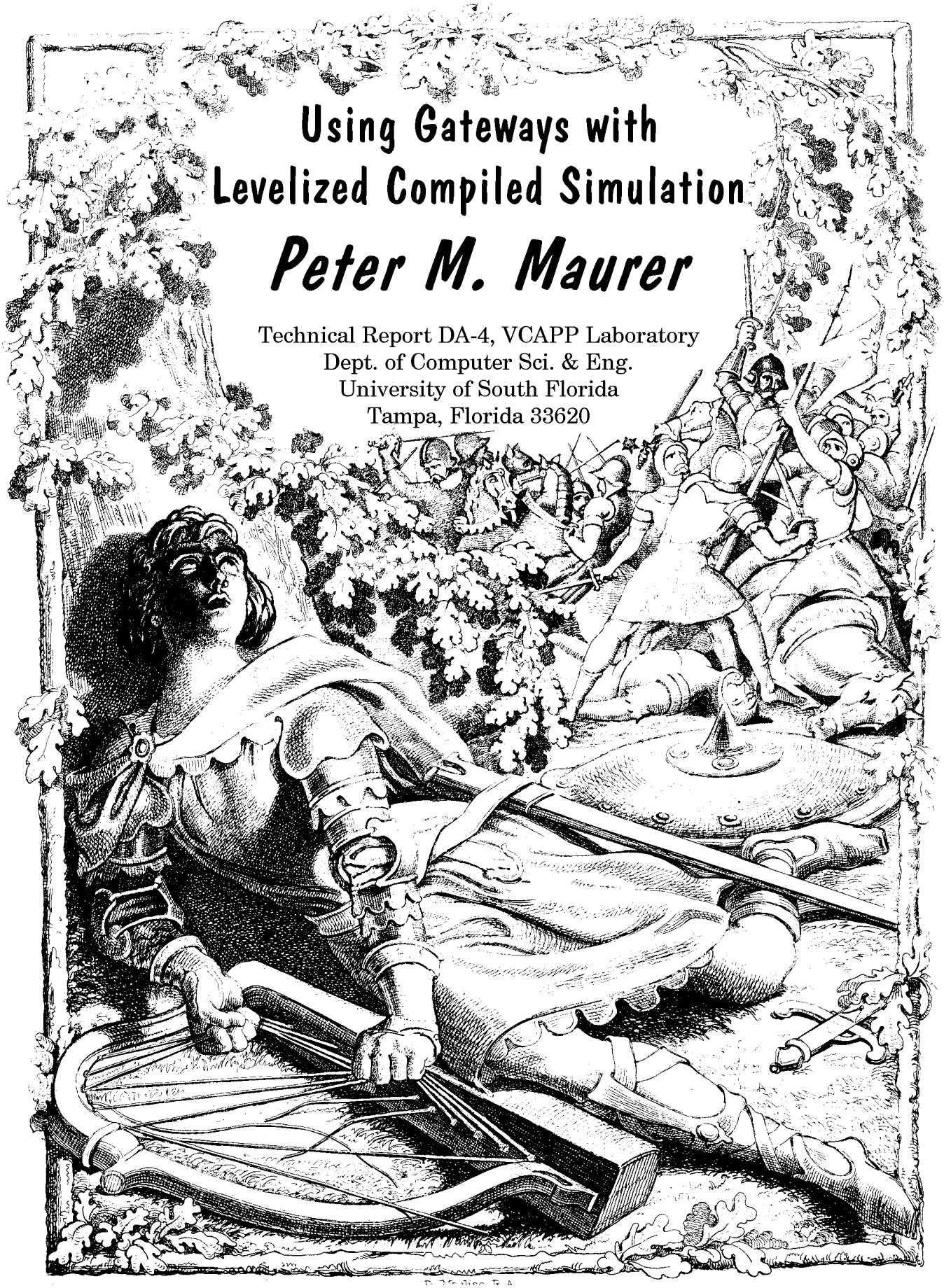
Using Gateways with Levelized Compiled Simulation

Peter M. Maurer

Technical Report DA-4, VCAPP Laboratory
Dept. of Computer Sci. & Eng.
University of South Florida
Tampa, Florida 33620



USING GATEWAYS WITH LEVELIZED COMPILED SIMULATION

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

ABSTRACT

Although Levelized Compiled Code simulation performs well under moderate to high activity conditions, there are many circuits that exhibit extremely low activity rates, either overall, or in certain subcircuits. The techniques presented here allow event-driven behavior to be added to Levelized Compiled Code simulations, with the aim of improving the performance of circuits with very low activity. Two techniques are presented, one which can be applied selectively on a block-by-block basis, and one that can be applied selectively on a gate-by-gate basis. Both concepts are based on the concept of a gateway, or retargetable branch. Performance results are presented for both techniques.

USING GATEWAYS WITH LEVELIZED COMPILED SIMULATION

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

1. Introduction.

Compiled simulation has received a great deal of attention lately from several different researchers [1-11]. Most of this research has focused on the unit-delay timing model, and many of the techniques are oblivious, that is, they simulate a fixed number of gates per input vector rather than performing dynamic tests to determine whether certain gate simulations are required. Of all compiled simulation techniques that are currently known, the highest performance is generally reported for the Levelized Compiled Code (LCC) technique [9], which is oblivious and based on the zero-delay timing model. The purpose of this paper is to explore two techniques for improving the speed of levelized compiled code simulations when the number of gates that need to be simulated per input vector is extremely low. The underlying mechanism of both techniques is the "gateway" or retargetable branch that has been employed in several other simulators [2,5,6,7]. The fundamental idea behind the gateway is that it can be used to switch segments of code into and out of the instruction stream. Gateways can be used in many different ways to add event-driven behavior to oblivious compiled code simulations. In [5], gateways are used to create a chain of executable blocks, each one of which is chained to the next using retargetable branch instructions. In [2], retargetable branches are used to go to the top address in a stack of dynamic branch addresses.

The activity rate of a circuit is the percentage of gates whose inputs change value during the simulation. Although LCC simulation provides the fastest form of simulation available [9] when the activity rate of a circuit is moderate to high, when the activity rate drops below some critical level (usually around 1-2%), interpreted unit-delay simulation begins to outperform LCC simulation. There are many circuits for which one would expect the activity rate to be extremely low. For example, certain parts of the data path of a microprocessor may be used only for certain uncommon instructions. There may be other circuits which exhibit a high activity in certain subsections, even though the over-all activity rate is very low. For example, the control circuitry of a microprocessor may exhibit an extremely high activity rate, even though the activity rate for the data path is low. Using gateways it is possible to add event-driven behavior to an LCC simulation, either on a block by block basis, or on a gate-by-gate basis.

In this work, gateways are used in two different ways. The first method uses a chain of executable blocks similar to that described in [5], but because the zero-delay timing model of the original simulation must be preserved, a scheduling method similar to that described in [7] must be used. This method lends itself well to adding event-driven behavior on a block-by-block basis, but not on a gate-by-gate basis. The second method uses gateways as shunts to skip the execution of unnecessary parts of the code. In this case, code is generated in levelized order, which preserves the zero delay model of the original simulation. Because the shunt method has very little impact on the structure of the original compiled simulation, it can be used either on a block-by-block basis or a gate-by-gate basis. On the other hand, shunts can cause more overhead than the levelized queues

algorithm, because all retargetable branches must be executed, regardless of the previous input activity.

2. Levelized Queues.

The first step in generating a levelized queue simulator is to assign a level number to every gate and net in the circuit. Primary inputs and constant signals are assigned a level number of zero. Once the inputs of a gate have been assigned level numbers, the level number of the gate is computed by finding the maximum of the level numbers of the inputs of the gate, and incrementing it by 1. The outputs of a gate are assigned the same level number as the gate. If there are wired connections, the level of a net is the maximum of the levels of all gates which drive the net.

During the simulation, several queues are maintained, one for each different level of the circuit. Each gate is permanently assigned to the queue that corresponds to its level, but the gate may or may not be present in the queue during the simulation. The elements of a queue are simulation routines, each of which is terminated by a gateway as illustrated in Figure 1. These routines are linked using the gateway address, so simulation routines will be executed sequentially in the order in which they are queued.

One simulation routine is generated for each gate in the circuit, but because levelized zero-delay simulation does not require net-processing routines, no net processing routines are generated. Each gate simulation routine has the form illustrated in Figure 1.

```
G1Address:
/* processing for gate G1 */
newA = oldB & oldC;
G1Flag = 0;
if (newA != oldB)
{
        oldA = oldB;
        Schedule gates in the fanout
        list of G1;
}
goto *G1Target;
```

Figure 1. A Gate Simulation Routine.

The central component of the generated code is the queue header array, which contains the address of the first block of code in each queue. When the simulation of an input vector commences, all queues are set to empty. The usual method of setting a queue to empty is to assign NULL to the queue header, but this would require the services of a central scheduler to examine the queue headers for the NULL pointer and branch when necessary. Furthermore since the execution of a simulation routine does not alter the queue header, some additional mechanism must be used to determine whether all blocks in the queue have been executed. To solve both of these problems, a queue is set to empty by initializing the corresponding queue header with a pointer to a queue termination routine, as is done in the tortle_c simulator [2].

Regardless of the state of the queue, the queue termination routine is always the last routine executed for a particular queue. This routine first stores its own address in the queue header, thereby emptying the queue, and then branches to the address contained in the following queue header. The queue termination routine is illustrated in Figure 2. The queue termination routine for the last queue in the list terminates the simulation of a vector rather than advancing to the next queue.

```
QueueTerm:
/* Queue Termination Routine */
Queue_Header[current_queue] = &QueueTerm;
current_queue++;
goto *Queue_Header[current_queue];
```

Figure 2. The Queue Termination Routine.

The queues described in [7] are first-in-first-out queues, which require two pointers, one for the head and one for the tail. Since within a particular level, there is no restriction on the order that gates must be simulated, we elected instead to use last-in-first-out queues, which require only a header pointer. (Last in first out queues are also used in [2].) The code required to add a simulation routine to a queue is illustrated in Figure 3. The simulation routine for a particular gate is always placed in its assigned queue, regardless of when the queueing operation takes place. Since the assigned queues are known at compile time, the queue assignment is hardwired into the generated code.

```
if (G1Flag == 0)
{
    G1Flag = 1;
    G1Target = Queue_Header[G1Level];
    Queue_Header[G1Level] = &G1Address;
}
```

Figure 3. The Code for Queueing a Gate.

The variable "G1Flag" which appears in Figures 1 and 3 is used to prevent a particular simulation routine from being queued twice. If a routine is queued twice, a cycle will be created in the queue, which will cause the simulator to run forever. Any time a gate has more than one input, there is a potential for queueing the gate more than once.

Although our simulator is designed to simulate only combinational circuits, the modifications for handling cyclic circuits are straightforward, and are virtually identical to those described in [7]. The performance of the levelized queues simulator is described in Section 4.

3. Shunts.

Although the levelized queue technique can be used selectively on a block-by-block basis, it is difficult to apply it on a gate-by-gate basis. Shunts are designed to be used on a gate-by-gate basis, although the overhead of this technique can be larger under extremely low activity conditions. On the other hand, because of its simplicity, the shunt algorithm generally has less overhead under high activity conditions than the levelized queue technique. Figure 4 illustrates a gate simulation with shunt code added.

```
G1Begin:
/* a gate simulation routine */
goto *G1Target;
G1Target = &G1End;
NewA = OldB & OldC;
if (NewA != OldA)
{
      OldA = NewA;
      G2Target = &G2Begin;
      …
}
G1End:
```

Figure 4. Shunt Code.

One simulation routine such as that illustrated in Figure 4 is generated for each gate, and these routines are generated in levelized order, just as in LCC simulation. The advantages of using shunts are that a gate simulation can be scheduled by performing a simple assignment, and that no flags are needed to prevent a gate from being scheduled twice. The disadvantage is that the retargetable branch that precedes every simulation must always be executed, even if none of the primary inputs of the circuit have changed. To minimize the number of branches that must be executed when there is no activity in the circuit, we have investigated a technique called *double shunts*. When using double shunts, the circuit is partitioned in some fashion, and a second shunt is inserted in front of every partition. This shunt will bypass all code for the partition if no input net to the partition changes.

It is not yet clear what form of partitioning would be most effective for double shunting. One possibility would be to use fan-out free blocks as was done in [7], but for simplicity in proving the concept, we elected instead to partition the circuit by levels. Each level in the circuit was supplied with a shunt that will bypass all simulation for the level if no input to the level has changed state. This method of partitioning is not optimal, because if a change is propagated from the primary inputs at level 0 to a primary output at level n, the shunts of all gates on levels 1 through n must be executed. For more optimum performance with moderate activity rates, some method of vertical partitioning, such as fanout-free blocks, should be used. The existing scheme performs well with consecutive identical input vectors, as is shown in the following section. This is important, because in many realistic circuits, the extremely low activity rates observed for some portions of the circuit can usually be attributed to sets of consecutive identical vectors.

Double shunting has the advantage that fewer retargetable branches must be executed when there have been no changes in the primary inputs, but it has the disadvantage of doubling the number of operations required to schedule a gate. When a gate is scheduled, both the shunt for the gate and the shunt for the gate's partition must be retargeted. The experimental data presented in Section 4 illustrates these points.

4. Experimental Data.

Figure 5 contains the experimental results for the levelized queues algorithm. All experiments were run on a SUN 4/IPC with 12 megabytes of memory and a dedicated disk drive. This system was dedicated to running these experiments, but could not be completely isolated. The ISCAS85 benchmark circuits were used as the basis of all experiments [12]. These circuits have been used by several researchers to measure the performance of their simulators. Each circuit was run on 5000 randomly generated vectors. Times were obtained using the UNIX /bin/time command. To minimize the errors in this command, each experiment was run five times and the results were averaged. The

numbers reported are CPU seconds of execution time, and do not include the time to read and print vectors. For purposes of comparison, Figure 5 also contains data for oblivious zero-delay simulation and for interpreted unit delay simulation. The improvement percentages in all figures are the percentage improvement for a particular algorithm over interpreted unit-delay simulation, using the formula ((interp_time - compiled_time) / interp_time)*100.

Levelized Queues			
Ckt	Interpreted Unit-Delay	Oblivious Zero-Delay	Gateways Zero-Delay
c432	23.4	0.5	1.7
c499	26.1	0.6	2.9
c880	46.3	1.2	7.9
c1355	93.8	1.9	12.8
c1908	172.9	4.4	24.7
c2670	192.1	5.3	40.0
c3540	277.1	8.4	46.8
c5315	519.1	21.7	83.1
c6288	5108.6	30.1	81.6
c7552	795.1	40.7	118.4
Max Improv. %		99	98
Min Improv. %		95	79
Avg Improv. %		97	87

Figure 5. The Levelized Queue Results.

As can be seen from Figure 5, oblivious zero-delay simulation outperforms the levelized queue algorithm by at least a factor of 2 on our random input vectors. However, the activity rate for these vectors is quite high, averaging 58.7% overall. The execution time of the levelized queues algorithm is proportional to the activity rate, while the execution time for oblivious zero-delay simulation is fixed. Therefore when the activity rate drops below a certain threshold, the levelized queue algorithm will begin to outperform the oblivious algorithm. For the circuits we tested, the threshold was around 13% on the average. The threshold ranged from a low of 7% to a high of 22%.

Figure 6 gives the results for single and double shunts. As for Figure 5, the results are reported in seconds of CPU execution time. For these experiments, shunts were added to the oblivious simulation in three different ways. In the first set of experiments, shunts were added to all gates. In the second set, shunts were added to all gates except those at level 1. In the third set, shunts were added only to those gates whose level number was more than half of the circuit depth. The results of the second and third sets of experiments are reported in the columns labeled "After 1" and "Halfway" respectively.

Shunts						
Ckt	Single Shunts			Double Shunts		
	Full	After 1	Halfway	Full	After 1	Halfway
c432	1.4	0.9	0.8	1.7	1.0	0.9
c499	1.7	1.4	1.0	2.0	1.5	1.1
c880	4.3	2.7	1.6	5.7	2.7	1.7
c1355	8.0	4.6	3.2	10.8	4.5	2.9
c1908	19.1	17.2	6.0	22.5	20.4	7.1
c2670	32.8	24.1	5.0	37.7	28.3	7.7
c3540	39.5	32.3	13.2	45.1	36.4	14.6
c5315	66.5	56.4	22.0	76.7	63.0	22.5
c6288	67.2	62.3	36.1	76.4	71.7	40.8
c7552	96.9	82.1	40.3	109.8	93.4	42.2
Max	99	99	99	98	99	99
Min	83	87	95	80	85	95
Avg	90	92	97	88	91	96

Figure 6. The Results for Shunts.

A comparison of Figures 5 and 6 shows that shunts outperform the levelized queues algorithm by a significant amount for the input vectors we tested. Although the best performance is exhibited by the "Halfway" algorithms, it is important to remember that the running times of the shunt simulators are not proportional to the activity rate. In the case of the double shunt algorithm, it is difficult to determine the relationship between the activity rate and the performance of the algorithm without more specific knowledge about the activity rate with respect to the circuit partitions. For the "After 1" and "Halfway" algorithms, the overall activity rate is not as significant as the activity rate with respect to the shunted portion of the simulation.

To gain some appreciation of how activity rate affects the performance of the shunt algorithms, we ran a set of "zero-activity" experiments, which were run using a set of 5000 zero-vectors. Although some simulation must be done for the first input vector to put the circuit into a consistent state, no simulation will be done after the first vector has been processed. These vectors allow us to determine the minimum amount of work that must be done to simulate a set of 5000 vectors, and the maximum amount of time that can be saved due to event driven processing. The results of these experiments are reported in Figure 7.

Baseline							
	Levelized	Single Shunts			Double Shunts		
Ckt	Queues	Full	After 1	Halfway	Full	After 1	Halfway
c432	0.1	0.4	0.3	0.5	0.0	0.0	0.4
c499	0.0	0.4	0.5	0.4	0.0	0.3	0.3
c880	0.3	1.1	1.3	1.2	0.3	0.2	1.3
c1355	0.2	2.1	2.0	2.2	0.1	0.4	0.9
c1908	0.2	3.3	3.6	4.3	0.2	0.5	2.4
c2670	1.2	8.9	7.3	5.2	1.2	1.0	7.3
c3540	0.2	10.1	10.4	9.9	0.2	0.8	6.2
c5315	0.9	17.3	16.0	18.9	1.0	1.3	17.1
c6288	0.6	16.3	17.6	20.5	0.5	2.3	12.2
c7552	1.0	28.8	26.0	35.1	0.8	0.6	32.1

Figure 7. The Zero-Activity Results.

The results of Figure 7 shed new light on the differences between the various algorithms presented in this paper. Although the Single-Shunt Halfway algorithm provides performance rivaling that of oblivious LCC simulation, only a small fraction of that time can be saved through event-driven techniques. For levelized queues, all but a small fraction of the simulation time can be saved. When interpreting the results for double shunts, it is important to remember that the running time of the algorithm is not proportional to the activity rate of the circuit, with the highest increase in execution time coming at the lower activity levels. As mentioned in Section 3 above, dividing the circuit on some basis other than levels would probably result in a better correlation between execution time and activity rate.

5. Conclusions.

The techniques presented here can be used to improve the performance of LCC simulation when the activity rate is below 1-3%. The levelized queue technique provides performance that is proportional to the activity rate, but provides lower performance than the single-shunt algorithm under high-activity conditions. Under low activity conditions, the single-shunt method has a relatively high baseline performance that may limit its usefulness as an over-all technique. On the other hand, the single-shunt method can be applied to the circuit on a gate-by-gate basis, which allows it to be used selectively to those parts of the circuit that are seldom simulated. There is also no reason why the single-shunt method could not be applied to units larger than a single gate.

The double shunt method provides excellent performance under zero input activity conditions. This is important because extremely low activity rates can usually be attributed at least in part to consecutive identical vectors. Other partitioning methods exist that could improve the overall performance of the double shunt technique, but it is important to keep in mind that at high to moderate activity rates, the double shunt method does not perform as well as the single shunt method. More research on effective partitioning methods for the double shunt technique is needed.

These techniques should prove to be an effective means for speeding up the performance of low-activity LCC simulations.

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, "A Hierarchical Compiled Code Event-Driven Logic Simulator," IEEE Transactions on CAD, Vol 10, No. 6, June 1991, pp. 726-737.
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. P. Maurer, "Gateways: A Technique for Adding Event-Driven Behavior To Compiled Unit-Delay Simulations," Submitted for publication.
6. P. Maurer, "The Shadow Algorithm: A Scheduling Technique for Both Compiled and Interpreted Simulation," submitted for publication.
7. 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.
8. 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.
9. M. Chiang, and R. Palkovic, "LCC Simulators Speed Development of Synchronous Hardware," *Computer Design*, Mar. 1, 1986, pp. 87-91.
10. 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.
11. C. Hansen, "Hardware Logic Simulation by Compilation," *Proceedings of the 25th Design Automation Conference*, 1988, pp. 712-715
12. F. Brglez, D. Bryan, K. Kozminski, "Combinational Profiles of Sequential Benchmark Circuits," *Proceedings of ISCAS-89*, pp. 1929-1934.