Author's Note: Although this paper was published many years ago in both conference and journal form (DAC/TCAD), I am reissuing this as a Baylor Computer Science Technical Report in the interest of keeping all simulation papers in the same place.

Two New Techniques for Unit-Delay Compiled Simulation

Peter M. Maurer
Department of Computer Science
Baylor University
Waco, TX 76798

Abstract

The PC-set method and the parallel technique are two methods for generating compiled unit-delay simulations of acyclic circuits. The PC-set method analyzes the network, determines the set of potential change times for each net, and generates gate simulations for each potential change. The parallel technique, which is based on the concept of bit-parallel simulation, is faster and generates less code than the PC-set method, but is not amenable to data-parallel simulation of multiple input vectors. Both techniques are based on the well known levelization algorithm used to generate zero-delay Levelized Compiled Code simulations. Although the parallel technique provides for efficient simulations with a reasonable amount of generated code, there are several opportunities for optimization. The two optimization schemes presented here are called bit-field trimming and shift-elimination. Two different methods of shift elimination are presented, one which is based on breaking cycles in an undirected graph, and one which is based on tracing paths through a network. Performance results are presented for both simulation techniques, and for all optimizations. Comparisons with interpreted event-driven simulation using the ISCAS 85 benchmarks show a factor of 4 improvement for the PC-set method and a factor of ten improvement for the parallel technique. Similar studies for the optimization schemes show an average performance improvement of 47% over the unoptimized simulations.

Two New Techniques for Unit-Delay Compiled Simulation

Peter M. Maurer
Department of Computer Science
Baylor University
Waco, TX 76798

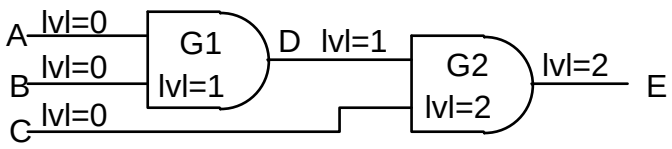
1 1. Introduction

Recently there has been much renewed interest in compiled simulation, primarily because it provides a significant increase in performance over interpreted methods [1-7]. As was pointed out in [2], some of the reasons that complied simulation has received less attention than interpreted simulation is that the traditional methods for producing Levelized Compiled Code (LCC) simulators are oriented toward zero-delay simulation and synchronous circuits, while interpreted event-driven simulation is oriented toward asynchronous circuits and more realistic timing models. While the natural bias of Levelized Compiled Code simulation is indeed toward zero-delay simulation, more sophisticated code generation algorithms can be used to generate code for other timing models. For example, the COSMOS compiled simulator [1] incorporates an event queue to perform unit-delay simulation, while the tortle.c simulator[7] uses a threaded code technique[8] and a distributed event scheduler. The techniques presented here are also oriented towards unit-delay simulation, but are similar to LCC simulators in that no event queue is needed, and the generated code executes in "straight-line" fashion without tests or branches. On the average, our fastest technique runs in about 1/10 the time required for an interpreted event-driven simulation.

The algorithms presented in this paper assume that each gate has a delay of 1 time-unit, which is called a *gate delay*. The length of a gate delay is not specified, but is tacitly assumed to be small enough so that any number could elapse between two successive input vectors. All circuits are assumed to be acyclic, however our algorithms can be applied to a wide variety of synchronous sequential circuits by requiring that any cycle in the network contain at least one flip-flop. The circuit could then be broken at the flip-flops by treating the flip-flop inputs as primary outputs and the outputs as a primary inputs. We are currently working to extend our techniques to asynchronous sequential circuits.

The algorithms described in this paper are based on the well-known technique of Levelized-Compiled-Code (LCC) simulation [2,9]. In LCC simulation every net and every gate in the circuit is assigned a level number, with primary inputs, constant signals and flip-flop outputs being assigned level zero. Once all inputs of a gate have been assigned levels, the maximum of these levels is found, incremented by one, and assigned to the gate and all of its outputs. If there are wired AND or OR connections in the circuit, the net is assigned the maximum of the levels of its source gates. Levelization is a variation of topological sorting for directed graphs[10], and like topological sorting, requires its input to be acyclic. The levelization process is illustrated in Fig. 1. Fig. 1 also contains a typical example of the code generated by a LCC simulator. Note that code for the individual gates is generated in ascending order by levels. In this and all other code examples, the bit-wise logical operations of the C language are used. The symbols &, | and

~ represent bit-wise AND, OR, and NOT respectively, and the symbols << and >> represent left and right shifts respectively. The expression *a*<<2 means, "shift *a* to the left 2 bits."




```
int A,B,C,D,E
...
D=A&B;
E=C&D;
```

Fig. 1. Levelized Compiled Code generation.

From another point of view, the level of a net corresponds to number of gates on the longest path between the primary inputs of the circuit and the net. In unit-delay simulation, the level also represents the *time* (in gate delays) at which all changes in the primary inputs will have propagated to the net, and is therefore, the latest time at which the net is permitted to change value.

As stated above, the algorithms presented in this paper are based on the levelization algorithm. One simple variation of this algorithm is the algorithm to determine the time at which changes in the primary inputs *first* reach the net, which call the *minlevel* of the net. The minlevel algorithm is identical to the levelization algorithm, except that when a gate is processed, the *minimum* of the minlevels of the inputs of the gate is computed, incremented by one, and assigned to the gate and its outputs as their minlevel. For wired AND and OR connections the minlevel of the net is equal to the *minimum* of the minlevels of the driving gates.

A generalization of both the levelization and minlevel algorithms is presented in section 2. This algorithm is used as a basis for generating *unit-delay* Levelized Compiled Code, and for optimizing the algorithm presented in Section 3. Section 3 presents another very different algorithm for generating unit-delay Levelized Compiled Code. This algorithm uses bit-parallel operations to reduce both the time and space needed to perform a simulation. Section 4 presents optimizations for the algorithm presented in section 3, Section 5 presents experimental results using the ISCAS-85 combinational benchmarks[11], and Section 6 draws conclusions.

2 2. The PC-Set Method of Simulation.

The level and minlevel of a net are the lengths of the shortest and longest paths between the net and the primary inputs of the circuit, where the length of a path is defined to be the number of gates on that path. The level and minlevel are also the latest and earliest times at which a net is permitted to change value, where time is measured in gate delays. These two facts can be generalized into the following lemma the proof of which is left to the reader.

Lemma 1. The value of a net is permitted to change at time *t* if and only if there is a path of length *t* between the net and the primary inputs of a circuit.

The set of times at which a net is permitted to change value is called the *PC-set* (Potential Change set) of the net, and is computed using the following algorithm, which is a variation of both the levelization and the minlevel algorithm.

1. Assign a count, *k*, to every net *n* and every gate *g* in the circuit. If *g* is a gate, then then *k* is equal to the number of inputs of *g*. If *n* is a net, then *k*, is equal to the number of gates driving the net.
2. Place any net *n* whose count is zero on the processing queue. (Note: these will be primary inputs and constant signals.)
3. If the processing queue is empty, then stop, otherwise remove either a gate *g* or a net *n.*
4. If a net *n* was selected then
   a. Form, U, the union of the PC-sets of all gates driving *n*.
   b. If U is empty, set it equal to {0}.
   c. Assign U to *n* as its PC-set.
   d. For every gate *g* in the fanout set of *n*, reduce the count of *g* by 1. (Note: if *n* appears twice in the input list of a gate *g* then the count of *g* is reduced by 2, and so forth.) If the count of *g* is now zero, then place *g* on the processing queue.
5. If a gate *g* was selected then
   a. Form, U, the union of the PC-sets of all input nets of *g*.
   b. Increment every element of U by 1 to form U'.
   c. Assign U' to *g* as its PC-set.
   d. For every output net *n* of *g*, reduce the count of *n* by 1. If the count of *n* is now zero, then place *n* on the processing queue.
6. Go back to step 3.

In the PC-Set algorithm, every primary input and constant signal assigned the set {0}. Once all PC-sets have been calculated for the inputs of a gate, the union of these sets is formed and each element is incremented by one. This new PC-set is assigned to the gate and each of its outputs. For wired connections, the PC-set of the net is the union of the PC-sets of its source-gates.

The PC-set represents the lengths of all paths between a net and the primary inputs. For example, if net A has the PC-set {3,7,15} then there are paths of length 3, 7, and 15 between the net and the primary inputs. The size of the PC-set of any net is restricted to *level-minlevel*+1. Note that the PC-set contains both the level and the minlevel of a net.

Because, by lemma 1, the PC-set of a net determines when the net is permitted to change value, it can be used to generate Levelized-Compiled-Code for unit-delay simulation. However, it is first necessary to determine which nets must retain their values from the previous input vector. To see why this is necessary, consider the gate pictured in Fig. 2.

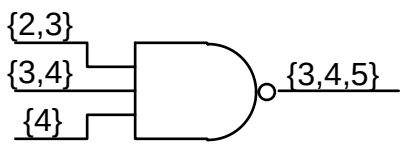


Fig. 2. A gate with PC-sets.

Three output values must be calculated for the gate pictured in Fig. 2, at times 3, 4, and 5 respectively. When the value at time 3 is calculated, the values of the input nets at time 2 must be used. However, two of the input nets will not change value until times 3 and 4. This implies that the calculation of the first output value must use the values of these nets that have been retained from the previous input-vector. In general, to determine which nets must retain their values from the previous input-vector, the minlevels of the inputs of each gate are compared. If the inputs of a gate do not have identical minlevels, any net whose minlevel is not minimal for the gate must retain its value from the previous input vector. If a net must retain its value from the previous vector, the element zero will be added to its PC-set, as illustrated in Fig. 3.

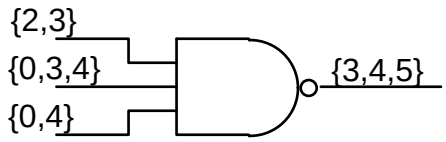
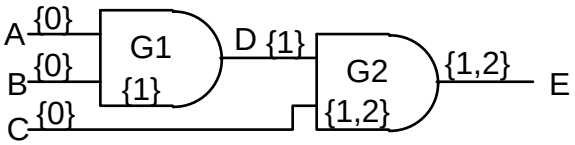


Fig. 3. Adding zero to the PC-set.

Once zeros have been added to the appropriate PC-sets, one variable is generated for each element of the PC-set of each net. Next, initialization code is generated which assigns values to all variables that correspond to time zero. Code is generated to read input vectors and assign values to primary inputs, and code is generated to initialize those nets that have had zeros added to their PC-sets. The initialization for nets with added zeros consists of moving the final value of the net into the variable that corresponds to the zero PC-set element. This value can always be found in the variable that corresponds to the maximum PC-set value.

Next, the simulation code itself is generated. Gates are processed in levelized order, and one gate-simulation is generated for each element of the gate's PC-set, exclusive of the zero element, if any. The operands of each gate simulation are obtained by searching the PC-sets of the input nets for the largest element that is strictly smaller than the PC-element for which code is being generated, and using the corresponding variable. The insertion of zero elements guarantees that such an element always exists. Fig. 4 shows a network and part of the code that is generated from it. For simplicity, the code to read input vectors and print results has been omitted.




```
int   A_0,B_0,C_0,D_0,D_1,E_1,E_2;
...
D_0 = D_1;
D_1 = A_0 & B_0;
E_1 = D_0 & C_0;
E_2 = D_1 & C_0;
...
```

Fig. 4. Code generated by the PC-set method.

In the code of Fig. 4, it is assumed that the low-order bit of each integer variable contains the value of the net. Two variables, E_1 and E_2, are generated for net E. The PC-set for D must have a zero added, so two variables are generated for D as well. Since D has a zero added to its PC set, an initialization statement, "D_0 = D_1;" must be generated for it. The code pictured in Fig. 4 is executed once per input vector and creates a complete history for the vector.

The final step in code generation is to create the output routine. The output routine is treated as if it were a special gate of type PRINT, whose inputs correspond to the the set of monitored nets. Zero insertion is performed on the set of monitored variables, just as if they were all inputs to a single gate. The union of the PC-sets of the monitored variables is computed and the result is used as the PC-set of the output routine. One print statement is generated for each element of output routine PC-set. Each print statement creates one output vector. A procedure identical to that for handling gate inputs is used to determine which values to place in each output vector.

Because PC-sets represent potential changes rather than actual changes, the PC-set method causes some unnecessary computation to be done. Nevertheless, our studies on standard benchmark circuits show that the PC-set method is significantly faster than interpreted event-driven simulation. (See section 5.)

3 3. The Parallel Technique.

One of the major drawbacks of the PC-set method is that it tends to generate an enormous amount of code (over 100,000 lines for c6288, one of our benchmark circuits). The parallel technique of compiled unit-delay simulation was originally intended to reduce the amount of code generated by the PC-set method, but it turns out that it not only generates significantly less code than the PC-set method, it also runs much faster. On the other hand, the PC-set method is amenable to bit-parallel simulation of multiple input vectors[12], while the parallel technique is not.

Code generation for the parallel technique begins by allocating an *n*-bit field for each net, where *n* is the number of different levels in the circuit. (In other words, *n* is equal to *d*+1, where *d* is the depth of the circuit.) The bit-fields are then mapped into variables. Our current implementation maps the bit-fields into 32-bit words with the low-order bit of each field aligned with the low-order bit of the first word. If the bit-field width is not an exact multiple of 32, it is rounded up to the next highest multiple. Each bit in the field corresponds to one time unit. The low-order bit represents time zero, and contains either the initialization value of the net or the final value calculated from the previous input vector.

As in the PC-set method, the first part of the generated code sets the value of all bit-positions that represent time zero. The low-order bits of the fields corresponding to primary inputs are read from an input vector. For all other fields, the final value of the net, which is located in the high-order bit, is shifted into the low-order bit of the field. Since primary inputs change only at time zero, all bits in a primary-input field must be identical. Therefore, the low-order bit is propagated through all bits such fields. For other fields, all bits other than the low-order are set to zero to facilitate further processing.

Next, simulation code is generated for each gate in levelized order. Bit-parallel operations are used to simulate the function of a gate, and left-shifts are used to model the delay of the gate. Fig. 5 shows the simulation of a 2-input AND gate on a 4-bit field. In this example, the bit-fields of the inputs, *a* and *b*, contain values for times 0, 1, 2, and 3. Due to the delay of the gate, which is equal to one, the bit-field, *i*, produced by the bit-parallel AND operation contains the value of the gate output at times 1, 2, 3, and 4. The intermediate result, *i*, must be shifted to the left one

bit to make its alignment match that of the bit-field of the gate output, *c*. The shifted intermediate result is ORed with the bit-field of *c* to preserve the low-order bit *c*0. Fig. 6 contains a portion of the code generated by the parallel technique for the network pictured in Fig. 2. Fig. 7 illustrates the effect of executing the code of Fig. 6, by showing the bit-fields that would be computed for each net.

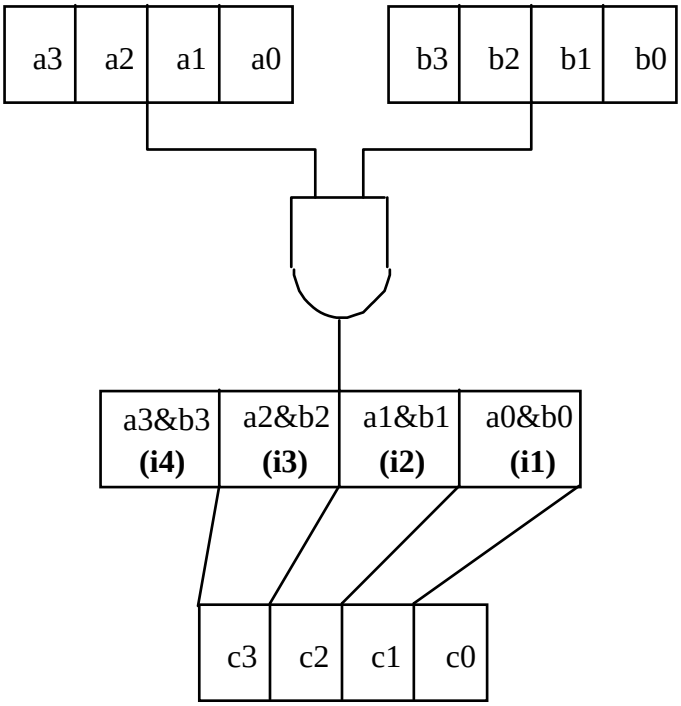


Fig. 5. The operations performed by the parallel technique.

```
char A,B,C,D,E;
...
D = (D>>2)&1;
E = (E>>2)&1;
D = D | ((A & B)<<1);
E = E | ((D & C)<<1);
...
```

Fig. 6. Code generated by the parallel technique.

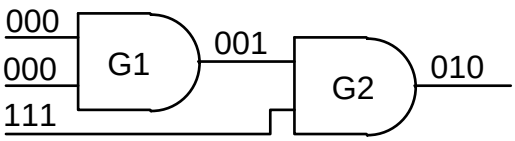


Fig. 7. Bit-fields computed by the parallel technique.

If bit-fields are longer than 32 bits, they must be split into several words. Gate simulations must be replicated for each word, and the left-shift operation must transfer bits between words.

For an individual gate, the simulation time for a two-word bit-field is more than double that for a one-word bit-field. Fig. 8 illustrates the difference between a two-word gate simulation and a one-word simulation.

```
...
temp =  A & B;
C = C | (temp<<1);
...

...
temp_0 = A_0 & B_0;
temp_1 = A_1 & B_1;
C_1 = temp_0>>31;
C_0 = C_0 | (temp_0<<1);
C_1 = C_1 | (temp_1<<1);
...
```

Fig. 8. One-word simulation vs. two-word simulation.

The final step in code generation is to generate output code for monitored variables. A trace of the value of all monitored variables is printed by using a sliding mask to determine the value of each net at each time unit. Although the current implementation of the parallel technique does not perform hazard analysis, such analysis could be done quickly by using a binary search technique and comparison fields of the form 0...01...1 and 1...10...0.

4 4. Optimizing the Parallel Technique.

Even without optimization, the Parallel Technique provides the highest performance for the benchmark circuits that we have tested, however there is still considerable opportunity for further reducing the execution time. We have investigated two different optimization techniques, the first of which seeks to reduce computation time for circuits whose bit-fields are larger than 32-bits, and the second of which concentrates on eliminating the shifts that follow each gate evaluation. Most of the effort has gone towards eliminating shift operations, because much of the simulation time for deep circuits is consumed by these operations.

Most circuits of a practical size have a depth greater than 32, which implies that multiple-word bit-fields must be used, with the additional computation illustrated in Fig. 8. *Bit-field trimming* is the process of eliminating unnecessary operations from simulation code and shift operations when multiple-word bit-fields are used. It has no effect on circuits whose bit-fields fit in a single word. Bit-field trimming makes use of the PC-sets described in Section 2 above. Each element of a PC-set corresponds to a bit-position in the bit-field of a net. If the PC-set contains the value *x*, then position *x* in the bit-field is called a *PC-set representative*.

The first step in bit field trimming is to identify those nets whose minlevel is greater than or equal to 32. The low-order word (or words) for these nets must contain the final value of the net computed from the previous input vector in every bit-position, which implies that no new computation is required to determine the value of these words. When a new input vector is read, the value computed from the previous vector is propagated throughout every bit of the low-order words of such nets, and no simulation code is generated for them.

The next step is to examine the PC-set of each net for gaps. A *gap* is any word, other than those identified in the previous step, that contains no PC-set representatives. When a gap is encountered, the high-order bit from the previous word is propagated throughout every bit of the gap, and no simulation code is generated for it. Gap detection is especially important for nets near the primary inputs of a circuit, which have no PC-set representatives in their high-order words.

Bit-field trimming can also eliminate parts of a shift operation. To do this, corresponding words of the shifted and unshifted output are checked for the presence of PC-set representatives. If neither word contains any PC-set representatives, then the shifted and unshifted word must be identical, and no shift operations are generated for it. Fig. 9 illustrates the various operations performed during bit-field trimming.

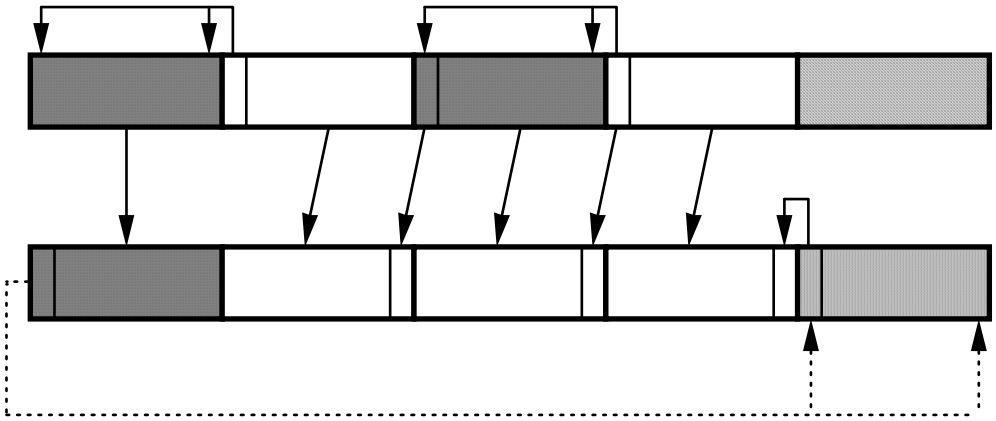


Fig. 9. Operations performed during bit-field trimming.

In Fig. 9, it is assumed that bit-fields are implemented as five words. The unshifted intermediate result and the final shifted result are for a single gate simulation are illustrated. The shaded words are those that contain no PC-set representatives. The dotted line represents the initialization step that copies the high-order bit of the bit-field into all of the bits of the low-order word. The low-order word of the unshifted bit-field does not participate in either the simulation step or the shift. The third and fifth words of the unshifted bit-field represent gaps. The high-order bit of the word preceding each gap is copied into every bit of the gap. During the shift operation, only the second, third, and fourth words are shifted. The low-order bit of the second word is taken from the high-order bit of the first word, and the fifth word is copied without shifting. The effect of bit-field trimming on performance is noted in Section 5.

As stated above, most of the optimization effort has gone into eliminating the shifts that follow each gate evaluation. To see why these operations can be eliminated, consider why they are necessary in the first place. Each bit-field is aligned so that the low-order bit corresponds to time zero. Since a gate simulation requires one time unit, the unshifted intermediate result is aligned so that the low-order bit corresponds to time 1. Thus it is necessary to shift the intermediate result so that its alignment matches that of the bit-field of the output net. By allowing different bit-fields to have different alignments, shift operations can be eliminated. For example, consider the network pictured in Fig. 10. Because the smallest element of the PC-set of net E is 1 (see Fig. 4), the alignment of net E must be 1 or smaller, otherwise some changes in E could be lost. If the alignment of net E is set to 1,the alignment of nets C and D can be set to

zero. Finally, the alignment of nets A and B can be set to minus one. It is now possible to eliminate all shifts from the simulation of this network and, surprisingly, it is also possible to reduce the width of the bit-fields from 3 to 2. The required bit-field width for each net, which is computed using the formula *level-alignment*+1, gives a maximum value of 2 for any net. (In this example, it may be wise to keep the 3-bit bit-fields, because the glitch in the output would be easier to detect, but in general, eliminating shifts can make the size of the bit field either larger or smaller.)

Negative alignments may seem peculiar, but they affect only the processing of primary inputs where they are handled in the following manner. When a primary input is assigned a new value from an input vector, its previous value is copied into all bits whose index is negative. The new value is propagated through those bits whose indexes are zero or positive. Note that the code of Fig. 10 omits the initializations for nets D and E. When negative alignments are used for primary inputs, initialization code is not required for any nets other than primary inputs. For other nets, the values from the previous input vector are automatically recomputed wherever they are required.

```
char A,B,C,D,E;
...
D = A & B;
E = D & C;
...
```

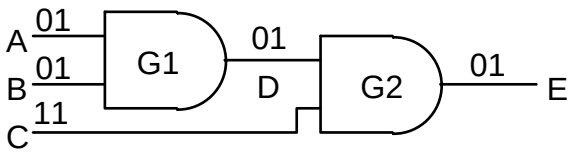


Fig. 10. Optimization of the parallel technique.

Note that the code illustrated in Fig. 10 is identical to the code that would be produced for a zero delay LCC simulation. The only difference in the two simulations is the way that input vectors are processed. Unfortunately, it is impossible to eliminate all shifts from all simulations. To see why this is so, first observe that if all shifts are to be eliminated from a simulation, alignments must be adjusted so the following four conditions hold.

1. The alignment of every net must be less than or equal to its minlevel.
2. All input nets of a gate must have have the same alignment.
3. All output nets of a gate must have the same alignment.
4. The alignment of the output nets of a gate must be one larger than the alignment of the input nets.

For most networks it is impossible to force these conditions to be true simultaneously, as Fig. 11 shows.

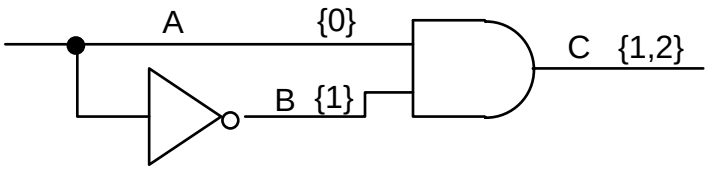


Fig. 11. A network that requires one shift.

The reason that conditions 1-4 cannot be enforced for the network of Fig. 11, is that condition 2 requires that the alignments of A and B be equal, while condition 4 requires that the alignment of B be one larger than the alignment of A. Because conditions 1-4 cannot be enforced, it is impossible to eliminate all shifts from the simulation of this network. The shift after the AND can be eliminated, but the shift after the NOT must be retained. In general, if a network contains reconvergent fanout along differing length paths, then it will be necessary to retain some shift operations. However, there are networks without reconvergent fanout that must retain shift operations, as Fig. 12 illustrates.

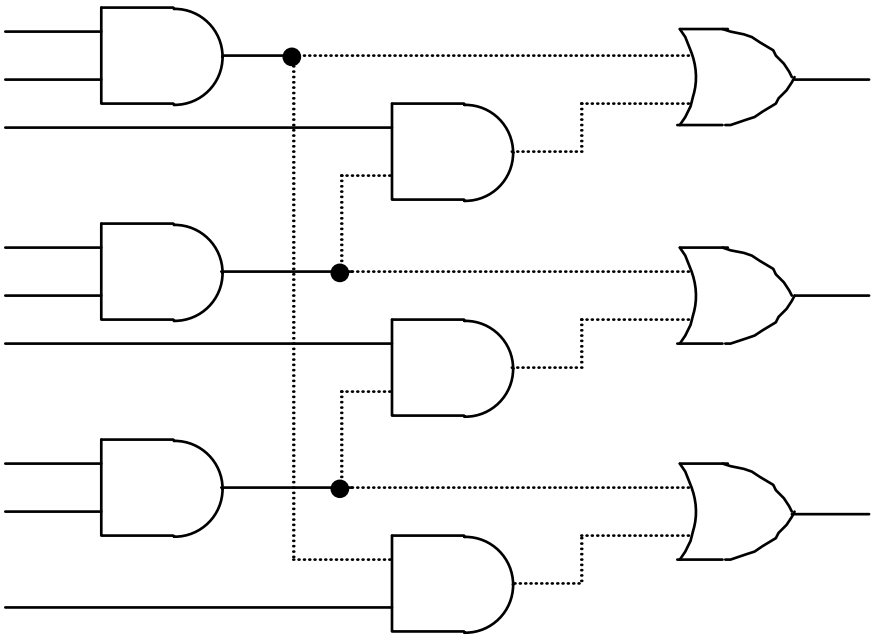


Fig. 12. A network without reconvergent fanout that requires one shift.

The dotted lines in Fig. 12 illustrate the portion of the network that prevents conditions 1-4 from being enforced. The concept of an *undirected network graph* can be used to characterize networks that must retain shifts. An undirected network graph contains one vertex for each gate and each net in the circuit. All edges in the undirected network graph connect a gate vertex to a net vertex. There is an *undirected* edge between a gate vertex and a net vertex if and only if the gate uses the net either as an input or an output. Fig. 13 illustrates the undirected network graph for the network of Fig. 11.

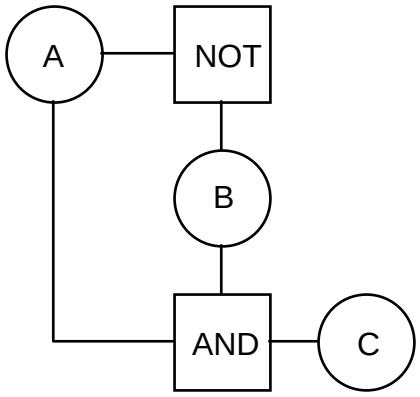


Fig. 13. An undirected network graph.

Note that the graph of Fig. 13 is cyclic. Although the absence of cycles in the undirected network graph is sufficient for conditions 1-4 to be enforceable, the presence of cycles is not sufficient to guarantee that they are *unenforceable*. To determine whether a cycle will prevent conditions 1-4 from being enforced, it is necessary to traverse the cycle and assign weights to each the vertices in the cycle. (The cycle may be traversed in either direction.) For simplicity, the traversal is assumed to begin on a net-vertex. Each net-vertex is given a weight of zero, while gate vertices can be given given weights of zero, one, or minus one. Assume that when visiting a gate-vertex *G*, the path *NGM* is traversed. (Only simple cycles are considered, so *G* can be visited only once.) The vertices *N* and *M* must be net vertices that correspond to inputs or outputs of *G*. If *N* and *M* are both inputs or both outputs, then *G* is assigned a weight of zero. If *N* is an input and *M* is an output then *G* is assigned a weight of one. If *N* is an output and *M* is an input then *G* is assigned a weight of minus one. The weight of the cycle is the total of the weights of its vertices. A necessary and sufficient condition for a cycle to prevent the enforcement of conditions 1-4 is that its weight be non-zero. Assuming that the cycle of Fig. 13 is traversed in a clockwise direction, vertices A, B, and AND will be given the weight zero, and the vertex NOT will be given the weight 1. The cycle will have the weight 1. The choice of the initial direction affects only the sign of the weight, not its magnitude. The cycle represented by the dotted lines in Fig. 12 has a weight of 3 or -3 depending on direction.

If a cycle of non-zero weight does not share vertices with any other cycle, it will require one shift in the simulation code, the magnitude of which is equal to its weight. (Shifts are no longer restricted to one bit, and either a right or a left shift may be required.) The minimum number of shifts required by a network is less than or equal to the number of edges that must be removed from the undirected network graph to make it acyclic, but surprisingly, there are cases where it is *undesirable* to eliminate all shifts. To see why this is so, consider the undirected network graph pictured in Fig. 14.

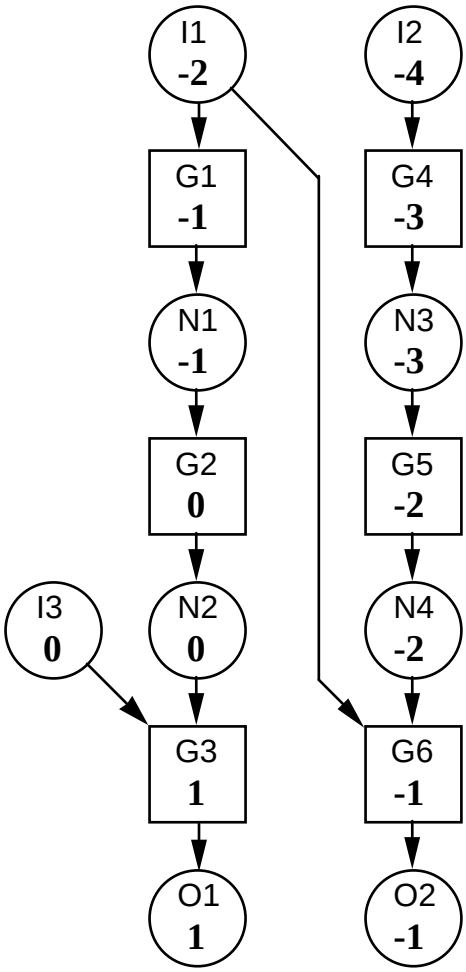


Fig. 14. The undesirability of eliminating all shifts.

In Fig. 14, it is assumed that alignment begins with the primary output "O1." The network contains 4 levels, so the width of the bit-field is 4 before alignment. After alignment the required bit-field width of I2 is 5. (Level=0, Alignment=-4, Required Width= 0 - (-4) + 1 = 5.) All shifts have been eliminated from the simulation of the network, but at the expense of expanding the bit-field. Although expansion of the bit-field from 4 to 5 bits will have no effect on the simulation of this circuit, there are cases where expansion of the bit-field can have an extremely detrimental effect on the simulation time. For example, if the width of the bit-field expanded from 32 bits to 33, the amount of simulation time could more than double.

We have experimented with two approaches for eliminating shifts, one that is based on a general cycle-breaking algorithm, and another which traces paths upward through the network from primary outputs to primary inputs. The cycle-breaking algorithm uses the undirected network graph to detect cycles and to generate alignments for each net in the circuit. A depth-first search is performed on the undirected network graph, and when a cycle is found, the most recently traversed edge is removed. Once all cycles have been broken, an alignment is generated for each gate and net in the circuit by performing another depth-first search. The search starts at an arbitrary primary output, which is aligned to its minimum PC-set value. When a net-vertex is visited, all gates that use the net as an output are assigned an alignment equal to the alignment of

the net, and all gates that use the net as an input are assigned an alignment equal to the alignment of the net plus 1. When a gate-vertex is visited, all inputs of the gate are assigned an alignment equal to the alignment of the gate minus one, and all outputs of the gate are assigned an alignment equal to the alignment of the gate. Fig. 15 illustrates how alignments are assigned. This algorithm does not guarantee that the alignment of every net will be less than or equal to its minlevel. To guarantee this, a second pass is required to (possibly) reduce all alignments by a constant amount.

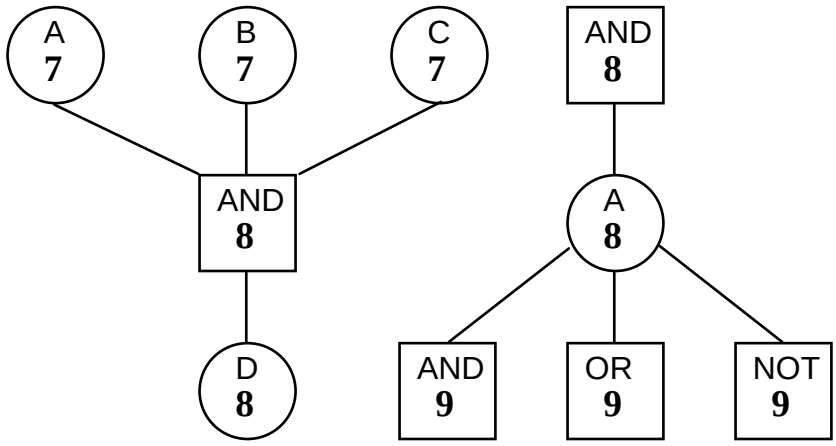
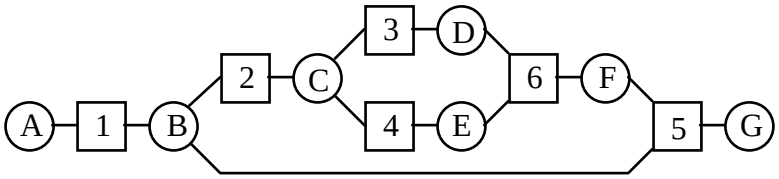
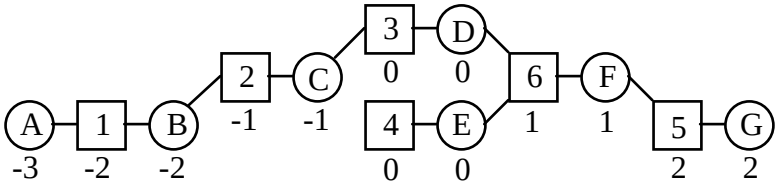


Fig. 15. Examples of assigning alignments.

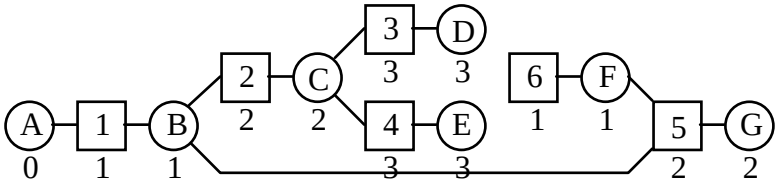
Although the cycle breaking algorithm gives better results than path tracing for pathological circuits such as that pictured in Fig. 12 (one three-bit shift as opposed to three one-bit shifts), it tends to perform much worse than path tracing on more realistic circuits (see Section 5). This result was somewhat surprising, because the cycle-breaking algorithm removes the minimum number of edges required to make the undirected network graph acyclic. The number of edges that must be removed from each connected component is equal to $F=E-V+1$, where E is the number of edges in the component, and V is the number of vertices. F is precisely the number of back-arcs that will be detected during the depth-first search of the component. However, because not all cycles require shifts, the number of retained shifts tends to be quite sensitive to precisely which edges are removed from the directed network graph. Fig. 16.a illustrates an undirected network graph that must have two edges removed to become acyclic. Figs. 16.b and 16.c show two different ways that this can be done, along with the alignments that would be produced in each case.. The network of Fig. 16.b requires one shift to be retained (from net B to gate 5), while that of Fig. 16.c requires two shifts to be retained from nets D and E to gate 6).



a. An undirected network graph



a. One shift required



a. Two shifts required

Fig. 16. The Effect of Removing Different Edges.

By far the worst drawback of the general cycle-breaking algorithm is that tends, to greatly expand the size of the bit-fields. (See Section 5.) The path tracing algorithm does not expand the size of the bit-field, and may actually reduce it. The path-tracing algorithm begins with a primary output, and proceeds upwards toward the primary inputs. The algorithm does not require that edges be removed from the undirected network graph before alignments are assigned. The alignment of the starting primary output is set equal to its minimum PC-set value. Once the alignment of a net has been assigned, the alignment of all gates that drive the net is set equal to the alignment of the net. Similarly, once the alignment of a gate has been determined, the alignment of all inputs of the gate is set to one less than the alignment of the gate. If the algorithm encounters a gate or a net whose alignment has been previously assigned, it compares the previous alignment with the alignment that would be assigned in the current step. If the new alignment is smaller than the old alignment, the algorithm continues as if the old alignment did not exist. Otherwise the algorithm leaves the old alignment unchanged and does not search upward through that gate or net. Fig. 17 gives pseudo-code for this algorithm.

```
Initialize all alignments to some large value;

for each primary output p
    net_align(p, minlevel of p)
endfor

net_align(net, new_alignment)
begin
    if new_alignment < net.alignment
        net.alignment = new_alignment;
        for each gate g that drives net
            gate_align(g, new_alignment);
        endfor
    endif
end  /* net_align */

gate_align(gate, new_alignment)
begin
    if new_alignment < gate.alignment
        gate.alignment = new_alignment;
        for each input n  of gate
            net_align(n, new_alignment-1);
        endfor
    endif
end  /* gate_align */
```

Fig. 17. Pseudo-code for the path-tracing algorithm.

To show why the path-tracing algorithm does not expand the width of the bit-field, it is necessary to distinguish between two types of alignment. In the "gate_align" algorithm of Fig. 17, the alignments of the *inputs* of a gate are adjusted to match the alignment of the gate. When alignments move in the direction of the primary inputs of a circuit, as they do in both the "gate_align" and "net_align" subroutines of Fig. 17, we call this forcing alignments *up the network*. When alignments move toward the primary outputs, which would be the case if the *outputs* of a gate were aligned to match the alignment of the gate, we call this forcing alignments *down the network*. The width of the bit-field expands only when alignments are forced down the network, which implies that the path tracing algorithm cannot expand the size of the bit-field. The reason that forcing alignments up the network cannot expand the width of the bit field (which is calculated using the formula *width=level-alignment*+1) is that any time *alignment* becomes smaller (which would tend to make *width* larger), *level* is guaranteed to become smaller by *at least* an equal amount (which will keep *width* at the same size or make it smaller). On the other hand, when forcing alignments down the network, *alignment* can increase by at most 1, but *level* can increase by an arbitrarily large amount. A careful examination of Fig. 14 will show that the expansion of the bit-field from 4 to 5 is caused by forcing the alignment down from net I1 (level=0) to gate G6 (level=3).

In addition to not expanding the width of the bit-field, the path-tracing algorithm has other advantages over the cycle-breaking algorithm. Every gate is guaranteed to be aligned in

accordance with one of its outputs, and every net is guaranteed to be aligned in accordance with one of the gates that use it as an input. This implies that if a net that does not fan out it will not require a shift, and that any fanout-free region of the circuit will be simulated without shifts.

Another advantage is that the path-tracing algorithm generates only right shifts, while the cycle-breaking algorithm can generate both right and left shifts. Right shifts cause bits to be introduced at the high-order end of the bit field. These bits can simply be replicated from the high-order bit. On the other hand, left shifts (such as those used by the unoptimized algorithm) may need to shift in bits from the previous input vector. Depending on the alignment of the net, these bits may not be available unless special provisions are made to retain them.

Happily, the path-tracing algorithm usually outperforms the cycle-breaking algorithm in an absolute sense, in that it tends to retain fewer shifts than the cycle-breaking algorithm. The cycle-breaking algorithm does perform better for some circuits. Data on bit-field widths and retained shifts for both algorithms is presented in Section 5.

The process for generating code when bit-fields have differing alignments is somewhat different from that used by the unoptimized compiler. In the unoptimized compiler, the shifts are performed on the output of a gate. However when gates and nets have a variety of alignments, each fanout branch of a net may require a different shift, as illustrated in Fig. 18.

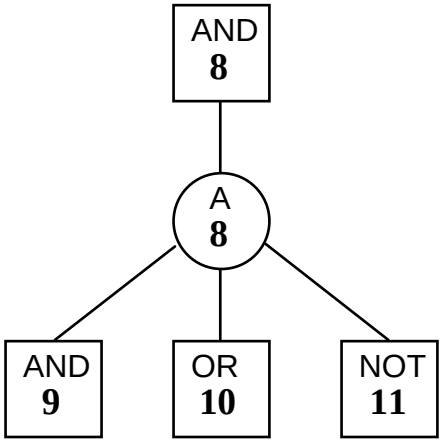


Fig. 18. Multiple shifts for a single net.

Because a net may need to be shifted many times, shifts are done at the inputs of a gate rather than the outputs. When code is generated for a gate, code is also generated to align the bit-fields of the inputs to correspond to the alignment of the gate. No shift is performed following the gate-simulation. For the cycle-breaking algorithm, the shifts may be to the right or to the left, while for the path-tracing algorithm generates only right shifts. For either algorithm multiple-bit shifts may be required. In terms of instruction count, multiple-bit shifts are just as efficient as single-bit shifts. When left-shifts are permitted it is necessary to guarantee that the alignment of each of the affected nets is strictly smaller than its minlevel. This will guarantee that the value from the previous input vector is available for the shift. As noted above, the initialization for non-primary-input nets which is done by the unoptimized compiler is unnecessary when differing net-alignments are used.

5 5. Experimental Results.

This section presents the results of several experiments to compare the performance of different simulation techniques and optimizations. All experiments were based on the ISCAS85 benchmarks which have been used by several researchers for measuring the performance of logic simulators [11]. To obtain the simulation results, each circuit was simulated on 5,000 randomly generated vectors. All simulators were developed locally. All experiments were run on a SUN 3/260 with twelve megabytes of memory and a dedicated disk drive. This system was isolated during the simulations, to eliminate interference from other users. The timings were obtained using the UNIX "/bin/time" command. To minimize the error in this command, each experiment was run five times and the results of the five trials were averaged. None of the execution times presented in this section include the time required for reading vectors, printing output, or compiling circuit descriptions. The reader should note that the results depend on the quality of implementation of the various simulators. While we do not claim that our implementations are optimal, we believe that they are comparable in quality.

Fig. 19 contains the basic results of a comparison between interpreted event-driven unit-delay simulation and the two simulation techniques presented in this paper. The first two columns present simulation results from two conventional unit-delay event-driven simulators, which used a three-valued and a two-valued logic model respectively. Since three-valued logic is the more natural model for event-driven simulation, the first column contains the most realistic numbers. The two-valued simulation results were provided to demonstrate that the increased performance of the PC-set method and the parallel technique are *not* due to the difference in logic models. (Both compiled simulators use a two-valued logic model.) For the parallel technique, circuits c432-c1355 required one word per bit-field, while all others except c6288 required two words per bit-field. C6288 required 4 words per bit-field. The simulations for the parallel technique were run *without* optimization.

Simulation Time in Seconds				
	Interp. 3 value	Interp. 2 value	PC-Set Method	Parallel Technique
c432	46.4	41.2	9.9	3.4
c499	51.1	44.3	5.2	4.4
c880	87.1	78.1	22.4	8.1
c1355	177.2	157.7	84.9	9.8
c1908	330.2	295.9	162.7	54.3
c2670	368.2	346.1	89.9	90.7
c3540	531.1	479.1	211.6	122.2
c5315	1024.0	894.7	245.2	176.0
c6288	9555.9	8918.3	1757.3	369.3
c7552	1483.2	1348.5	395.2	269.7

Fig. 19. Simulation results without optimization.

Fig. 19 clearly shows the advantage of the PC-set method over interpreted event-driven simulation, and also demonstrates the superiority of the parallel technique. The anomaly between the PC-set method and the parallel technique for circuit c2670 is due to the unusually small size of the PC-sets for this circuit. On the average, the PC-set method runs in one fourth the time required for an interpreted event simulation, while the parallel technique runs in about

one tenth the time. To put these numbers in perspective, we performed a similar study for zero delay simulation. Our results for zero-delay simulation show that on the average a compiled simulation runs in 1/23 the time of an interpreted simulation.

Fig. 20 illustrates the effect of bit-field trimming on the performance of the parallel technique. Column 1 lists the number of levels in each circuit, with the number of 32-bit words required to represent a bit-field in parentheses. Column 2 gives the CPU time in seconds for the unoptimized parallel technique, while column 3 gives the results for bit-field trimming. The improvement in performance ranges from 20% to 36% with an average of 26%.

	Levels	Parallel Technique	Bit-Field Trimming
c432	18(1)	3.4	3.3
c499	12(1)	4.4	4.4
c880	25(1)	8.1	8.1
c1355	25(1)	9.8	11.6
c1908	41(2)	54.3	37.0
c2670	33(2)	90.7	64.8
c3540	48(2)	122.2	97.7
c5315	50(2)	176.0	137.1
c6288	125(4)	369.3	266.8
c7552	44(2)	269.7	205.5

Fig. 20. The performance of bit-field trimming.

Fig. 21 gives the number of shifts retained by the path-tracing and cycle-breaking shift-elimination algorithms, while Fig. 22 shows the bit-field width for the two algorithms. Note that the path-tracing algorithm *reduces* the width of the bit-field for some circuits.

Retained Shifts			
	Unoptimized	Cycle-Breaking	Path-Tracing
c432	160	65	100
c499	202	72	96
c880	383	140	163
c1355	546	223	296
c1908	880	437	398
c2670	1269	532	461
c3540	1669	827	713
c5315	2307	1123	1060
c6288	2416	1397	1764
c7552	3513	1875	1830

Fig. 21. Retained shifts for two algorithms.

Bit Field Widths			
	Unoptimized	Cycle-Breaking	Path-Tracing
c432	18	18	16
c499	12	25	11
c880	25	26	19
c1355	25	35	22
c1908	41	122	38
c2670	33	110	28
c3540	48	113	41
c5315	50	187	46
c6288	125	448	121
c7552	44	323	38

Fig. 22. Bit-field widths for the two algorithms.

Fig. 23 contains the performance results for both the path-tracing algorithm and the cycle-breaking algorithm on the ten ISCAS-85 benchmark circuits. The results of Fig. 23 show that the path-tracing algorithm provides significant increases in performance for all circuits. The performance increase ranges from 24% to 84% with an average of 43%. On the other hand, for all but the smallest circuits, the cycle-breaking algorithm performs significantly *worse* than the unoptimized algorithm. This points out that the expansion of the bit-field more than negates the beneficial effects of eliminating shifts. The results for circuits c6288 and c7552 are not shown in Fig. 23, because a bug in the C compiler prevented compilation of the generated code. There is a straightforward but very time consuming fix to the code generator to avoid this bug, but the results of Fig. 22 suggest that the run times for these circuits would be unacceptably large, so it is not worth the effort to insert this fix.

	Unoptimized	Cycle-Breaking	Path-Tracing
c432	3.4	2.2	2.4
c499	4.4	3.0	2.9
c880	8.1	4.9	4.9
c1355	9.8	15.7	7.4
c1908	54.3	85.0	21.9
c2670	90.7	110.2	14.4
c3540	122.2	169.7	68.9
c5315	176.0	339.0	108.0
c6288	369.3	**	240.1
c7552	269.7	**	160.4

** Cancelled Due to C Compiler Bug

Fig. 23. Performance of the path-tracing and the cycle-breaking algorithms

The results of Fig. 23 do not include the benefits of performing bit-field trimming along with shift-elimination. The algorithms for bit-field trimming used with shift-elimination are essentially the same as those without shift elimination, with the exception that initialization for the low-order words of a bit-field must be reintroduced for the low-order words of the bit-field

that do not contain PC-set representatives. Since negative alignments are possible with shift-elimination, the probability of eliminating simulation code for low-order portions of the bit-field is greater than for the unoptimized compiler. Fig. 24 shows the performance results of combining bit-field trimming with the path-tracing shift-elimination algorithm.

	Unoptimized	Path-Tracing	With Trimming
c432	3.4	2.4	2.4
c499	4.4	2.9	2.9
c880	8.1	4.9	5.0
c1355	9.8	7.4	7.4
c1908	54.3	21.9	18.1
c2670	90.7	14.4	14.1
c3540	122.2	68.9	58.4
c5315	176.0	108.0	91.4
c6288	369.3	240.1	196.9
c7552	269.7	160.4	133.4

Fig. 24. Combining shift-elimination with bit-field trimming.

As before, the performance gain ranges from 24% to 84%. (The minimum and maximum for shift-elimination are achieved on circuits whose bit-fields fit in a single word, so trimming will have no effect.) However, the average performance increase is 47%, somewhat higher than that achieved using shift-elimination alone.

6 6. Conclusion.

Two techniques for compiled unit-delay simulation have been presented which provide significant performance improvements over event-driven interpreted simulation. The PC-set method of generating code is amenable to bit-parallel simulation of multiple input vectors, but usually generates much larger programs than the parallel technique. Although the unoptimized parallel technique is faster than the other techniques studied, it provides significant opportunities for optimization. These techniques can be used to simulate a wide variety of circuits, including many synchronous sequential circuits. Work is currently under way to extend adapt these techniques to asynchronous circuits, and to adapt them to even more accurate timing models.

7 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. M. Chiang, and R. Palkovic, "LCC Simulators Speed Development of Synchronous Hardware," *Computer Design*, Mar. 1, 1986, pp. 87-91.

3. 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.

4. C. Hansen, "Hardware Logic Simulation by Compilation," *Proceedings of the 25th Design Automation Conference*, 1988, pp. 712-715

5. 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.

6. Z. Wang and P. M. Maurer, "Scheduling High-Level Blocks for Functional Simulation," *Proceedings of the 26th Design Automation Conference*, 1989, pp. 87-90.

7. D. M. Lewis, "Hierarchical Compiled Event-Driven Logic Simulation," *IEEE International Conference on Computer Aided Design*, 1989, pp. 498-501.

8. J. R. Bell, "Threaded Code," *Journal of the ACM*, Vol. 16, No. 6, June., 1973, pp. 777-85.

9. M. A. Breuer and A. D. Friedman, *Diagnosis and Reliable Design of Digital Systems*, Computer Science Press Inc., Rockville MD, 1976.

10. D. E. Knuth, *The Art of Computer Programming, Vo1. 1,* Addison Wesley, Reading, MA, 1973.

11. F. Brglez, P. Pownall, R. Hum, "Accelerated ATPG and Fault Grading via Testability Analysis," *Proceedings of the International Conference on Circuits and Systems*, 1985, pp. 695-698.

12. R. E. Bryant, "Data Parallel Switch-Level Simulation," *IEEE International Conference on Computer Aided Design*, 1988, pp. 354-357.