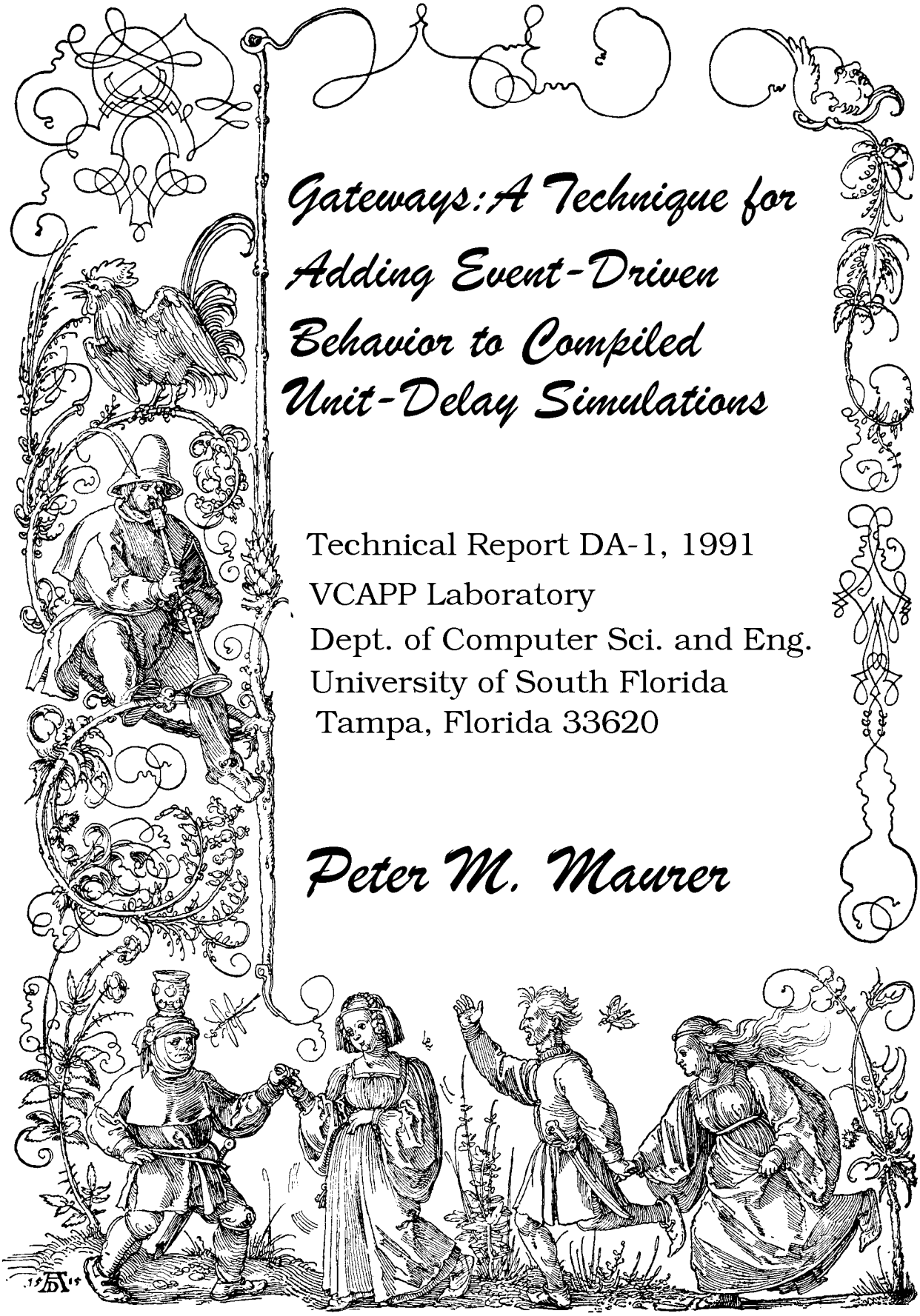

Gateways: A Technique for Adding Event-Driven Behavior to Compiled Unit-Delay Simulations

Technical Report DA-1, 1991

VCAPP Laboratory
Dept. of Computer Sci. and Eng.
University of South Florida
Tampa, Florida 33620

Peter M. Maurer

GATEWAYS: A TECHNIQUE FOR ADDING EVENT-DRIVEN BEHAVIOR TO COMPILED UNIT-DELAY SIMULATIONS

Peter M. Maurer
Department of Computer Science and Engineering
University of South Florida
Tampa, FL 33620

ABSTRACT

The gateway technique is a method for switching segments of code into and out of the instruction stream. When added to the straight-line code generated by a compiled simulator, gateways can be used to enhance the performance of the generated code by switching only those segments of code that actually need to be executed into the instruction stream. The convergence algorithm is an oblivious compiled code algorithm that can be used with many different types of circuits, including cyclic asynchronous circuits. In its oblivious form, the convergence algorithm provides only modest gains in performance over interpreted event-driven simulation, but with the addition of gateways, the performance of the algorithm increases significantly. Experimental data shows that with gateways, the convergence algorithm runs in about 1/5th the time required for an interpreted event-driven simulation. Additional work has been done to reduce the amount of code generated by the convergence algorithm, and to enhance the locality of the code to improve its performance on machines with caches.

GATEWAYS: A TECHNIQUE FOR ADDING EVENT-DRIVEN BEHAVIOR TO COMPILED UNIT-DELAY SIMULATIONS*

Peter M. Maurer
Department of Computer Science and Engineering
University of South Florida
Tampa, FL 33620

1. Introduction.

Recently there has been considerable interest in using compilation techniques as a means of improving the performance of various types of simulation [1-9]. With some exceptions [1,2,5], the simulators in question are oblivious simulators [3,4,7]. The term "oblivious" means that neither the values of the inputs nor the values produced by simulating various gates (or transistors in the case of switch-level simulation) affect the progress of the simulation. The amount of work done per input vector is constant. In contrast to oblivious simulation, event driven simulation tests input values and the values produced by gate simulations to reduce the number of gates simulated. The amount of work done varies widely from vector to vector. (In this paper, we consider the terms "non-oblivious" and "event-driven" to be synonymous.)

Although oblivious techniques typically simulate many more gates than event-driven techniques, the overhead of scheduling gate simulations is eliminated, which allows oblivious simulation to outperform interpreted event-driven simulation except when the activity rate (the percentage of gates actually simulated for event-driven simulation) is very low. The number of gate simulations required in an oblivious simulation depends on the timing model used, and on the structure of the circuit. For zero-delay simulation of acyclic circuits it is sufficient to simulate every gate once. For unit-delay simulation of the same circuit it will usually be necessary to simulate the same gate several times.

The PC-Set method [3] is an oblivious technique that is applicable to acyclic and synchronous cyclic circuits. A technique similar to levelization is used to determine the possible times at which the inputs of a gate are permitted to change, and one gate simulation is generated for each potential change in the input nets of the gate. In a sense, this approach determines the maximum number of events that could occur during the simulation of a circuit, and generates code for the worst-case number of events.

The PC-Set method works well for acyclic circuits, but unfortunately, when the same concept is applied to asynchronous cyclic circuits, less satisfactory results are obtained. The convergence algorithm (which is described in detail in the next section) is an extension of the PC-Set method to acyclic synchronous circuits. For each unit of simulated time, the convergence algorithm determines the maximum number of events that could occur in that time period and generates code to handle the worst-case number of events. As time progresses, the number of potential events per unit time tends to increase until a convergence point is reached. Once the convergence point is reached, the same set of gates is simulated repeatedly until no further changes take place. For most of the test-circuits we

---

* Manuscript Received _____. This work was supported in part by the National Science Foundation under grant number MIP-906444 and the USF Center for Microelectronics Research (CMR).
The author is with the Department of Computer Science and Engineering, University of South Florida, Tampa, FL 33620

have tried, the repetitive portion of the code includes most, if not all of the gates in the circuit. This is in stark contrast to the behavior one would normally expect from a well-designed circuit. In most cases, one would expect the number of events to decrease sharply as time progresses. If it were necessary to loop through a sequence of gates several times, one would expect this set of gates to be quite small.

Because of the huge number of gates simulated, the convergence algorithm performs only slightly better than event-driven interpreted simulation in the best case, and actually performs worse for some circuits. After experimenting with the convergence algorithm and other oblivious techniques for simulating asynchronous cyclic circuits, we came to the conclusion that these techniques generated far too many gate simulations to be useful in practice. We experimented with some rough techniques for adding event-driven behavior to oblivious simulations, but these led to relatively modest gains in performance. Finally we were led to the gateway technique, which can be used to introduce event-driven behavior at various levels of granularity. At the current time we have used the gateway technique only in a limited way, but the results have been quite satisfactory. For the circuits tested, the compiled code runs in one fifth the time required for an interpreted event-driven simulation. The circuits tested came from the ISCAS-89 sequential benchmarks and were simulated as asynchronous circuits. We have experimented with other techniques that give better performance, but only if the test circuits are simulated as synchronous circuits.

2. The Gateway Technique.

The fundamental concept behind the gateway technique is the idea of switching code segments into and out of the instruction stream. In most cases, the generated code will resemble straight-line oblivious code. Gate-simulation code is generated in more or less the same manner as in existing oblivious algorithms, with "gateways" or retargetable branches added to allow portions of the code to be switched into and out of the instruction stream. Additional code is generated to manipulate the gateways. Although we have focused on one particular code-generation scheme, we believe that gateways can be used to enhance many different code generation schemes.

The basic gateway structure is illustrated in Fig. 1. The code in this figure is expressed in the C language. Gateways are represented by statements of the form "Goto *variable" which represents an indirect branch. Since this is not legal C, our compiler actually generates assembly code for the gateway operation. Additional assembly code is generated to create labels and to initialize variables with label addresses. In all other cases, C code is generated.

```
Current_Gate = Address_of_Gateway_1;
*Current_Gate = Address_of_Exit;
Test Primary Input #1;
...
Test Primary Input #n;
Goto *Gateway_1;          /* GATEWAY */
...
Gate simulation code;
...
Exit:
     return;
```

Fig. 1. The basic gateway structure.

In Fig. 1, the first gateway terminates the primary input tests. This gateway is initially aimed at the return statement, but the primary input tests may retarget it into the simulation

code. The variable "Current_Gate" is used for this purpose. During the simulation, "Current_Gate" always points to the gateway that terminates the last segment of code to be executed. This variable (which is kept in a register for efficiency) is used by various routines to switch code segments into the execution stream, as Fig. 2 illustrates.

```
if (new_net_k != old_net_k)
{
    old_net_k = new_net_k;
    *Current_Gate = Address_of_Label_i;
    Current_Gate = Address_of_Gateway_i;
}
```

Fig. 2. An input test.

Fig. 2 illustrates a test for a primary input which fans out to a single gate. Depending on the simulation algorithm, it may also be necessary to add a third statement to initialize Gateway_i. Because it is possible to assign the variable "Current_Gate" to a register, the gateway-manipulation statements of Fig. 2 will require only one instruction each on most architectures. The gateway itself can also be implemented as a single instruction on some architectures, if the proper addressing modes are available. (In our current SUN-4 implementation, 3 instructions are required for this operation.) The two final statements of Fig. 2 are used to add block *i* to the execution stream. Block *i* begins with the label "Label_i" and ends with the gateway statement "goto *Gateway_i." Because adding a block to the execution stream is an extremely common operation, it will be designated by the function call "AddBlock(i)" in the remaining figures. Similarly, the gateway statement will be designated by the function call "Gateway(i)."

Although we believe that gateways can be used with many different underlying algorithms, the work reported here is based on the convergence algorithm because this algorithm provides several opportunities for optimizing the generated code. Although gateways can be used at different levels of granularity, in the work presented here we chose to use them in a way that is reminiscent of interpreted event-driven simulation, because this tends to minimize the number of gates simulated.

The convergence algorithm is a technique for generating compiled unit-delay code, and is applicable to many types of circuits, including asynchronous sequential circuits. This algorithm first generates a sequence of states, S_1 through S_n, each of which contains one or more gates from the circuit to be simulated. (A particular gate may appear in several states.) Each state contains the gates that will be simulated at a particular time-step. State S_1 contains all those gates that are directly connected to primary inputs or constant signals. For $k>1$, S_k contains all those gates that are direct successors of the gates in S_{k-1}. For acyclic circuits, the sequence terminates when an empty state is generated. For cyclic circuits, the algorithm will eventually generate two states S_i and S_j such that $i<j$ and $S_i=S_j$. If j is the smallest integer such that $S_j=S_i$ for some $i<j$, then the sequence terminates with S_{j-1}. The subsequence S_1 through S_{i-1} is called the prefix portion and the subsequence S_i through S_{j-1} is called the iterative portion. Code is generated in ascending order by state number. The code for the prefix portion is executed unconditionally, while the code for the iterative portion is executed repeatedly until the outputs of the gates in S_{j-1} stabilize. In practice we have found it necessary to add additional tests to reduce the execution time of the generated code to an acceptable level.

In unit-delay simulation, the value of a net may change several times during the simulation of a single input vector. For some algorithms, such as the PC-set algorithm [3], it is necessary to generate several variables for each net to guarantee that the correct input values are used for each gate-simulation. For algorithms such as the convergence algorithm, which simulate gates in order by time, it is sufficient to store the results of each

gate simulation in temporary variables until all gates for the time-step have been simulated. The convergence algorithm eliminates some of these variables by scheduling gates in order by net usage, but with the addition of gateways, it is no longer possible to guarantee that gate simulations are done in any particular order, so temporary variables must be used for all gate simulations. The use of temporary variables implies that processing must have at least two phases, one for gate simulation, and one for copying temporary results into their permanent locations. The second phase of processing also provides a convenient place for testing gate outputs for changes.

The simulation code generated by the convergence algorithm for a particular state consists of two segments, a gate processing segment and a net processing segment. The gate-processing segment consists of one simulation routine for each gate in the state, while the net-processing segment consists of one routine for each gate-output in the state. These routines can be individually switched in and out of the instruction stream.

In addition to the simulation operations, it is necessary to perform other tasks such as printing output vectors, and testing for oscillations. Net-trailer and gate-trailer routines are generated for each state to provide a convenient place to perform these operations. These routines are always the final routines executed in their respective phases. Because these routines will always be executed, it is possible to eliminate the gateway initialization statement from routines such as that illustrated in Fig. 2. Fig. 3 illustrates the four basic routines that are used to simulate a single state. The code of Fig. 3 immediately follows the code illustrated in Fig. 2.

```
/* net trailer */
Label_1:
        /* test for termination */
        if (*current != Address_of_Exit)
        {
                /* Add Gate Trailer */
                AddBlock(3);
        }
        Print_Output_Vector();
        Gateway(1);

/* gate simulation for AND gate */
Label_2:
        new_net_a = old_net_b & old_net_c;
        /* Add a net handling routine */
        AddBlock(4);
        Gateway(2);

/* gate trailer */
Label_3:
        Oscillation_Test();
        /* Add next net trailer */
        AddBlock(x);
        /* used to test for termination */
        *current = Address_of_Exit;
        Gateway(3);

/* net handling routine */
Label4:
        if (new_net_a != old_net_a)
        {
                old_net_a = new_net_a;
                /* Add gate simulation routine(s) */
                AddBlock(y);
        }
        Gateway(4);
```

Fig. 3. The structure of the generated code.

The code illustrated in Figs. 2 and 3 contains a small but important bug. If a gate has two or more inputs, both of which change in the previous state simulation, the gate can be added to the execution stream twice. In most cases, this will cause some blocks of code to be lost from the execution stream. To get around this problem a flag is established for each gate which indicates whether it is currently switched into the execution stream. Before switching a gate simulation routine into the execution stream, the net-handling routine tests the flag, and if it is set, it does not perform the switch. The flag is set when the routine is first switched into the execution stream, and reset by the gate simulation routine. If the circuit contains wired connections (i.e. nets that are driven by two or more gates), similar precautions must be taken for net-handlers. Since none of our test circuits contain such connections, these precautions are unnecessary for these circuits.

Printing output vectors is done in the net trailer routine, because this is the final step in the simulation of any state, and because at this point, all temporary values produced by the gate simulation routines have been made permanent. Oscillation testing is done in the gate trailer routine, because at this point all gate flags have been reset, and it is safe to abort the

simulation. Aborting the simulation at this point also simplifies the process of identifying those nets that are in oscillation, because the temporary values produced by the simulation routines have not yet been copied into the permanent variables. Oscillation testing is done *only* in the first state of the iterative portion of the code. When the number of times the iterative portion has been executed exceeds a user-specified limit, the simulation of the current input vector is aborted.

Each of the four types of routines illustrated in Fig. 3 can switch exactly one other type of routine into the execution stream. Gate trailers add Net trailers, and vice-versa. Gate simulation routines add net handlers, and vice versa. The gate simulation routines will add net handlers for the same state. The net handlers add gate simulation routines from the following state. For sequential circuits, the net handlers for the final state add gate simulation routines from the first state in the iterative portion. Fig. 4 clarifies this relationship between the various types of routines. The solid lines in Fig. 4 represent gateways, while the dotted lines show which routines establish the gateway targets.

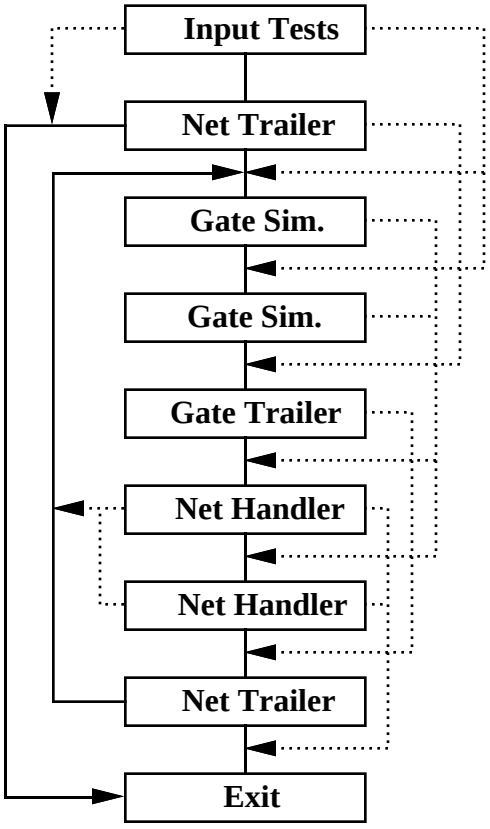

Fig. 4. The relationship between routine-types.

As the gate-simulation routine of Fig. 3 illustrates, the simulation produced by our compiler is a two-valued simulation. The logic model was chosen to correspond to that used by the oblivious convergence algorithm. However, since adding gateways to this model makes it non-oblivious, there is now a problem with initializing nets to their proper values. For example there may be a NOT gate whose input and output nets are both initialized to zero. One way to guarantee that the nets of a circuit have consistent values is to force every gate to be simulated at least once during the simulation of the first input vector. This can be done easily with three valued logic by initializing all nets to the unknown value. It is also possible to explicitly test for the first vector when comparing net

values. Our solution to the problem is to adopt some features of the three valued model without fully implementing three-valued simulation. Each logic value is represented as two bits. The low-order bit represents the value of the net, while the high-order bit is a flag which indicates whether the simulator has ever computed a new value for the net. The high-order bit is initialized to 1 for all nets. When a new value is computed for a net, the high-order bit is explicitly set to zero. This causes the net comparison to fail for any net for which the simulator has not yet computed a value, which in turn will cause all gates to be simulated at least once during the simulation of the first input vector.

Although the number of instructions generated for each routine depends heavily on the underlying architecture, it is still possible to make an estimate of the number of instructions required to simulate a single gate. In this estimate, we will ignore the work done by the net and gate trailer routines. Fig. 5 contains an estimate of the number of instructions required to simulate a two-input AND gate with a fan-out of two. It is further assumed that three-operand memory-to-memory instructions are available, and that indirect addressing is available for the gateways.

Routine	Operation	Instructions
Gate Simulator	Simulate Gate	1
	Set H. O. B. to zero	1
	Set Flag to zero	1
	AddBlock	2
	Gateway	1
Net Handler	Compare value	2
	Compare flag	2+2
	AddBlock	2+2
	Set Flag to one	1+1
	Gateway	1
Total		19

Fig. 5. An estimate of instruction counts.

As Fig. 5 shows, a two input AND with a fanout of two can be simulated using less than 20 instructions. The majority of the gates in our sample circuits [10] have a fanout of 1, which would reduce the estimated instruction count by 5.

3. Optimizations of the Gateway Technique.

Although the use of gateways permits non-oblivious simulation to be done with a relatively low per-gate instruction count, there are several opportunities for further optimization of the code. The first optimization concerns NOT gates and single-input buffers, as illustrated in Fig. 6.

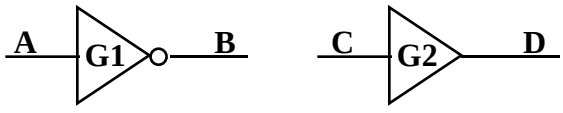

Fig. 6. Two single-input gates.

For the two gates pictured in Fig. 6, if the input to the gate changes, the output is also guaranteed to change after the gate is simulated. Since the two output nets, B and D will have their own processing routines, it is possible to eliminate the first test from each of these routines, thereby reducing the instruction count of these routines by 2. Since the

NOT gate is common (it is the first or second most commonly used gate in virtually all of our test circuits), this optimization can save a significant amount of execution time.

The second optimization concerns the flags that are used to determine whether a gate has been switched into the instruction stream. In many cases, these flags are not required. To see why this is so, consider the gates pictured in Fig. 6. Since these gates have only a single input, there is only one net-processing routine that could possibly switch the gate into the execution stream. This net-processing routine will be executed only once per state, so when this routine is executed, the flag is guaranteed not to be set when the routine is executed, and the flag will never be tested once it has been set. Therefore, for NOT gates and buffers, it is never necessary to set or test the gate flag. Eliminating this code reduces the instruction count of the net processing routine by three, and that of the gate processing routine by one.

Although the application of the second optimization to single-input gates is obvious, the use of the convergence algorithm allows this optimization to be applied to multiple-input gates as well. Recall that if a particular gate appears in several states, then several simulation routines will be generated for it, one for each state in which it appears. If G is a multiple-input gate in state *i*, and only one of the inputs of G appears as an output of a gate in state *i*-1, then the flag for G is not needed *in state i*. Therefore it is possible to omit the flag-handling code in both the net processing routine of state *i*-1 and the gate simulation routine of state *i*. It may be necessary to use the flag of G in other states. Fig. 7 illustrates a circuit for which no gate flags are required.

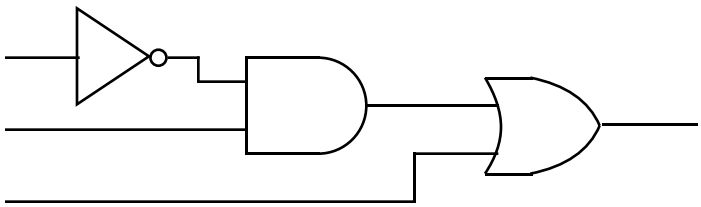

Fig. 7. A flag-free circuit.

The basic technique for eliminating flag tests is as follows. Before processing state *i*, the compiler makes a list, *L* , of all output nets for state *i*-1. (If *i* is equal to 1, then *L* is the set of all primary inputs.) For each gate *G* in state *i*, a count is made of all nets in *L* which are inputs *G*. This count is called the "touch count" of *G* in state *i*. If the touch count of *G* in *i* is one, then no gate flag is required for *G* in state *i*. If the circuit is cyclic and *i* is the first state in the iterative portion of the code, then a slightly different technique must be used to obtain the touch count. Since this state actually has two predecessor states, both predecessor states must be taken into account when computing the touch count. Let *n* be the last state in the iterative section. Two touch counts are computed, one using the outputs of state *i*-1, and one using the outputs of state *n*. The final touch count of a gate is the maximum of the two touch counts.

The third and final optimization is called "clumping." This optimization is based on the observation that, under certain circumstances, certain sets of gates are always switched into the instruction stream as a group. Fig. 8 illustrates this concept.

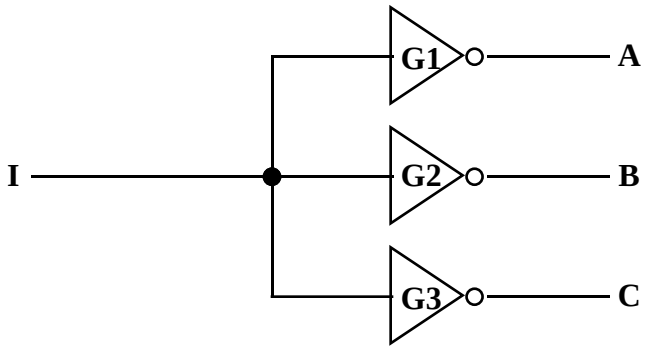

Fig. 8. Gates which can be clumped.

In Fig. 8, whenever the net *I* changes value, all three gates, G1-G3, will be scheduled for simulation. This also implies that whenever the net processing routine for net A is scheduled, the net processing routines for B and C will also be scheduled. Observe that the touch counts for G1, G2, and G3 will be one, regardless of the states in which they occur. This situation is reasonably rare, however, because it is unusual for a net to fan out only to single input gates. Whenever two or more gates must always scheduled together, they can be "clumped" into a single segment of code which is then scheduled as a single unit.

Unfortunately, it is not always easy to determine whether two gates will always be scheduled together. To simplify matters, we have divided clumping into two categories which we call "first order clumping" and "second order clumping." First order clumping is applied only to gates whose touch count is 1. Before the code for state *i* is generated, the list of output nets for state *i*-1 is examined. If a net has a fanout greater than one, and two or more of the gates to which the net fans out have a touch count equal to one, then these gates are clumped. If two or more gates are clumped, their output nets are also clumped. (Wired connections cannot be clumped unless all of the driving gates are members of the same clump.) Second order clumping applies to gates that may have a touch count greater than one. At the present time, our simulator does not perform second order clumping.

The rules for first order clumping can be applied to all states, except the first state of the iterative section. Since the first state of the iterative section has two predecessor states, gates can be clumped only if they can be clumped with respect to both predecessor states. Touch count is unreliable for this state, as Fig. 9 shows.

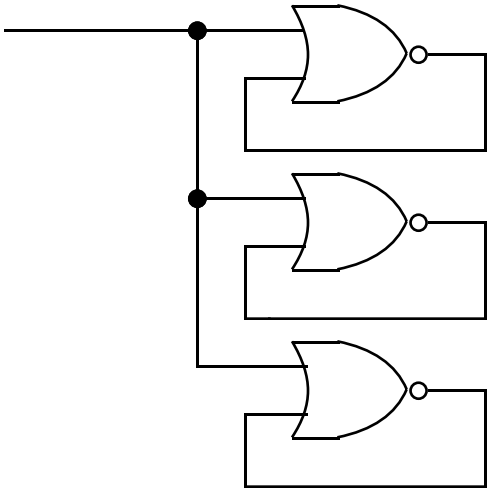

Fig. 9. The unreliability of touch counts.

For the circuit of Fig. 9, the convergence algorithm will produce a single state containing all three gates. Even though the touch count of all gates is one, they cannot be clumped because they must be scheduled independently if there are changes in their outputs. (Although the circuit pictured in Fig. 9 is unrealistic, we have encountered more complex circuits in practice that exhibit the same phenomenon.) Because the first state of the iterative section has two predecessors, only second order clumping can be done for this state. However, it is possible to alter the state sequence generated by the convergence algorithm so that all states, including the first state of the iterative section, can be clumped using first order techniques. Fig. 10 illustrates how this alteration is performed.

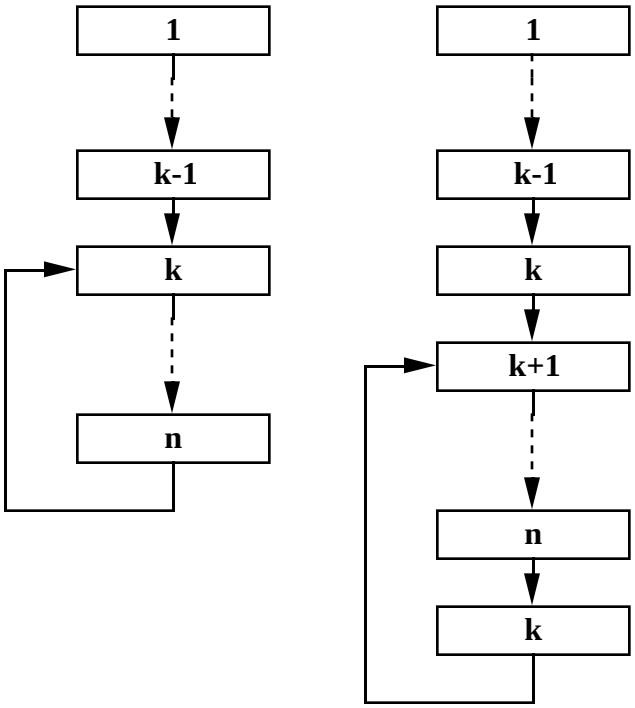

Fig. 10. Alteration of the state sequence for clumping.

In Fig. 10, the first state of the iterative section is copied and placed at the end of the iterative section. The original state is then moved out of the iterative section into the prefix section, and the subsequent state becomes the new first state of the iterative section. The sequence of states is unaltered, but now the first state of the iterative section has two *identical* predecessors, which makes first order clumping techniques applicable.

Fig. 11 contains the results of several experiments that were run to compare the gateway technique to interpreted event-driven simulation.

The Convergence Algorithm							
		Expanded Flip-Flops			Unexpanded Flip-Flops		
Ckt	Interp.	Unoptim	Tst-Elim	Clump	Unoptim	Tst-Elim	Clump
s27	2.0	0.9	0.8	0.6	0.4	0.4	0.3
s208	6.9	2.8	2.3	2.0	1.7	1.5	1.2
s298	10.2	5.9	4.7	4.8	2.3	2.1	1.6
s344	16.6	11.1	10.8	7.4	6.4	4.6	3.8
s349	16.3	11.2	10.8	7.5	5.9	4.5	3.8
s382	13.1	9.0	8.2	5.2	3.2	3.3	2.1
s386	15.4	6.0	5.1	4.3	4.6	3.9	3.0
s420	12.8	6.0	5.6	3.4	4.1	2.7	2.3
s444	14.7	10.5	7.1	5.5	4.0	3.5	2.2
s510	9.5	8.5	8.2	7.1	5.6	5.5	4.0
s526	15.3	9.4	8.6	6.9	3.9	3.4	2.5
s526n	15.2	9.3	8.7	6.8	3.8	3.4	2.4
s641	31.9	19.1	15.6	10.9	**	**	**
s713	34.4	21.0	16.8	12.6	**	**	**
s820	29.3	12.4	11.1	9.2	10.2	9.3	7.3
s832	29.6	12.3	11.1	9.5	10.4	9.1	7.5
s838	24.8	15.5	12.1	9.2	7.9	6.7	5.0
s953	36.8	21.9	16.8	12.0	11.8	8.9	6.5
Max Improv. (%)		59	67	70	80	80	85
Min Improv. (%)		11	14	25	41	42	58
Avg Improv. (%)		42	49	61	69	73	80

Fig. 11. Experimental results for the gateway technique.

The numbers in Fig. 11 are in CPU seconds of execution time. The test circuits are taken from the ISCAS-89 sequential circuit benchmarks [10]. All experiments were run on a SUN-4 IPC with 12 megabytes of memory and a dedicated disk. This system was dedicated to these experiments while they were being run, but could not be completely isolated from all network interaction. The times reported do not include the time required to read or print test vectors. The procedure for obtaining these numbers was to run one set of experiments with the output routines commented out, and a second set of experiments with everything but the input routines commented out. The reported times are the differences between these two sets of experiments. CPU times were obtained using the UNIX /bin/time command. To minimize error in this command, each experiment was run five times and the results were averaged (with truncation after the division) to obtain the reported times.

The times reported in the "interpreted" column were obtained from a locally developed interpretive event-driven simulator. We do not claim that our simulators are optimal, but we believe that all implementations are comparable in quality. The interpreted simulator used a three-valued logic model. The enhanced two-valued logic model used in the gateway technique is described above. For the gateway technique, two different methods for simulating flip-flops were used. For the first set of experiments, flip-flops were expanded as recommended in [10] using the implementation illustrated in Fig. 12. (Note that the ISCAS-89 benchmarks contain only single-output D flip-flops.) In both sets of experiments, flip-flops were supplied with a clock as recommended in [10]. The clock appears as a primary input to each of the test circuits.

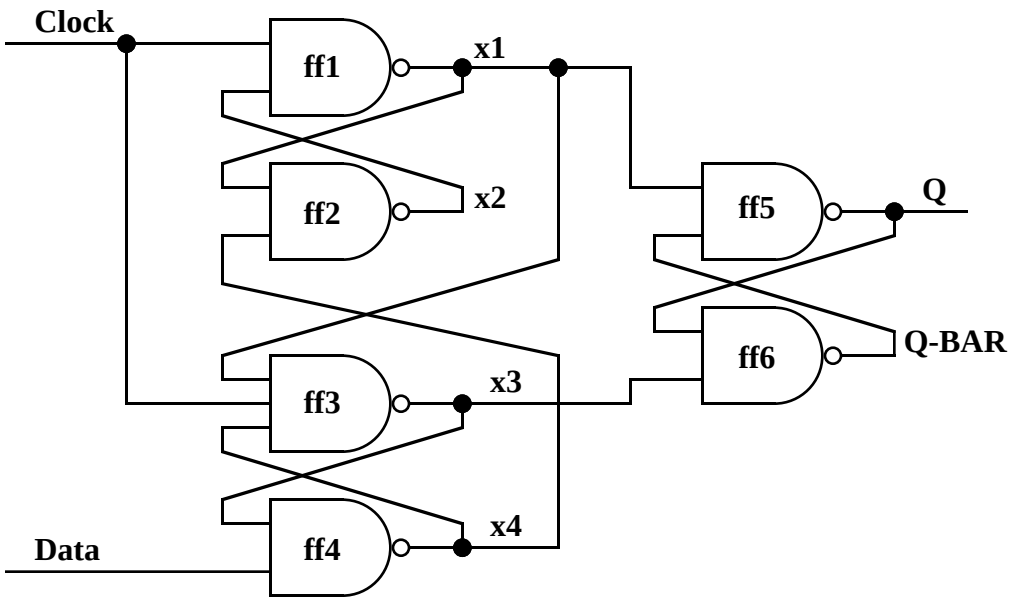

Fig. 12. The expansion model for D flip-flops.

In the second set of experiments, and in the interpreted simulations, flip-flops were simulated as single gates. This changes the total delay of the flip-flop, and also has a profound effect on the convergence sequence of some circuits. In particular, the convergence sequences for s641 and s713 were so long that the generated code was too large to be compiled on our test system. (Hence the ** entries in Fig. 11.) For both methods of handling flip-flops, three sets of experiments were run. The first is the "pure" convergence method without enhancements. The second set includes the two optimizations that remove output tests for NOT gates and buffers, and the flag tests for gates with a touch count of one. The times for these experiments are reported in the "Tst-Elim" columns of Fig. 11. The final set of experiments adds first-order clumping to the test-elimination optimizations.

The improvement percentages were computed using the formula (I - G)/I, where I is the time for interpreted simulation, and G is the time for the gateway technique. An improvement percentage of 80% implies that the gateway technique runs in one fifth the time required for the interpreted simulation. The best results were obtained for full optimization with flip-flops modeled as single gates, which gave an average performance improvement of 80%. Unfortunately, the size of the generated code for this algorithm makes it applicable only to certain circuits. Because of the unique nature of the convergence algorithm, the complexity of the circuit has more effect than the size of the circuit. The critical factor is the least common multiple of the lengths of the various cycles in the circuit. The experiments for the two circuits s641 and s713 demonstrate this point, since treating flip-flops as single gates reduces the size of the circuit, but tends to increase the length of the longest cycle.

4. Reducing Code Size.

As noted in the previous section, the convergence algorithm can generate an excessive amount of code for some circuits. There are two reasons why it is desirable to reduce the amount of generated code, even for those circuits that were simulated successfully. First, reducing the amount of generated code will reduce compilation time, and second, reducing the amount of generated code will tend to enhance simulation performance on machines

with caches, as long as locality can be increased at the same time. Although the problem of compilation time can be eliminated by generating object code directly, cache performance is still an important issue, particularly because many of today's high performance workstations derive a significant amount of their performance from a cache.

The basic technique that we have investigated for reducing code size is collapsing several states into a single state. For example, all of the states in the iterative section can be collapsed into a single state. To collapse two states, one simply computes the union of the sets of gates that comprise the original states. The new state must now be executed twice in succession to obtain the same effect as the original state. Without the addition of gateways, collapsing states can significantly impair the performance of the convergence algorithm. (The correctness of the simulation is not affected, since the inputs of the extra gates will not have changed.) However, with the addition of gateways, the only detrimental effect is the loss of some of the optimizations. Since a collapsed state has two or more predecessor states, first-order clumping is not possible for such states. Furthermore, since the touch count must be computed as the maximum of the touch counts generated by all predecessor states, fewer gate-flags can be eliminated from the gate simulations.

The worst case for collapsing states is to collapse all states into a single state. In this case each gate will have a single simulation routine, and each net will have a single net-handler. First-order clumping can no longer be done for any part of the circuit, and the elimination of gate flags is possible only for single input gates and for those two-input gates whose input nodes consist of one primary input and one internal node (for an example, see Fig. 9). We have performed several experiments to measure the performance of the single-state algorithm. The results of these experiments are reported in Fig. 13.

The Single-State Algorithm					
		Expanded Flip-Flops		Unexpanded Flip-Flops	
Ckt	Interp.	Unoptim	Tst-Elim	Unoptim	Tst-Elim
s27	2.0	0.9	0.8	0.4	0.4
s208	6.9	2.1	2.0	1.2	1.0
s298	10.2	5.1	3.4	1.7	1.5
s344	16.6	6.9	6.3	2.9	2.6
s349	16.3	7.0	6.2	2.8	2.5
s382	13.1	8.4	8.6	2.2	2.0
s386	15.4	4.2	2.9	2.2	2.1
s420	12.8	6.7	5.0	3.3	1.8
s444	14.7	9.1	8.4	3.9	2.0
s510	9.5	5.7	4.9	4.1	3.7
s526	15.3	8.5	9.0	3.3	3.7
s526n	15.2	8.4	9.0	3.3	3.6
s641	31.9	15.9	12.7	9.3	6.7
s713	34.4	16.3	13.4	10.4	8.4
s820	29.3	8.8	8.5	7.4	7.3
s832	29.6	8.8	8.8	7.7	7.1
s838	24.8	14.1	13.4	6.6	5.8
s953	36.8	16.3	15.8	9.1	7.0
Max Improv. (%)		73	81	86	86
Min Improv. (%)		36	34	57	61
Avg Improv. (%)		53	58	77	80

Fig. 13. The performance of the single-state algorithm.

The numbers reported in Fig. 13 were obtained in the same manner as those reported in Fig. 11. The same equipment was used for all experiments. The effect of locality on cache performance can be gauged by examining the "Unoptim" columns of both figures. It is important to note that in both sets of experiments *precisely* the same sequence instructions was executed for the unoptimized versions of the algorithms. The differences in the "Unoptim" column between Figs. 11 and 13 is due *entirely* to the increased locality of the single-state algorithm. On systems with no cache and sufficient main memory to prevent paging, we would expect identical performance out of both unoptimized algorithms.

The increase in locality raises the average performance improvement from 42 and 69% to 53 and 77% for the two methods of simulating flip-flops. With test elimination, the increase in locality provides performance improvements that rival those of the convergence algorithm with full optimization.

5. Conclusion.

The gateway technique has proven to be an effective technique for adding event-driven behavior to unit-delay simulations. For the circuits tested, an average performance improvement of 80% was observed. This implies that the techniques used will allow simulations to complete in one fifth the time required for interpreted event-driven simulation. These results come close to the performance improvements observed for the parallel technique of unit-delay compiled simulation [3], provides a performance improvement of around 90%. Although the gateway technique cannot match this performance, the parallel technique is applicable only to acyclic circuits and synchronous cyclic circuits. The algorithms studied here are applicable to all circuits, including asynchronous cyclic circuits.

One important point that is demonstrated by our comparison of the convergence algorithm and the single-state algorithm, is that cache effects will play an important role in the future of compiled simulation. Traditionally compiled simulation techniques, such as Levelized Compiled Code simulation, rely, to a certain extent, on loop unrolling for their performance. Unfortunately, loop-unrolling can have a detrimental effect on performance when a cache is used. If unrolling loops makes the repetitive portion of the code too large to fit in the cache, performance can degrade rather than improve. (We have performed some basic experiments with unrolled loops on the SUN-4 IPC that prove this point.) In the future, compiled simulators will not be able to blindly ignore locality issues when generating code. Techniques such as the gateway technique can be useful in increasing locality, if used properly.

Although our studies were confined to two different algorithms, we believe that the gateway technique can be used to improve the performance of a number of different algorithms. We are currently investigating the possibility of using gateways with other simulation algorithms, such as the parallel technique. We also believe that the gateway technique can be used with other timing models such as the zero-delay and multi-delay models. We believe that there is a considerable amount of work that must be done to determine the applicability of the gateway technique to various different types of simulation. Regardless of the outcome of this research, it is clear that the gateway technique is an effective means of improving the performance of certain types of compiled unit-delay simulation.

REFERENCES

1. R. E. Bryant, D. Beatty, K. Brace, K. Cho and T. Sheffler, "COSMOS: A Compiled Simulator for MOS Circuits," *Proceedings of the 24th Design Automation Conference*, 1987, pp. 9-16.
2. D. M. Lewis, " Hierarchical Compiled Event-Driven Logic Simulation," *Proceedings of ICCAD-89*, pp.498-501.
3. P. M. Maurer and Z. Wang, " Techniques for unit-delay compiled simulation", *Proceedings of the 27th Design Automation Conference*, 1990, pp. 480-484.
4. P. Maurer, "Optimization of the Parallel Technique for Unit-Delay Compiled Simulation," *Proceedings of ICCAD-90*, pp. 70-73.
5. Z. Wang and P. M. Maurer, " LECSIM : A Levelized Event Driven Compiled Logic Simulator", *Proceedings of the 27th Design Automation Conference*, 1990, pp. 491-496.
6. Z. Barzilai, J. L. Carter, B. K. Rosen and J. D. Rutledge, "HSS -- A High Speed Simulator," *IEEE Transactions on Computer-Aided Design*, Vol. CAD-6, No. 4. July 1987, pp. 601-617.
7. M. Chiang, and R. Palkovic, "LCC Simulators Speed Development of Synchronous Hardware," *Computer Design*, Mar. 1, 1986, pp. 87-91.
8. L. Wang, N. Hoover, E. Porter and J. Zasio, "SSIM: A Software Levelized Compiled-Code Simulator," *Proceedings of the 24th Design Automation Conference*, 1984, pp. 473-478.
9. C. Hansen, "Hardware Logic Simulation by Compilation," *Proceedings of the 25th Design Automation Conference*, 1988, pp. 712-715
10. F. Brglez, D. Bryan, K. Kozminski, "Combinational Profiles of Sequential Benchmark Circuits," *Proceedings of ISCAS-89*, pp. 1929-1934.