LECSIM: A LEVELIZED EVENT DRIVEN COMPILED LOGIC SIMULATOR*

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

Abstract

LECSIM is a highly efficient logic simulator which integrates the advantages of event driven interpretive simulation and levelized compiled simulation. Two techniques contribute to the high efficiency. First it employs the zero-delay simulation model with levelized event scheduling to eliminate most unnecessary evaluations. Second, it compiles the central event scheduler into simple local scheduling segments which reduces the overhead of event scheduling. Experimental results show that LECSIM runs about 8-77 time faster than traditional unit-delay event-driven interpretive simulator. LECSIM also provides the option of scheduling with respect to individual gates or with respect to fan-out free blocks. When the circuit is partitioned into fan-out free blocks, the speed increases by a factor of 2-3. With partitioning, the speed of LECSIM is only about 1.5-3.4 times slower than a levelized compiled simulation for the combinational circuits we have tested.

1. Introduction

The event driven simulation technique[1] has been used for many years to implement different types of simulators. The great success of this algorithm stems from the elegance of the selective trace approach (i.e. evaluating only the active components), together with its ability to easily handle asynchronous designs and timing analysis. Though much effort has been made in the past two decades to improve the speed of event driven simulation[2,3], efficiency is still a major problem. Three factors contribute to the inefficiency of the algorithm. First, not all the events produced by the evaluation of active components are necessary to produce useful output. In unit-delay simulation these events are useful for detecting hazards and race conditions, but for today's highly complex synchronous circuits it is usually simpler to test the functional behavior of the circuit before performing hazard analysis. Zero-delay simulation is usually adequate for high-level functional testing. Our experiments have shown that unit-delay event driven simulators can generate as many as 26 times more events than necessary for certain types of circuits. These false events seriously impair the performance of the simulator. Second, the centralized event scheduler often introduces an enormous amount of overhead, which is particularly true when the primitive components are simple and only require a few instructions to evaluate. A primitive gate, for example, needs only two or three instructions for evaluation, but it may take hundreds instructions to schedule its evaluation. Third, almost all event-driven simulators are interpretive and can not use the optimization techniques of the compilation process.

While the traditional event driven algorithm continues to improve[4,5], many researchers have tried to improve the efficiency by using different methodologies. The demand driven algorithm employed in BACKSIM[6] is such attempt. By assigning a time window to each value encountered during backward traversal, demand driven simulation evaluates the components only when their values are needed to provide simulator outputs, and at those simulation time steps where they are valid. While the demand driven algorithm improves efficiency by eliminating most unnecessary evaluations, the recursive back tracking routine employed in demand driven algorithm incurs a severe penalty, particularly when the circuit is deep.

The levelized compiled simulation technique takes a totally different approach [7,8]. Instead of translating the circuit description into internal data structures operated on by a separate simulation kernel, compiled simulation translates the circuit description directly into code. The code is arranged by the levels to ensure that whenever a component is evaluated, the correct values of its inputs are available. The simulation is performed by sequentially executing the code, and each component is evaluated exactly once for every input vector. Since this approach eliminates the need for event management, it is extremely efficient. There are, however, problems with this approach that restrict its usefulness. Levelized compiled simulation in general lacks the ability to handle asynchronous circuits which tends to limit its application to combinational and synchronous circuits. Furthermore, the strict sequential execution of this approach makes it difficult to perform timing analysis. An interesting point we would like to mention here is that the "evaluate everything" nature of levelized compiled simulation is generally considered a drawback. However, as it is pointed out in [11] and confirmed by our experimental results, levelized compiled simulation is inferior to event driven interpretive simulation only when the circuit's activity is lower than 1%, a situation which rarely occurs in practice.

Although levelized scheduling has traditionally implied a compiled implementation while event-driven scheduling has implied an interpretive implementation, researchers have recently begun to recognize that these concepts are independent and that there are advantages to various non-traditional combinations. Some of the possible combinations are illustrated in Figure 1.

The switch level simulators COSMOS [9] and SLS [10] have explored the combination of compiled implementation and event-driven scheduling. Both simulators gain high performance by compiling the circuit into code which is then manipulated by a central scheduler during simulation. However, they retain the traditional concept of a centralized event scheduler. HSS/4 [11] is the first compiled fault simulator which incorporates event-driven concept. In addition to generating the code for block evaluations, it also generates

* This work was supported in part by the Defense Advanced Research Projects Agency under grant 2114-033-LO and by the University of South Florida Center for Microelectronics.

code that controls the activation of successor block trees for later evaluations. Tortle_c [12] provides another efficient implementation of fully compiled event driven simulation, together with a hierarchical subcircuit feature to allow incremental compilation.

Scheduling Algorithms	Implementation Techniques
E : Even driven	**I** : Interpretive
L : Levelized	**C** : Compiled

E + I = Event driven Interpretive

L + C = Levelized Compiled

E + C = Compiled Event driven

L + E + C = Levelized Event driven Compiled

Figure 1. The structure of Simulators

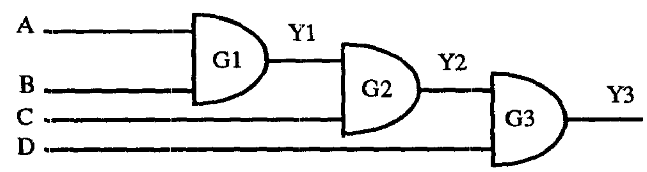
In this paper, we present a new simulator LECSIM, a LEvelized event driven Compiled SIMulator. In addition to combining event driven scheduling with a compiled implementation, LECSIM also employs a network levelization algorithm and zero-delay simulation model to suppress most unnecessary events. Furthermore, LECSIM generates a single piece of code for most Strongly Connected Components. A segment of a circuit is *strongly connected* if it is connected, and the output of every gate in the segment depends on the output of every other gate in the segment. A strongly connected component is a maximal strongly connected segment of a circuit. That is, adding one more gate to the segment would cause it to become *not* strongly connected.

The code generated by LECSIM for a strongly connected component contains its own iteration control mechanism which limits the iteration to a small fragment of code. Consequently, the event scheduling overhead is reduced and the overall scheduling is simplified. These techniques, together with an efficient implementation of the event insertion and dispatch algorithms, gives LECSIM a substantial performance advantage over both interpretive algorithms and compiled event-driven algorithms based on the unit-delay model.

2. The Zero Delay Model and Levelization

Unit delay simulation model is widely used in event driven simulators when accurate timing analysis is not needed. While this model simplifies the process of event scheduling, it often generates many unnecessary evaluations. Although some of these unnecessary evaluations can be used to derive a rough analysis of the hazards and race conditions in a circuit, there are many situations in which this analysis is not required. In such cases the unnecessary evaluations do nothing but add to the overhead of the simulation.

Figure 2 illustrates how unnecessary evaluations occur. Assume the initial states on all the nets in the circuit are 0s and the input vector 1111 is applied to the circuit at time 0. A unit delay event driven simulator will complete the simulation in 3 time steps and 5 evaluations as shown in the table. It is obvious that the evaluations on G2,G3 in time 1 are unnecessary and that the circuit can be simulated using only 3 evaluations. This simple example accounts for the fact that unit-delay event-driven simulators can generate as many as 26 times more events than necessary (in our experiments) for some purely combinational circuits.



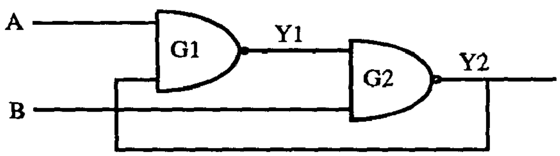
Time	A	B	C	D	Y1	Y2	Y3	Gates to Evaluate
Init.	0	0	0	0	0	0	0	
0	1	1	1	1	0	0	0	G1 G2 G3
1	1	1	1	1	1	0	0	G2
2	1	1	1	1	1	1	0	G3
3	1	1	1	1	1	1	1	

Figure 2. The Source of False Events.

The levelization algorithm [7] in conjunction with zero delay simulation model handles this problem effectively. The levelization process assigns G1 to level 1, G2 to level 2 and G3 to level 3. If the simulation is ordered strictly by level, only 3 evaluations are needed to obtain the correct result. Though this approach has been widely used in levelized compiled simulation, the following problems must be solved before it can be adapted to event driven simulation.

1. The levelization technique can not be applied to circuits containing feedback paths.
2. Zero delay components may create serious problems such as infinite loops in event driven simulation [2].

LECSIM solves the two problems by pre-processing the circuit in the following way. A standard depth first search algorithm is used to identify strongly connected components[15]. Within a strongly connected component, each fanout branch of each net is identified as either a forward path or a feedback path. (This identification is a natural byproduct of the depth first search algorithm.) LECSIM then levelizes each strongly connected component, ignoring the feedback paths. Although strictly speaking one strongly connected component cannot be embedded in another, it is sometimes advantageous to treat very large strongly connected components hierarchically. This is especially true when a strongly connected component contains some feedback paths that are considerably longer than others. For example, the implementation of a complex finite state machine might have several very long feedback paths in its control section, and several short feedback paths in the flip-flops that maintain the current state. In such cases it is advantageous to break the long feedback paths first and then identify the strongly connected components of the resultant circuit. At the lowest level of the hierarchy, LECSIM generates one block of code for all gates in the strongly connected component. An example of this type of strongly connected component is illustrated in Figure 3. Note that the code contains its own iteration control. The iteration stops when the strongly connected component stablizes or when a predetermined limit is reached. The iteration limit is determined by two ways. If a strongly connected component has *m* feedback arcs, and *m* is small, then the limit is set equal to 2^m+1. If *m* is larger than a certain number (currently 4), a user-specified default is used (currently 20).




```
{
   int x0;
   int count=0;
   do {
      x0 = Y2;
      Y1 = !(A&x0);
      Y2 = !(Y1&B);
      count++;
   } while (!(x0==Y2) && count<3);
   if (count==3 && x0!=Y2) unstable_error();
}
```

Figure 3. A Strongly Connected Component and Its Code.

The advantage of local loop control is that the iterations are performed in small fragments of code and do not involve queue insertion and event dispatch operations, which significantly reduces scheduling overhead. In the case where a strongly connected component contains a large number of gates or other strongly connected components, however, this approach may not efficient. For this type of strongly connected component, LECSIM uses a top level loop control scheme, which will be discussed in detail in the next section.

3. The Scheduling Algorithm

In the following discussion, we use the term "block" to represent the basic components to be scheduled. A block may contain a single gate, or it may be a cluster of gates such as an strongly connected component. Each block has an index number to indicate its level, which will be used for the block insertion operation.

The scheduling algorithm employed in LECSIM uses the concept of level-mapping in conjunction with a set of circular lists. The data structure used by the scheduler is organized by levels, as illustrated in Figure 4. This data structure is created at compile time rather than being allocated dynamically.

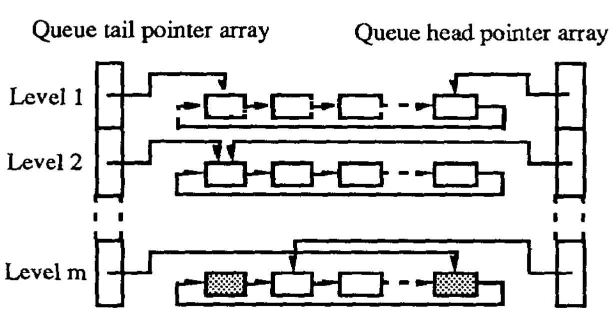


Figure 4. The data structure for the scheduling algorithm.

In Figure 4, the shaded boxes contain the blocks which need to be evaluated. Each level consists of a circular list and a pair of pointers. The circular list serves as the event queue for that level and only those blocks with the proper level index may be inserted into it. Each list contains n_i+1 slots, where n_i is the total number of blocks in level i. The queue head pointer points to the empty space in the list for next insertion and the queue tail pointer points to the block in the list which will be dispatched next. When the two pointers are the same, the queue for that level is empty.

The scheduling process involves a number of iterations, as illustrated in Figure 5. Each iteration consists of two level scanning operations. On top level, the scheduler scans through the queues by level, starting from level 1. At each level, it scans the circular list and performs all necessary operations such as evaluation and new event insertion. The unstable flag is set when one or more blocks are inserted into queues which have already been scanned during this iteration. This condition indicates that the circuit is not yet stable and another iteration is required. This process continues until the circuit reaches stable state or a user-specified iteration limit has been exceeded.

```
initial state : queue[i] contains the blocks in level i
                which need to be evaluated;

unstable = 1;
count = 0;
while (unstable = 1 and count < iteration_limit)
{
  unstable = 0;
  count = count + 1;
  for (current_level = 1 to m)
  {
    for (each block in queue[current_level])
    {
      evaluate the current block;
      if (new event generated)
      {
        update the output of the current block;
        for (each fan-out block of the current block)
        {
          (index denotes the level of the fan-out block)
          if (the block is not in the queue[index]
          {
            insert it into queue[index]
            if (index < current_level)
            {
              unstable = 1;
            }
          }
        }
      }
    }
  }
}
```

Figure 5. The scheduling algorithm

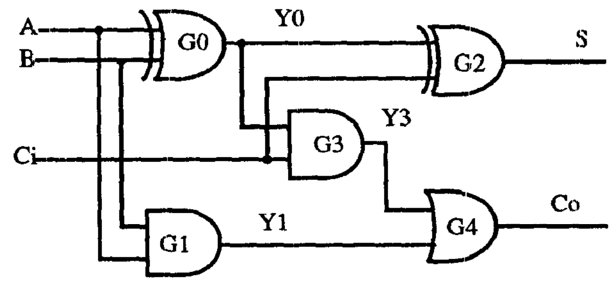
Though similar techniques have been proposed [5], this algorithm has the following distinguished features. First, the one pass levelized event scheduling technique, in conjunction with zero delay simulation, eliminates most of the unnecessary evaluations encountered in a two pass unit delay simulation. Second, the circular list structure simplifies event manipulation. Event insertions and dispatches involve only about 12 machine instructions. Third, small-sized tightly-coupled feedback loops are hidden within the strongly connected component code as discussed in section 2. These strongly connected components are presented to scheduler as if they were normal gates and will not induce top-level iterations. The only loops which will cause top-level iteration are those which contain many gates or contain other strongly connected components. Our experience indicates that these large loops usually require fewer iterations than the small-sized tightly coupled ones. In fact, we have observed that many circuits need only one iteration to reach stable state.

4 Implementation

Since the evaluation of a normal gate requires only a few instructions, an efficient implementation of the scheduling algorithm is critical for the overall performance of simulator. For event driven compiled simulation, the scheduling process involves scheduling the execution of a set of pre-generated routines. These routines are independent of each other and each represents a single block. This structure implies subroutine calls, which are used successfully in COSMOS where the evaluation of each routine takes a substantial amount of time. For gate-level simulation, however, the overhead of stack operations during subroutine calls becomes significant as compared to the execution time of the sub-routine. The most efficient implementation for gate level simulation is to make the starting address of each routine available so that the scheduler can jump to the routine directly. This approach is similar to threaded code [13] and has been used in SLS and Tortle_c. One difficulty of employing this approach is that it is necessary to store the routine addresses into variables which is quite difficult to do in a high-level programming language such as C. One could, of course, use assembly language for the output of the circuit-compiler, but this would severely impair the portability of the compiler. We have adopted a middle-of-the-road approach to simplicity and portability. The output of LECSIM is primarily C code with a few lines of assembly code inserted to implement the dispatcher. Although this impairs the portability of the compiler, the amount of generated assembly code is small, and can be quickly changed to adapt the compiler to a new environment. A sample of the generated code is illustrated in Figure 6.

The circuit of Figure 6 contains five blocks and three levels. The data structure consists of five integer arrays: the block address array *ad*, block flag array *fg*, the queue head pointer array *qh*, the queue tail pointer array *qt* and the array *bq* which reserves the memory space for three circular lists.

The code generated for this circuit contains three parts. The initialization procedure, which is not shown in Figure 6, is called at the beginning of simulation. It loads in the block addresses into array *ad* and constructs the circular lists. It also inserts all the blocks into queues and simulates circuit once to establish the initial state. The dispatcher is implemented in MC68020 assembly code (in SUN assembler format). The assembly code is inserted into the C program by calling C built-in function "asm." The dispatcher performs the task of scanning queue[i], the circular list of level i. The level is controlled by a level scanning routine, and is passed to dispatcher by loading address registers a4 and a3 with the queue head pointer and queue tail pointer respectively. The dispatcher checks to see if the queue is empty by comparing the contents of a4 and a3. If the queue is not empty, then it fetches the address of the block pointed to by the queue tail pointer, updates the queue tail pointer and then jumps to the block to be evaluated. The block routine, as shown in Figure 6 for the block BK0, contains both evaluation code and fan-out processing code. It first removes the block being evaluated from the queue by setting the its flag to 0. It then evaluates the block and if a new event has been generated, it processes any fan-out blocks. A fan-out block will be inserted into the queue indexed by its level if it is not already in the queue. When this process finishes, the program jumps to the dispatcher and is ready for next block.




```
int ad[5], fg[3], qh[3], qt[3], bq[16];


DISP:  asm("cmpl   a4,a3\n");
    asm("jeq      BK0\n");
    asm("movl   a3@,a0\n");
    asm("movl   a3@(4),a3\n");
    asm("jra       a0@\n");
    asm("KK0:\n");

BK0:    *(fg+0) = 0;
    new = A ^ B;
    if (Y0 != new) {
        Y0 = new;
        if (*(fg+2) == 0) {
            *(fg+2) = 1;
            qhp = qh+1;
            *((int *)*qhp) = *(ad+1);
            *(qhp) = *((int *)*qhp+1);
        }
        if (*(fg+3) == 0) {
            *(fg+3) = 1;
            qhp = qh+1;
            *((int *)*qhp) = *(ad+3);
            *(qhp) = *((int *)*qhp+1);
        }
    }
    goto DISP;
```

Figure 6. A Full Adder and Part of Its Code.

One obvious advantage of the compiled implementation of the event scheduler is its simplicity. This comes in two ways. First , the indices to the arrays are pre-calculated so that multi-indirect addressing is eliminated. Second, some simplification can be done during code generation. For example, the scheduling algorithm as shown in Figure 5 requires level checking after each block insertion to see if the unstable flag needs to be set. In an interpretive implementation, this requires at least two instructions to test the condition and one instruction to set the flag. In our implementation, the checking operation is done at code generation time. If the fan-out block level index is not lower than the current block level, which is usually the case, no level checking code will be generated. As a result, the event scheduling operation for most blocks requires only five machine instructions for the dispatcher and seven instructions for each fan-out. This contributes significantly to the efficiency of LECSIM.

5. Experimental Results

Ten small to medium size benchmark circuits from ISCAS85 were used to evaluate the performance of LECSIM. These circuits have been used frequently to benchmark the performance of ATPG packages and simulators. All circuits are combinational and their characteristics are shown in Figure 7. Two versions of LECSIM have been tested. The first treats singles gate as the primary elements and the second, called LECSIMp, treats fan-out-free blocks as the primary elements. These blocks are obtained by invoking a fan-out-free partitioning procedure in LECSIMp during circuit compilation. The characteristics of the partitioned circuits are shown in Figure 7.

Fan-out free partitioning was introduced to reduce the amount of scheduling time for the lowest-level components of the circuit. For example, suppose a fanout-free block contains 3 gates. Without partioning each of these gates must be scheduled individually. With partitioning the block containing the three gates will be scheduled as a unit, reducing the scheduling time by 66% for that block.

	No Partitioning		With Partitioning	
Circuit	Gates	Levels	Blocks	Levels
c432	160	17	60	13
c499	202	11	58	5
c880	383	24	105	17
c1355	546	24	258	15
c1908	880	40	377	38
c2670	1269	32	539	29
c3540	1669	47	555	44
c5315	2307	49	806	35
c6288	2416	124	1458	123
c7552	3513	43	1331	38

Figure 7. The ten ISCS85 benchmark circuits

The performance comparison between LECSIM and two other simulation packages, FHDL and EUSIM, is summarized in Figure 8. FHDL is a traditional levelized compiled logic simulator and EUSIM is a traditional unit delay, two-pass event driven interpretive logic simulator. Both simulators were developed from our previous research. All tests were performed on SUN 3/260 with 12 Mbyte of main memory. Each circuit was simulated with 5000 randomly generated vectors. The results of the comparison are listed in Figure 8. To provide accurate comparison of the algorithms and implementation techniques, we list only the net evaluation time. These figures do not include the time required to read vectors and print output.

The test results show that LECSIM runs about 8 to 77 times faster than EUSIM. The zero delay model and levelized event scheduling make significant contribution to the performance. On the average, the number of gates evaluated by LECSIM is only half to one third of that evaluated by EUSIM. For one particular example, the C6288 which has 124 levels, only 1 out of 26 gates are evaluated by LECSIM as compared to EUSIM. The rest of the performance improvement is due to the superiority of the compilation technique employed by LECSIM over the interpretive technique employed by EUSIM. As we expected, the event scheduling process, though efficiently implemented, introduces substantial amount of overhead. While LECSIM evaluates only about 50 to 60 percent of all gates, it is still 3 to 6 times slower than FHDL, which evaluates all gates for each test vector. This scheduling overhead can be reduced if we partition the circuits into fan-out-free blocks and schedule the events on the block level. Although LECSIMp evaluates more gates than LECSIM, it operates only about 1.5 to 3.4 times slower than FHDL.

Run Time in Seconds (SUN 3/260)				
Circuit	LECSIM	LECSIMp	EUSIM	FHDL
c432	95	134	190	160
c499	128	141	213	202
c880	219	330	356	383
c1355	309	372	763	546
c1908	500	605	1466	880
c2670	707	892	1550	1269
c3540	875	1321	2318	1669
c5315	1347	1684	3815	2307
c6288	1487	1586	39032	2416
c7552	2133	2603	6275	3513

Average Evaluations per vector				
Circuit	LECSIM	LECSIMp	EUSIM	FHDL
c432	3.5	2.2	41.7	1.4
c499	4.2	2.3	44.2	1.5
c880	7.4	4.3	79.6	3.2
c1355	17.0	8.8	171.7	5.0
c1908	32.0	16.2	399.1	7.1
c2670	47.6	27.2	432.2	9.8
c3540	61.4	36.8	560.7	12.1
c5315	100.1	56.2	877.1	16.5
c6288	117.6	90.6	9129.0	30.1
c7552	164.0	95.6	1389.0	39.0

Notes:
1. LECSIMp is LECSIM with fan-out-free partitioning.
2. EUSIM is a unit delay event driven interpretive simulator.
3. FHDL is a levelized compiled simulator.

Figure 8. Performance Comparison.

For EUSIM, the event-driven interpretetive unit-delay simulator, the average activity rate for the ten benchmark circuits is about 22% with random stimuli. This activity rate was calculated by first counting the number of gates that would be simulated by EUSIM in the worst case and dividing this number into the actual number of gates simulated. The number of gates simulated in the worst case was obtained by simulating the circuit and forcing each gate-evaluation to produce an event regardless of whether the output had changed. The average activity rates for LECSIM and LECSIMp are higher, 58% and 66% respectively for random stimuli. The activity rate of LECSIMp was measured with respect to blocks rather than gates. These figures suggest that LECSIM will out perform FHDL when the activity rate is less than 10-19%, and that LECSIMp will out perform FHDL when the activity rate is less than 19-44%. Therefore, we feel that LECSIM, and especially LECSIMp, will exhibit performance comparable to that of a levelized compiled simulator for many applications. On the other hand, EUSIM will out perform FHDL only when the activity rate is lower than 0.2-0.9%, an activity rate which we seldom expect to see in practice.

The major problems with LECSIM are that it requires more parse and compile time than an interpretive simulator, and it produces more generated code than a levelized compiled code simulator. In addition to the circuit parsing time, which is the same as for EUSIM, LECSIM consumes a significant amount of time in compiling the generated C program. This indicates that LECSIM is inefficient for the applications where only a few input vectors are simulated after each circuit modification. Figure 9 gives the parse and compile times for LECSIM and EUSIM, while Figure 10 gives the size of the generated code for LECSIM, LECSIMp, and FHDL.

Parse and compile time could probably be reduced through the use of incremental compilation[12]. Instead of recompiling whole circuit after each modification, incremental compilation recompiles only the part of circuit which has been changed. We have investigated this technique during the development of FHDL [14], and we believe that it will be readily adaptable to LECSIM.

At the time of this writing, we have not yet fully examined LECSIM's performance on sequential circuits.

	LECSIM			EUSIM
Circuit	Parse	Compile	Total	Parse
c432	1.0	7.4	8.4	0.9
c499	1.2	7.8	9.0	1.1
c880	2.5	13.8	16.3	2.2
c1355	3.5	63.6	67.1	3.0
c1908	4.7	28.4	33.1	4.1
c2670	8.1	42.6	50.7	7.5
c3540	10.6	49.7	59.1	9.4
c5315	17.1	76.2	93.3	15.2
c6288	18.9	380.0	399.0	17.2
c7552	28.0	553.0	581.0	25.4

Note: Time is measured in seconds on a SUN 3/260.

Figure 9. Circuit processing times.

	LECSIM	LECSIMp	FHDL
c432	51	41	32
c499	61	50	33
c880	106	76	41
c1355	135	104	57
c1908	184	125	82
c2670	278	197	117
c3540	334	222	124
c5315	483	321	184
c6288	526	417	208
c7552	683	443	270

Note: Numbers are size of object module in kilobytes.

Figure 10. Circuit memory requirements.

6 Conclusion and Future Work

LECSIM is a levelized event driven compiled logic simulator. It differs from traditional event driven interpretive simulators in that it employs levelization and compilation techniques to achieve high performance. It differs from traditional levelized compiled simulators in that it performs the simulation in a selective-trace, event-driven fashion. The experimental results demonstrate the superiority of this technique. LECSIM runs about 8 to 77 times faster than a traditional unit-delay event-driven interpretive simulator. While LECSIM is still slower than a traditional levelized compiled simulator (assuming that the same number of gates are simulated in each case), its event-driven approach will allow it to out perform levelized compiled simulators when circuit activity is low.

Although the techniques presented in this paper are oriented toward high-performance zero-delay simulation, we feel that they are readily adaptable to multi-delay simulation. In any case, the compiled implementation of event scheduler provides an efficient alternative to the central event management scheme used in most event driven simulators. We are presently extending LECSIM to include a multi-delay simulation ability which we hope will provide performance improvements comparable to those presented in this paper.

REFERENCES

1. M. A. Breuer, A. D. Friedman, *Diagnosis and Reliable Design of Digital Systems*, Computer Science Press, Woodland Hills, CA, 1976.
2. E. G. Ulrich, "Event Manipulation for Discrete Simulations Requiring Large Numbers of Events, " *Journal of the ACM*, Vol. 21, No. 9, Sept., 1978, pp. 777-85.
3. E. G. Ulrich, D. Herbert, "Speed and Accuracy in Digital Network Simulation based on Structural Modeling," *Proceedings of the 19th Design Automation Conference*, 1982, pp.587-93.
4. E. G. Ulrich, "Concurrent Simulation at the Switch,Gate,and Register Levels," *Proceedings of the 1985 International Test Conference*, 1985, pp.703-9.
5. S. Gai, F. Somenzi, M. Spalla, "Fast and Coherent Simulation with Zero Delay Elements," *IEEE Trans. on Computer-Aided Design*, Vol. 6, No. 1, Jan., 1987, pp.85-91.
6. S. P. Smith, M. R. Mercer, B. Brock, "Demand Driven Simulation: BACKSIM", *Proceedings of the 24th Design Automation Conference*, 1987, pp.181-87.
7. L. Wang, N. Hoover, E. Porter, J. Zasio, "SSIM: A Software Levelized Compiled- Code Simulator," *Proceedings of the 24th Design Automation Conference*, 1987, pp.2-8.
8. C. Hansen, "Hardware Logic Simulation by Compilation," *Proceedings of 25th Design Automation Conference*, 1988, pp.712-15.
9. R. E. Bryant, D. Beatty, K. Brace, K. Cho, T. Sheffler, "COSMOS: A Compiled Simulator for MOS Circuits," *Proceedings of the 24th Design Automation Conference*, 1987, pp.9-16.
10. Z. Barzilai, D. K. Beece, L. M. Huisman, V. S. Iyengar, G. M. Silberman, "SLS-A Fast Switch-Level Simulator," *IEEE Trans. on Computer-Aided Design*, Vol. 7, No. 8, August, 1988, pp. 838-49.
11. Z. Barzilai, J. L. Carter. B. K. Rosen, J. D. Rutledge, "HSS-A High-Speed Simulator," *IEEE Trans. on Computer-Aided Design*, Vol. 6, No. 4, July, 1987, pp. 601-16.
12. D. M. Lewis, "Hierarchical Compiled Event-Driven Logic Simulation," *proceeding of ICCAD-89.*
13. J. R. Bell, "Threaded Code," *Journal of the ACM*, Vol. 16, No. 6, June., 1973, pp. 777-85.
14. P. M. Maurer, Z. Wang, C. D. Morency, "Techniques for Multi-Level Compiled Simulation," *University of South Florida, Department of Computer Science and Engineering Technical Report*, CSE-89-04.
15. A. V. Aho, J. E. Hopcoff, J. D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, Reading, MA, 1983.