] ← v [6]; vs [6] ← v [5]; vd [6] ← v [5]; vb [6] ← v [3]; p [6] ← 1.5;
x [0] ← f (vg [0], vs [0], vd [0], vb [0], p [0]);
x [1] ← f (vg [1], vs [1], vd [1], vb [1], p [1]);
x [2] ← f (vg [2], vs [2], vd [2], vb [2], p [2]);
x [3] ← f (vg [3], vs [3], vd [3], vb [3], p [3]);
x [4] ← f (vg [4], vs [4], vd [4], vb [4], p [4]);
x [5] ← f (vg [5], vs [5], vd [5], vb [5], p [5]);
x [6] ← f (vg [6], vs [6], vd [6], vb [6], p [6]);
a [0] [0] ← a [0] [0] + x [0];
a [0] [1] ← a [0] [1] + x [1];
a [1] [0] ← a [1] [0] + x [2];
a [1] [1] ← a [1] [1] + x [3];
a [2] [2] ← a [2] [2] + x [4];
a [2] [3] ← a [2] [3] + x [5];
a [3] [2] ← a [3] [2] + x [6];
```

Figure 2. (b) Unrolled Code

The speedup offered by this technique is not dramatic; it might be a factor of three for typical simulators. Thus, in combination with the device model evaluator, the total speedup is still only about a factor of 15-30. The remaining speedup comes from examining the properties of the code in figure 2(b) and designing a processor that executes code with these properties quickly.

The properties can be summarized by four observations. First, there are very few branches. This eliminates one of the main problems in designing a fast machine, and allows a highly pipelined architecture. Second, there is a high degree of parallelism available. This parallelism encourages both parallel and pipelined architectures, and allows a large number of operations to be performed concurrently. Third, there are a large number of absolute memory references, but the access patterns are not regular. This has two implications: it makes parallelism easy to find, since any two instructions which don't refer to the same addresses can be executed concurrently, and it means that the processor should have a high memory bandwidth even with arbitrary access patterns. The unpredictable patterns of these accesses are one of the problems that limited the performance of vector processors on circuit simulation [3]. Fourth, information that was previously represented as data (**ng**, etc.) is now represented by code. There is therefore a large amount of code, but little data space is required. Although **vg**, etc., are now vectors, they don't need to have N entries; it is possible to perform several evaluations on a subset of the devices using a smaller **vg**, etc. It is only necessary to make **vg** long enough to amortize any startup overhead.

The Awsim-3 GP is designed to execute programs that have these properties. The GP is based upon a very long instruction word (VLIW) architecture [5], allowing multiple instructions to be executed per cycle. VLIW's have been previously proposed as hardware accelerators [4], but our architecture relies upon more software transformations of the problem. Fortunately, the small number of branches and simple addressing modes of the programs make the elaborate compilation techniques used for other VLIWs [6,8] unnecessary.

The GP takes advantage of the first two properties by allowing five instructions to be executed per cycle, and using a pipeline 13 clock cycles long. The GP accommodates the third and fourth properties by having a relatively small data memory, but allowing up to five accesses per cycle.

A simplified view of the data paths of the GP appears in figure 3. The processor has five banks of registers, RA through RE, and five banks of memory, MA through ME. Each bank can use independent addresses. The GP executes very long instructions (VLIs) that contain up to five instructions. We will use VLI to mean an instruction as executed by the GP, and instruction to mean a conventional two or three address instruction.

Each VLI allows the register banks to be read, memory accessed, arithmetic operations performed, and registers to be written with the results. there are five slices, each containing a set of registers, memory, and arithmetic units. Crossbars allow data to be routed from one slice to others. Each slice supplies two operands, one from registers, and one from memory. The crossbars allow the any of the operands to be sent to any of arithmetic units. A subsequent stage of crossbars allows any result to be written back into any register bank.

The high degree of interconnection between slices simplifies packing instructions into VLI's. As long as instruction ordering constraints are observed, packing instructions into VLIs is easily accomplished by finding instructions that reference different register and memory banks, and appropriate arithmetic units are available to do the work.

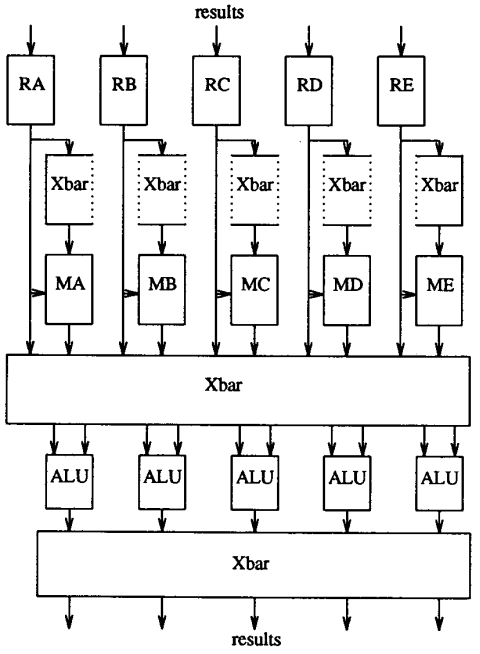

Figure 3. Data Paths

In order to efficiently use the resources provided by the processor, data is allocated in a manner that tends to make parallelism easy to find. In the example above, words of **v** are accessed randomly, but some locality of reference can be expected with real circuits. The vector **v** is placed with sequential words in different banks to increase the probability that a short sequence of code uses different banks. **v[0]** is placed at **MA@v**, **v[1]** is placed at **MB@v**, and so on up to **v[8] at MD@v+1**.

One element of each of the vectors **vg**, etc., is accessed per device evaluation, so each vector is placed in a single bank, **vg** in **RA**, etc. The hardware device evaluator requires that its parameters be in these registers, and returns its results in registers. In the example **f** only returns a single result per evaluation in **RA**, and is not efficient. In reality the device evaluator returns a function value in **RA** and its partial derivatives in other banks.

Finally, the array **a**, if treated as a sparse array, will have its nonzero elements allocated sequentially in different banks.

```
mov   MB@v,RA@vg
mov   MC@v,RB@vs
.
.
.
mov   ME@consts+3,RE@p+6
eval  vg,x,7
fadd  MA@a,RA@x,RA@temp1
mov   RA@temp1,MA@a
fadd  MB@a,RA@x+1,RB@temp2
mov   RB@temp2,MB@a
.
.
.
```

Figure 4. Assembler Code for Example

Figure 4 shows the assembler code that would be produced by a compiler for the example problem. This code is packed into VLIs by a instruction packer, which attempts to use all resources while obeying code ordering and pipeline constraints.

PERFORMANCE MODELLING

The performance of this processor will be determined by simulating the architecture running some real problems, and ultimately by constructing the machine. It is a considerable effort to develop this system, so we use analytical models to predict the performance before constructing the machine. The models are based upon a description of the algorithm that is at sufficiently detailed that it can be hand translated into machine code. This analysis calculates the exact number of processor cycles required. For this analysis to be accurate, our presentation must break the simulation into steps that are sufficiently simple to determine the exact machine code and number of cycles required. The result is unfortunately, but necessarily, detailed. This detail at least has the advantage that it illustrates the techniques used to structure the problem for solution on Awsim-3.

As an example, we present a performance analysis of direct simulation. A charge conserving formulation of the problem is expressed as:

$$\dot{Q}(v,u) = G(v,u)$$

where $v \in R^N$ is a vector of nodal voltages to be solved, $u \in R^M$ is a vector of voltages on input nodes, $Q \in R^{N+M} \rightarrow R^N$ is a vector of charges, and $G \in R^{N+M} \rightarrow R^N$ is a vector of currents.

We will describe the solution of this problem using direct simulation, backward Euler, and Newton iteration. This requires the discretization of time into time points t_k, $t_k = t_{k-1} + h_k$, and approximating $\dot{Q}$, as follows:

$$\dot{Q}_k(v,u) = \frac{Q_k(v_k,u_k) - Q(v_{k-1},u_{k-1})}{h_k}$$

The system of equations then becomes:

$$F(v_k) = h_k \times G(v_k,u_k) - Q(v_k,u_k) + Q(v_{k-1},u_{k-1}) = 0$$

This can be solved for v_k using Newton's method:

$$v_k^{i+1} = v_k^i - A^{-1}(v_k^i,u_k) \times F(v_k^i,u_k)$$

$$A_{i,j} = \frac{\partial F_i}{\partial v_j}$$

An accurate performance analysis of this problem requires more detailed formulation of the problem. This formulation is more closely related to the implementation and takes advantage of the special structure of the problem, and introduces optimizations for linear components. A useful specialization is networks containing active devices, linear capacitors, and linear resistors, which are faster to load into the matrix A. It is also useful to distinguish linear capacitors with one terminal grounded, which are even faster. Furthermore, devices are assumed to have 3 terminals in the example. Although MOSFETs are 4 terminal devices, in most circuits the bulk terminal is connected to a power supply, allowing Q_B to be ignored. Some of the other terminals are also connected to power supplies, so the number of terminals per device, T, is distinguished from the effective number of terminals per device, T_{eff}. The latter is the average number of terminals per device that are not connected to a power supply. The symbols for the various quantities, and actual values for an example, are shown in Table 1. The example circuit is a 478 transistor CMOS 16 bit counter using domino logic. The numbers J, β, and N_i reflect the number of operations required for matrix solution and the number of iterations required to solve each timestep. The number of non-zero entries in the matrix is assumed to be J×N, and decomposition time is assumed to be $(J \times N)^{\beta}$. A node tearing formulation is used [9]. All numbers in the example are taken from a implementation of direct solution on this circuit.

sym	meaning	example
N	nodes	264
D	T-terminal non-linear devices	478
T	terminals per device	3
T_{eff}	terminals per dev, not connected to power supply	2.5
F	linear 2-terminal capacitors	0
C	linear 2-terminal capacitors, one terminal grounded	264
R	linear 2-terminal resistors	0
J	Jacobian matrix entries per node	8.8
N_i	number of Newton iterations	2.3
β	complexity exponent of matrix decomposition	1.15

TABLE 1. Symbols for Direct Simulation Example

Using these symbols, the problem is specialized into several G and Q corresponding to the different kinds of devices.

$$G(v,u) = GA(v,u) + GR(v,u)$$
$$GR(v,u) = AR \times [v^T, u^T]^T$$
$$Q(v,u) = QA(v,u) + QC(v,u) + QF(v,u)$$
$$QC(v,u) = AC \times v$$
$$QF(v,u) = AF \times [v^T, u^T]^T$$

GA is a vector of currents due to active devices, and GR is a vector of currents caused by resistors. Similarly, QA is a vector of charges stored in active devices, QC is a vector of charges on grounded capacitors, and QF is a vector of charges on floating capacitors. The matrices $AR \in R^{(N+M)\times(N+M)}$, $AF \in R^{(N+M)\times(N+M)}$, and $AC \in R^{N\times N}$ give the values of the admittance matrices for resistors, floating capacitors, and one terminal grounded capacitors. Each of these matrices is sparse, and the number of non-zero entries in each matrix is the number of components (times two for AR and AF).

Exact analysis of the performance requires further detail about the structure of QA and GA. These can be considered to be the sum of T separate vectors, QD_i and GD_i, where each vector represents the contribution of one terminal of all the devices. For example, number the device terminals as gate=1, source=2, drain=3. Then QD_1 is the vector containing the gate charge for each of the active devices in the circuit. The contributions of each QD_i and GD_i to F are described by the connection matrices $AD \in R^{T\times(N+M)\times D}$. Each of the T matrices AD_i describe the connection of the terminal i of all devices. That is, $AD_{i,j,k}$ is 1 if terminal i of device k is connected to node j, else it is 0.

Each device makes T^2 contributions to the partial derivatives of Q and G. $\frac{\partial QA_i}{\partial v_j}$ and $\frac{\partial GA_i}{\partial v_j}$ are thus considered to be the sum of T^2 separate arrays, $DG_{t1,t2}$ and $DQ_{t1,t2}$. $DQ_{t1,t2}$ is a matrix that gives the contribution of all devices' $\frac{\partial Q_{t1}}{\partial v_{t2}}$ towards the matrix $\frac{\partial QD_i}{\partial v_j}$. Eg., $DQ_{1,2}$ is a matrix giving the contribution of all devices $\frac{\partial Q_G}{\partial v_S}$ towards the matrix A. DQ and DG are related to the vectors $fdq_{t1,t2}(v,u)$, and $fdg_{t1,t2}(v,u)$, which give the individual contributions of each device.

For example, $fdq_{2,3}$ contains the value of $\frac{\partial Q_S}{\partial v_D}$ for all devices. The contributions to $\frac{\partial F_i}{\partial v_j}$ of these charges and currents are described by $DAD \in R^{T\times T\times(N+M)\times(N+M)\times D}$. These constitute T^2 topological tensors $DAD_{i,j}$, which describe the connections corresponding to $\frac{\partial QD_i}{\partial v_j}$ and $\frac{\partial GD_i}{\partial v_j}$. $DAD_{i,j,k,l,m}$ is 1 if device m has terminal i connected to node k and terminal j connected to node l.

line	code	ops (per exec of line)	exec count (total)
1	$V_k^0 \leftarrow \frac{h_k}{h_{k-1}}\times\left(V_{k-1}-V_{k-2}\right)+V_{k-1}$	4N	1
2	gather V_{k-1} into registers	5D	1
3	$QP \leftarrow 0$	N	1
4	for (t $\leftarrow$ 1..T)		
5	$Q \leftarrow fq_t(V_{k-1})$	D (*)	1
6	$QP \leftarrow QP + AD \times Q$	$2D\times\frac{T_{eff}}{T}$	T
7	end for		
8	$QP \leftarrow QP + AF \times V_{k-1}$	6F	1
9	$QP \leftarrow QP + AC \times V_{k-1}$	3C	1
10	$i \leftarrow 0$		
11	while (not converged)		
12	gather V_k^i into registers	5D	N_i
13	$B \leftarrow 0$		
14	for (t $\leftarrow$ 1..T)		
15	$Q \leftarrow fq_t(V_k^i)$	D (*)	$T\times N_i$
16	$G \leftarrow fg_t(V_k^i)$	D (*)	$T\times N_i$
17	$B \leftarrow B + AD_t \times (h_k \times G + Q)$	$4D\times\frac{T_{eff}}{T}$	$T\times N_i$
18	end for		
19	$B \leftarrow B + h_k \times AR \times V_k$	7R	N_i
20	$B \leftarrow B + AF \times V_k$	6F	N_i
21	$B \leftarrow B + AC \times V_k$	3C	N_i
22	$B \leftarrow B - QP$	2N	N_i
23	$A \leftarrow 0$		
24	for (t1 $\leftarrow$ 1..T)		
25	for (t2 $\leftarrow$ 1..T)		
26	$DG \leftarrow fdg_{t1,t2}(V_k^i)$	(†)	$T^2\times N_i$
27	$DQ \leftarrow fdq_{t1,t2}(V_k^i)$	(†)	$T^2\times N_i$
28	for (j,k $\leftarrow$ 1..N)		
29	$A_{jk} \leftarrow A_{jk} + DAD_{t1,t2,j,k} \times (h_k \times DG+DQ)$	$4D\times\left(\frac{T_{eff}}{T}\right)^2$	$T^2\times N_i$
30	end for		
31	end for		
32	end for		
33	$A \leftarrow A + h_k \times AR + AF + AC$	5R + 4F + 2C	N_i
34	solve $A\times x = B$ for x	$3\times(D\times J)^\beta$	N_i
35	$V_k^{i+1} \leftarrow V_k^i - x$	N	N_i
36	dvmax $\leftarrow \|x\|_\infty$	N	N_i
37	LTE $\leftarrow \|V_{k-1} - V_k^{i+1}\|_\infty$	N	N_i
38	end for		
39	end while		

(†): derivatives are available from evaluation of fq and fg.
(*): each of these takes two clock cycles, equivalent to 10 instructions

TABLE 2. Direct Simulation Example

The analysis of the execution time for this simulation is shown in Table 2. Column 2 shows the code, which is written to emphasize its similarity with the formulation above. In the implementation, all **for** loops are unrolled and code between loops is merged. Only the **while** loop beginning in line 11 is not unrolled, since there would be no efficiency gain, and it is executed a variable number of times. Explicit assignments of 0 (lines 3,12,20) generate no code, because the first operation which uses the value is compiled to use 0 instead. Column 3 shows the number of instructions per execution of the line, without considering the unrolling. Column 4 shows the number of times the line would be executed in that formulation.

The number of instructions required can be calculated by counting the resource requirements. There are three kinds of resources: register references, memory references and arithmetic operations. The number of operations is calculated as the maximum of the number of memory references and the number of arithmetic operations. For example, consider the most time consuming line in the setup, $A_{jk} \leftarrow A_{jk} + DAD_{t1,t2,j,k} \times (h \times DG + DQ)$. DAD is binary valued, so no multiplication is required. The code for a single terminal of a single device might look like:

```
fmul    MC@h,RB@dg+15,RC@temp34
fadd    RB@dq+15,RC@temp34,RC@temp34
fadd    MA@a+23,RC@temp34,RC@temp34
mov     RC@temp34,MA@a+23
```

This requires five register reads, four memory reads, and three operations. The performance limit is the five register reads, so although the instructions will be placed in separate VLIs, they can be considered to take one VLI in total.

The estimated code size is 70918 instructions, for 14184 VLIs, and the total time predicted for a simulation timestep of the example circuit is 4.7 ms. This comprises $(1421 + N_i \times 12762) = 31548$ VLI instructions at 100ns each, and $D \times (2N_i + 1) \times T = 8030$ device function evaluations (of one terminal per evaluation) at 200ns each. If we estimate the time required for a fast software implementation as 2ms per device evaluation and ignore the time required for other computations, the total time on a conventional processor would be $D \times (N_i + 1) \times 2$ms, or 3.1 seconds. The uniprocessor hardware accelerator is estimated to be 660 times faster.

The actual circuit compiler is both better and worse. It keeps track of the location of variables, and so avoids redundant stores. On the other hand, it explicitly sets variables to 0, despite our previous claim. This will eventually be eliminated. The test circuit compiles into 79012 instructions, and is expected to take 5.0ms to execute. Most of the difference is is in the matrix solution stage, and is due to simplified operation counting in the program that estimates the number of operations required.

Simple improvements can increase performance approximately 10%. The preceding analysis has assumed that all devices have current in all 3 terminals. The fact that a MOS device has only two terminals that conduct current, and the fact that $I_D = -I_S$ can be used to reduce the number of device current evaluations from three to one. The number of arithmetic operations are also reduced, leading to about a 10% performance improvement.

A simple model estimates performance improvement for other simulation algorithms. Assume that a software simulator spends the

fraction f_{eval} of its time doing device evaluation, and f_{comp} doing other operations, $f_{eval}+f_{comp}=1$. Awsim-3 accelerates these operations by a_{eval} and a_{comp}. The total acceleration is then $\frac{1}{\frac{f_{eval}}{a_{eval}}+\frac{f_{comp}}{a_{comp}}}$. a_{eval} is large, typically $\frac{2ms}{3\times200ns}=3333$. a_{comp} is small and varies over a wider range, but is estimated at 50-300. Of this, a factor of two to three is due to the use of compiled simulation, and a factor of 25-100 due to specialized hardware for executing the resulting programs. Assuming $f_{eval}=.9$, and $f_{comp}=.1$, the resulting acceleration is in the range of 440 to 1660. Considering the direct simulation example, with a speedup of 660, and solving $\frac{1}{\frac{f_{eval}}{a_{eval}}+\frac{f_{comp}}{a_{comp}}}=660$ with the above assumptions gives $a_{comp}=81$ in this example. Direct simulation is believed to be towards the low end of the performance range because of the large amount of time spent solving linear equations (i.e., f_{comp} is probably actually larger than .1 for this example).

Awsim-3's architecture makes it important to use algorithms that perform large numbers of model evaluations, since they have the highest performance improvement. In contrast to software implementations, device evaluation is a relatively inexpensive operation in Awsim-3. Algorithms that work well on conventional hardware by trading device evaluations for simpler operations are not necessarily optimal for execution on Awsim-3.

Awsim-3 calls attention to some broader issues in hardware accelerators. Hardware accelerators often benefit from both hardware and software techniques without explicitly analysing the performance improvement attributable to each technique. Hardware accelerators for logic simulation typically use compiled simulation on special purpose hardware, but do not quantify the performance improvement due to each of these techniques. We hope to have been more careful here.

IMPLEMENTATION STATUS

We are continuing work on Awsim-3, and hope to construct a processor and run several simulation algorithms on it. A compiler for direct circuit simulation already outputs code corresponding to about 98% of the time estimated by the analytical model. An instruction level simulator that is faithful to all pipeline properties of the machine has been running for several months. The instruction packer that packs instructions into VLIs is barely working, but initial results are encouraging. System integration is expected to be a non-trivial task.

CONCLUSIONS

Awsim-3 offers high performance electrical simulation by combining general purpose processors with specialized processors, such as a hardware device modeller. The high performance of the specialized processors makes it necessary to use compiled simulation for maximum speed. The general purpose processor architecture is tailored for compiled simulation, and offers high bandwidth to the specialized processors. An analysis of direct circuit simulation suggests that it can be accelerated by a factor of 660, while other simulation algorithms can also be run at high speed.

ACKNOWLDEGEMENTS

This work was supported in part by the Natural Sciences and Engineering Council of Canada and by Bell Northern Research.

Tom Blank's comments on this paper are greatly appreciated.

REFERENCES

[1] J. T. Deutsch, "Algorithms and Architecture for Multiprocessor-Based Circuit Simulation", ERL Memorandum M85/39, Electronics Research Laboratory, University of California, Berkeley, May, 1985

[2] C. Dyson and A. Gray, "Mixed Mode Simulation on Transputers", in *International Workshop on Hardware Accelerators*, University of Oxford, Oct, 1987

[3] A. Vladimirescu and D. O. Pederson, "Circuit Simulation on Vector Processors", in *IEEE International Conference on Circuits and Computers*, 1982, pp 172-175

[4] S. P. Smith and B. Wood, "Architecture of a General Accelerator for CAD: The Proteus-1", in *Proceedings of the International Workshop on Hardware Accelerators*, University of Oxford, Oct., 1987

[5] J. A. Fisher, "Very Long Instruction Word Architectures and the ELI-512", in *Proceedings of the 10th Annual International Symposium on Computer Architecture*, June 1983, pp 140-150

[6] J. A. Fisher, "Trace Scheduling: A technique for global microcode compaction", in *IEEE Transaction on Computers* 21-12 , July 1981, pp 1411-1415

[7] 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

[8] J. R. Ellis, "Bulldog: A Compiler for VLIW Architectures", MIT Press, 1985

[9] I. N. Hajj, "Sparsity Consideration in Network Solution by Tearing", in *IEEE Transactions on Circuits and Systems*, 27-5, May 1980, pp 357-366

[10] D. Tsao and C.-F. Chen, "A Fast Timing Simulator for Digital MOS Circuits, in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 5-4, Oct. 1986

[11] A. Vlaimirescu and S. Lui, "The Simulation of MOS Integrated Circuits using SPICE2", Electronics Research Laboratory Memo UCB/ERL M80/7, University of California, Berkeley, 1980

[12] R. A. Saleh, "Nonlinear Relaxation Algorithms for Circuit Simulation", ERL Memorandum M87/21, Electronics Research Laboratory, University of California, Berkeley, April, 1987

[13] K. A. Sakallah and S. W. Director "SAMSON2: An Event Driven VLSI Circuit Simulator", in *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, 4-4, Oct. 1985

[14] L. M. Vidigal, S. R. Nassif, and S. W. Director, "CINNAMON: Coupled Integration and Nodal Analysis of MOS Networks", in *Proceedings of the 23rd Design Automation Conference*, 1986, pp 179-185