Optimization of the Parallel Technique for Compiled Unit-Delay Simulation

Peter M. Maurer

Technical Report DA-9, 1990, VCAPP Laboratory,
Dept. of Computer Sci. & Eng. University of South Florida
Tampa, Florida 33620

D. Maclise, R.A. H. Robinson

OPTIMIZATION OF THE PARALLEL TECHNIQUE FOR COMPILED UNIT-DELAY SIMULATION

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

ABSTRACT

The parallel technique of compiled simulation is a purely compiled method for unit-delay simulation that is based on the concepts of levelized compiled simulation and parallel fault simulation. Although the parallel technique provides very rapid simulations with a reasonable amount of generated code, there are several opportunities for optimizing the generated code. This paper presents two optimization schemes which are called bit-field trimming and shift-elimination. Two different methods of shift elimination are presented. Performance results are presented for all optimization techniques. These results show that using both optimizations schemes together provides for an average performance improvement of 47% (over the unoptimized simulations) for the circuits tested.

OPTIMIZATION OF THE PARALLEL TECHNIQUE FOR COMPILED UNIT-DELAY SIMULATION

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

1. Introduction.

Recently there has been much renewed interest in compiled simulation, primarily because it provides significant increases in performance over interpreted simulation techniques [1-7]. Although the fundamental techniques for compiled simulation have been known for many years[8], they have been largely abandoned in favor of slower but more accurate simulation techniques. However, current research in very large synchronous integrated circuits has provided a niche for compiled simulation in high-performance functional simulation[5]. Such high-level simulations generally do not require the accuracy provided by unit-delay and timing simulations, and also tend to require an enormous number of tests. In such situations the increase in performance more than compensates for any loss in accuracy.

Traditionally compiled simulators have been based on a zero-delay simulation model, but recent research has begun extending the scope of compiled simulation to the unit-delay model. Much of this research has focused on providing a compiled implementation of an event-driven simulation[2][7]. However, other research has focused on extending the traditional techniques of levelized compiled simulation to handle unit-delay and other types of simulation[9]. This paper focuses on one such technique called the "parallel technique" and demonstrates how this technique can be extensively optimized. Section 2 of this paper describes the parallel technique and discusses opportunities for optimization. Section 3 describes the optimization technique of bit-field trimming, while section 4 describes techniques for shift-elimination. Section 5 describes a technique for combining the two optimizations, and section 6 draws conclusions.

2. The Parallel Technique.

The parallel technique is a method for creating a unit-delay levelized compiled code simulation of a circuit, and is loosely based on the concept of parallel fault simulation[10].

The first step in the parallel technique is to levelize the circuit and allocate a bit-field for each net in the circuit and map the bit-fields into variables. During levelization, primary inputs are assigned to level zero. When all inputs of a gate have been assigned to levels, the outputs of the gate are assigned to the level which is one larger than the maximum of the levels of the inputs. Levelization requires that a circuit be acyclic, which implies that the parallel technique is restricted to acyclic circuits. Research is currently under way to remove this restriction, but this is beyond the scope of this paper.

The width of each bit-field is determined by the number of levels in the circuit, so if there are *n* levels in the circuit, an *n*-bit field is allocated for each net. Each bit in the field corresponds to one logical time unit. In a unit delay simulation, it takes *n*-1 logical time units to propagate a value through an n-level circuit, so *n* bits are sufficient to represent the initialization values at time zero plus all values calculated for the net during simulation. The low-order bit of each bit-field represents time zero, and contains either the initialization value of the net or the final value calculated from the last input vector. After the circuit has been simulated, the bit-field for a net contains a complete history of the net's value for the simulation of the current vector.

The parallel technique is a purely compiled simulation technique in that the generated code is straight-line without tests or branches. The generated code must be compiled, and then executed to simulate the circuit. The straight-line code is embedded in a loop that repeatedly reads input vectors, simulates them, and prints output. The first portion of the loop initializes all bit-fields by shifting the high-order bit into position zero, and clearing the remainder of the field to zeros. The high-order bit represents the final value of the net from the previous vector. The shift operation moves the final value of the net from the previous vector into logical-time zero for the current vector.

When the input vector is read, the value of each primary input is propagated throughout every bit in the field. This is necessary because a primary input can be used at any level of the circuit. Even if primary inputs were used only by level-1 gates, propagation of the input value throughout all bits of the bit-field would still be necessary to guarantee that the final value of each net ends up in the high-order bit of the bit-field.

Gate simulation code is generated in levelized order, and is quite similar to that produced by a zero-delay levelized-compiled-code simulator. The low-level logical operations are performed using bit-parallel operations on the entire bit-field, the result is shifted one bit to the left, and is ORed into the bit-fields representing the gate's outputs. The left-shift represents the delay of the gate, while the OR operation allows the low-order bit of the output-field to be preserved. Once all gate simulations have been performed, a

trace of all monitored nets is printed. Hazard analysis can also be performed at this point. For more information, see [9].

3. Trimming the Bit-Field.

One difficulty in implementing the parallel technique is that on most existing architectures, the width of a hardware bit-field is limited to 32-bits. This is too small for most practical applications, so wider bit-fields must be emulated in software. Even on architectures such as the IBM-370 which supports bit-fields of up to 2048 bits, the execution time of an instruction is proportional to the width of the bit-field. Therefore, it is advantageous to trim the size of a bit-field wherever possible.

When bit-fields must be broken into several words, the computation for a single gate consists of two steps. First the a temporary unshifted output is computed using bit-parallel operations on the bit-fields of the gate's inputs. These operations must be replicated for each word in the bit-field of the gate's output nets. Second, the temporary output is shifted to the left one bit, and assigned to the output net. The shift operation is complicated by the fact that bits must be shifted between words. Bit-field trimming is the process of eliminating unnecessary operations from the simulation code and from the shift operations. Bit-field trimming has no effect on circuits whose bit-fields fit in a single word.

To perform bit-field trimming it is first necessary to compute an object called the PC-set for each net (and each gate) in the circuit. The PC-set, which is the set of those logical times at which a net may potentially change value, is calculated using an algorithm similar to the levelization procedure described above. Every primary input is assigned the set {0}, which signifies that the value of the primary input changes only at initialization time. 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 to 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 elements of a PC-set correspond to bit-positions in the bit-field of a net. If the PC-set contains the value x, then bit-position x is called a *PC-set representative*.

The value of a net cannot change at any logical time that is not contained in its PC-set. For example, if a net has the PC-set {1,3,7}, then the value of the net may change only at logical times 1, 3 and 7. The value of the net at logical times 4, 5, and 6 will be identical to the value at logical time 3. This information can be used to eliminate operations on portions of the bit-field. In the following discussion it is assumed that all bit-fields are implemented as sets of 32-bit words, and that the length of any bit-field which is not a multiple of 32 is rounded up to the next highest multiple. (For any circuit, all bit-fields are assumed to be

the same width.) As stated above, the high-order bit of the high-order word of each bit-field contains the final value of the net after the gate driving the net has been simulated. (This follows from the fact that input values are propagated through every bit of their bit-fields.) Because the final value of the net is conveniently located, it can be propagated throughout every bit of a word using a right-shift with sign-extension.

The first step in bit field trimming is to identify those nets whose minimum PC-set value is greater than or equal to 32. The low-order words for these nets must contain the value computed from the previous vector in every bit-position. After a new input vector has been read, the value computed from the previous vector is propagated throughout every bit of the low-order words of such nets. Since these words now contain their final values, 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 first 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.

The PC-set can also be used to 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 are identical, and no shift operations are generated for it. Figure 1 illustrates the various operations performed during bit-field trimming.

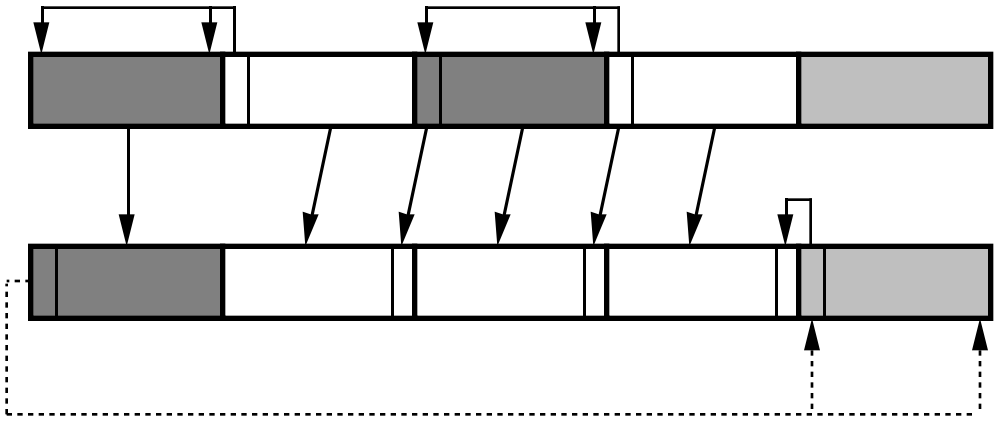

Figure 1. Operations Performed During Bit-Field Trimming.

Figure 1 shows the shifted and unshifted bit-fields for a gate-evaluation. Each bit-field is implemented as five words. The shaded words represent words 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 the simulation step or in the shift. The third and fifth words of the unshifted bit-field represent gaps, so the high-order bit of the previous word is copied into every bit. 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.

Figure 2 illustrates the effect of bit-field trimming on the performance of the parallel technique.

	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

Figure 2. The Performance of Bit-Field Trimming.

Figure 2 reports the results of running simulations on the ten ISCAS-85 combinational benchmarks[11]. These circuits have been used by a number of researchers to evaluate the performance of both simulators and automatic test generation software. 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. These figures do not include the time required to read and print vectors. These figures were obtained by running the generated code 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. The improvement in performance ranges from 20% to 36% with an average of 26%.

4. Eliminating Shifts.

Although bit-field trimming provides a significant increase in performance, it is effective only for those circuits with bit-fields wider than one word. Furthermore, much of the simulation time for deep circuits is consumed by the shift operations that follow each gate simulation, so the next optimization focuses on eliminating as many shift operations as possible.

To clarify the following discussion, it is necessary to review why the shift operations are used. To begin with, the low-order bit of each bit-field represents logical time zero. Since a gate simulation consumes one logical time unit, the low-order bit of the unshifted bit-field represents time one. Thus it is necessary to shift the bit-field so that its content is aligned with the bit-field of the gate's outputs. However, it is not necessary for all bit-fields to have the same alignment. In fact if different bit-fields are allowed to have different alignments, some shift operations can be eliminated. To illustrate this concept, consider the circuit pictured in Figure 3.

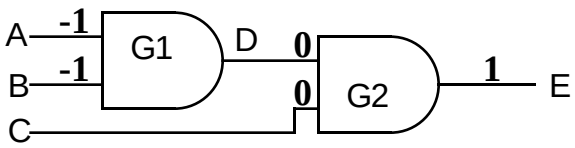

Figure 3. Bit-Field Alignments for a Simple Circuit.

In Figure 3, the bit-field for net E is aligned so its low-order bit represents logical time 1. (As will be explained below, this is the maximum permissible value for the alignment of net E.) Next, the bit-fields for nets C and D are aligned so their low-order bits represent logical time zero. Finally, the bit-fields for nets A and B are aligned so their low-order bits represent logical time -1. With these alignments, shifts are no longer necessary, but special processing is required to handle the negative alignments of nets A and B. When an input net has a negative alignment, all bit positions that correspond to negative logical times are initialized with the value from the previous input vector, and all other bit positions are initialized with the value from the current input vector. This is a special case of the general rule that all bit positions that represent logical times prior to the net's minimum PC-set value must contain the value computed from the previous input vector. The fact that values from the previous input vector are included in the bit-fields of the primary inputs eliminates the necessity for initializing every bit-field prior to simulating the circuit. Values from the previous input vector are recomputed wherever necessary.

Although there is no minimum value for a bit-field's alignment, there is a maximum value. Since the bit-field must contain a complete history of the net for the current input vector, the bit-field must contain a position corresponding to every element of the net's PC-set. Thus the maximum alignment for a net's bit-field is equal to the minimum element of the net's PC-set. In Figure 3, the minimum PC-set element for net E is 1, hence the maximum bit-field alignment is also 1.

In the ideal situation all shifts will be eliminated. For this to be true, the following four conditions hold.

1. The alignment of every net's bit-field is less than or equal to the minimum PC-set element of the net.
2. For a particular gate, all input nets must have have the same alignment.
3. For a particular gate, all output nets must have the same alignment.
4. The alignment of a gate's output nets must be one larger than the alignment of its input nets.

Unfortunately, for most networks it is impossible to force all four conditions to be true simultaneously. Figure 4 gives an example of such a network.

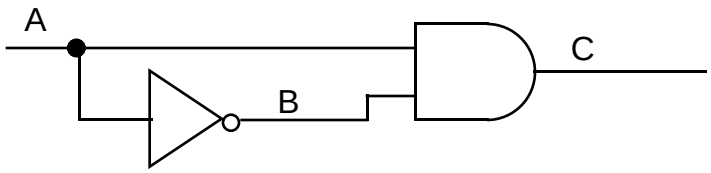

Figure 4. A Network That Requires One Shift.

Conditions 1-4 cannot be enforced for the network of Figure 4, because 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. It is necessary to retain the shift following the NOT gate, but the shift following the AND gate may be eliminated. In general, if a network contains reconvergent fanout along differing length paths, then it will be necessary to retain some shifts. There are also networks without reconvergent fanout that must retain some shift operations, but these tend to be less common. (See [9] for an example of such a network.)

To determine the minimum number of shifts required for a particular network, it is necessary to create an object called the undirected network graph. The undirected network graph of a circuit contains one vertex for each gate and each net in the circuit. If gate G

uses net N as an input or an output, there will be an undirected edge between the vertex representing G and the vertex representing N. Figure 5 illustrates the undirected network graph of the circuit pictured in Figure 4.

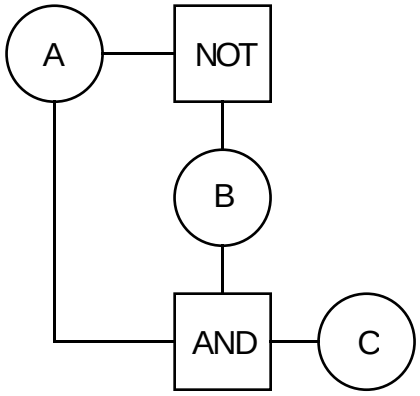

Figure 5. An Undirected Network Graph.

If the undirected network graph of a circuit is acyclic, then it will be possible to eliminate all shifts from the generated code for the circuit. In most cases, each cycle in the undirected network graph will require the retention of one shift. (For a complete characterization of the cycles that require shift retention, see [9].) Therefore, the minimum number of shifts is equal to the minimum number of edges that must be removed from the undirected network graph to make it acyclic. In general this is a very difficult problem, so it must be attacked heuristically. We have studied two different approaches to this problem. The first is a general cycle-breaking algorithm, while the second is a path-tracing algorithm.

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 eliminated. 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. Figure 6 illustrates how alignments are assigned.

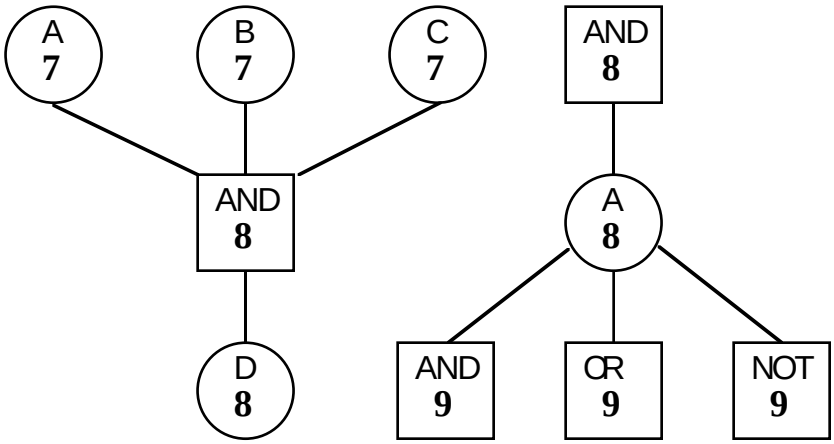

Figure 6. Examples of Assigning Alignments.

After each gate and each net has been assigned an alignment, a second pass is done to guarantee that each net's alignment is at least as small as its minimum PC-set value. (The previous steps do not guarantee that this will be true.) If any nets have been given alignments that are too large, a constant is subtracted from the alignment of each gate and each net in the component containing the invalid net. (The tree created by cycle-breaking may contain several components.) The code-generation method is independent of the method for computing alignments, so code-generation will be discussed following the discussion of the path-tracing algorithm.

Although the cycle-breaking algorithm allows many shifts to be eliminated from the generated code for a circuit, it contains a fatal flaw that makes future investigation of this technique unwarranted. The cycle-breaking algorithm may increase the width of the bit-field, sometimes by an enormous amount. This is truly a fatal flaw, since increasing the width of the bit-field from 30 bits (say) to 35 bits will double the amount of generated code. This will more than negate the effect of eliminating shifts.

To construct an algorithm that does not increase the width of the bit-field, it is necessary to understand how this phenomenon occurs. To begin with, it is necessary to distinguish between two methods of propagating alignments. When alignments are propagated in the direction of the signal flow, we call this *forcing alignments down* the network. In other words when an output net is aligned to match the gate that drives it, or when a gate is aligned to match one of its input nets, this is forcing alignments *down* the network. On the other hand, when alignments are propagated against the direction of the signal flow, we call this *forcing alignments up* the network. Forcing alignments *up* occurs

when a gate is aligned to match one of its output nets, or when a gate-input is aligned to match the alignment of the gate.

The width of the bit-field expands only when alignments are forced down the network. To see how this occurs, consider the circuit pictured in Figure 7.

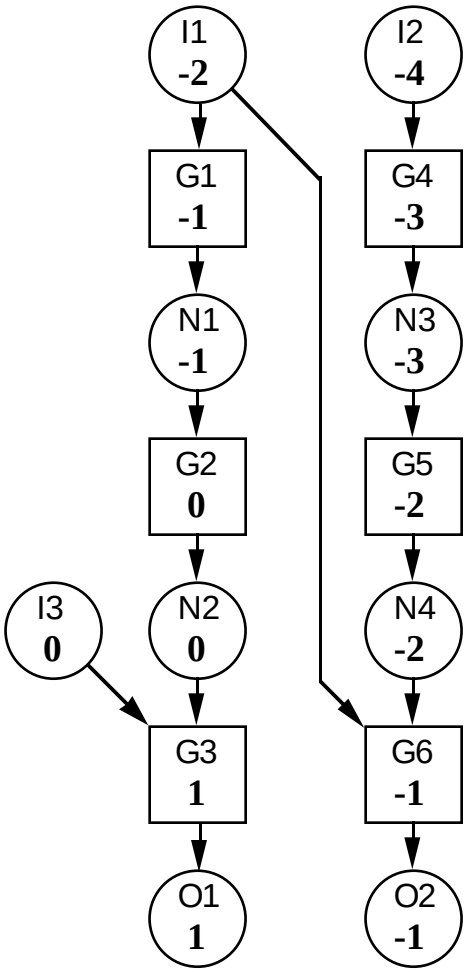

Figure 7. Expansion of the Bit-Field.

In the undirected network graph of Figure 7, it is assumed that alignment begins with the primary output "O1." It is further assumed that this graph was acyclic and did not require deletion of edges before alignment. Under these assumptions, the network contains 4 levels, so the width of the bit-field is 4 before alignment. However after alignment the width of the bit-field is 5. This can be computed from the alignment of the primary input "I2" using the formula *max-PC-alignment*+1. A careful examination of Figure 7 will show that it is the forcing of the alignment down from "I1" to "G6" that causes the expansion in the bit-field.

To show that forcing alignments up the network does not cause expansion in the bit-field, assume that an algorithm does not force alignments down the network, that it begins processing with an arbitrary primary output, and that it sets the alignment of the primary output to its minimum PC-set value. Assume further that the network contains L levels. The required bit-field width for a gate or a net is M-A+1, where M is the maximum PC-set value, and A is the alignment. Now if x is an element of any PC-set, then $0 \le x < L$, so if bw is the required bit-field width of the starting primary output then $bw = M - m + 1 \le L$, where M is the nets maximum PC-set value, and m is its minimum PC-set value.

Assume that a net is being processed, and that its alignment is nA and its maximum PC-set value is nM. By induction we assume that $nM - nA + 1 \le L$. Since alignments are forced up only, any gate alignment assigned in this step must be equal to nA. Furthermore, the maximum PC-set value gM of any gate that drives must be less than or equal to nM. Therefore the required bit-field width of any gate whose alignment is assigned in this step must be less than or equal to $nM - nA + 1 \le L$.

Now assume that a gate is being processed, that its alignment is gA, and that its maximum PC-set value is gM. Again by induction we assume that the required bit-field width of the gate, $gM - gA + 1 \le L$. Since alignments are forced only up, the alignment of any net assigned in this step must be equal to gA-1. The maximum PC-set value of any net assigned in this step must be less than or equal to gM-1. Therefore the required bit-field width of any net whose alignment is assigned in this step must be less than or equal to $gM - 1 - (gA - 1) + 1 = gM - gA + 1 \le L$.

We conclude from these arguments that forcing alignments up the network cannot increase the width of the bit-field, as long as the algorithm begins with a primary output and sets the alignment of that output to its minimum PC-set value. The path-tracing algorithm is just such an algorithm. The path-tracing algorithm begins with a primary output, and proceeds upwards toward the primary inputs. The algorithm never searches downward, and 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 to its minimum PC-set value. Alignments are forced up the network until a primary input is encountered. Because alignments are never forced down the network, the algorithm must search upward from *every* primary output. If the algorithm encounters a gate or a net whose alignment has been previously set, it compares its previous alignment with the new alignment that would be assigned by the current step. If the new alignment is smaller than the old alignment, the algorithm replaces the old alignment with the new alignment, and continues searching upward as though the gate or net had no previous alignment.

However, if the new alignment is greater than or equal to the old alignment, the algorithm leaves the old alignment unchanged and does not search upward through that gate or net.

In addition to not expanding the width of the bit-field, this algorithm has several other advantages over the cycle-breaking algorithm. First, 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. From this we conclude that any net that does not fan out will not require a shift, and that any fanout-free region of the circuit can be simulated without shifts. (For the purposes of this discussion, gates with more than one output are assumed to have fan-out, regardless of whether the individual outputs have fan-out.) This implies that if a circuit has no fan-out, the entire circuit can be simulated without shifts.

Another advantage of the path-tracing algorithm is that its results are independent of the order in which output nets are processed. This is in contrast to the cycle-breaking algorithm which is quite sensitive to the order of processing in the cycle-detection step. A third advantage, which will become clearer once code generation is discussed, is that the path-tracing algorithm generates only right shifts. The cycle-breaking algorithm can generate both right and left shifts. The path-tracing algorithm also tends to retain fewer shifts than the cycle-breaking algorithm for most circuits we have tried, although the cycle-breaking algorithm does better for some. For the circuit of Figure 7, the cycle-breaking algorithm will eliminate all shifts, while the path-tracing algorithm retains one shift. The cost of eliminating the extra shift is expansion of the bit-field. Figure 8 gives the number of shifts retained by the two algorithms for the ten ISCAS-85 benchmarks, while Figure 9 shows the bit-field width for the two algorithms on the same circuits. Note that the path-tracing algorithm *reduces* the width of the bit-field for some circuits.

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

Figure 8. Bit-Field Widths for Two Algorithms.

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

Figure 9. Retained Shifts for Two Algorithms.

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 associated with gate-simulations, however when nets may have differing alignments, it is necessary to associate the shifts with the fan-out branches of a net. As Figure 10 shows, the output of a gate might have to be shifted several times.

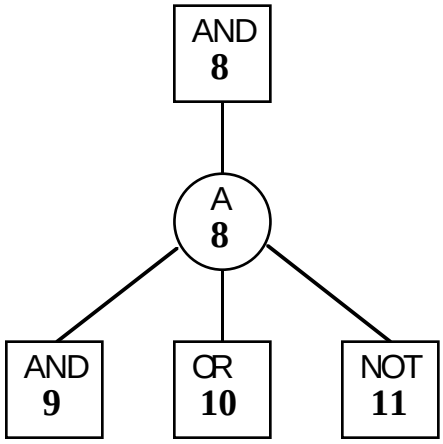

Figure 10. Multiple Shifts for a Single Net.

Because it is necessary to handle situations such as that pictured in Figure 10, shifts are done for 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. Because of this, no shift is required after the simulation is complete. Shifts may be to the right or to the left, and multiple-bit shifts may be required. (Multiple-bit shifts are just as efficient, in terms of instruction count, as single-bit shifts.) When a right shift is performed, the high-order bit of the bit-field is propagated using sign-extension.

However, for left shifts simple propagation of the low-order bit is not always possible. If the alignment of the net is equal to its minimum PC-set value, it is the value from the *previous input vector* that must be propagated through the low-order bits, rather than the current low-order bit value. Since this value is normally unavailable when the shift is being performed, it is necessary to guarantee that any net that is subjected to a left shift have an alignment that is strictly smaller than its minimum PC-set value. This emphasizes another advantage of the path-tracing algorithm over the cycle-breaking algorithm. Since the path-tracing algorithm generates only right-shifts, no special precautions for left-shifts are necessary.

As noted above, the net-initialization required by the unoptimized compiler is unnecessary when differing net-alignments are used, since the values from the previous vector are recomputed wherever necessary. Figure 11 contains the performance results of running both the path-tracing algorithm and the cycle-breaking algorithm on the ten ISCAS-85 benchmark circuits. The results of Figure 11 were obtained using the same method as those for Figure 2.

	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

Figure 11. Performance of The Path-Tracing and the Cycle-Breaking Algorithms

The results of Figure 11 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 Figure 11, 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 as

the results of Figure 8 show, the run times for these circuits would be unacceptably large, so it is not worth the effort to insert this fix.

5. The Combined Approach.

The results of the last section 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. Elimination of shift operations and elimination of simulation code for gaps is identical to that for the unoptimized compiler. Figure 12 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

Figure 12. 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. Conclusions.

Although the parallel technique already provides for very rapid unit-delay simulations with a reasonable amount of generated code, there are several opportunities for optimization. The two general optimization methods presented here are called bit-field trimming and shift elimination. The two methods of shift elimination discussed are called

path-tracing and cycle-breaking. By far the most effective technique is path-tracing, because it does not expand the width of the bit-field.

When bit-field trimming is used in isolation, it provides an average performance increase of 26% while the path-tracing algorithm used in isolation provides an average performance increase of 43%. Combining the two optimizations provides an average performance increase of 47% for the circuits tested.

These figures show that the optimization techniques presented here provide significant performance improvements in the parallel technique. There may be opportunities for additional performance improvements by performing a more extensive analysis of the PC-sets of a circuit. One opportunity might be to use differing width bit-fields for different portions of a circuit. Another might be to use a non-uniform time-scale for some portions of the circuit. For example, suppose the circuit-segment shown in Figure 13 has the PC-sets shown.

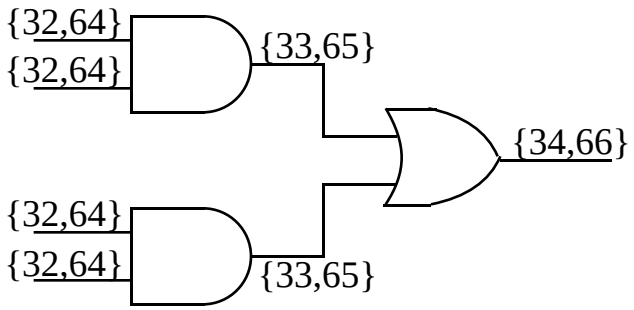

Figure 13. A Circuit Amenable to Bit-Field Compression.

Although the PC-sets illustrated in Figure 13 will occupy 2 32-bit words, there are only 3 significant values that must be computed, counting the final value from the previous input vector. If these bits were compressed into a single 32-bit word, half of the simulation code could be eliminated. Of course, additional code would be required for reformatting bit-fields, but if the reformatting code were small in relation to the size of the circuit-segment, significant savings could be realized.

In any case, the optimization techniques presented here provide an effective means for drastically improving the performance of the parallel technique of unit-delay compiled simulation.

REFERENCES

1. Wang, L., 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.

2. Bryant, R. E., 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.

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

4. Barzilai, Z., 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.

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

6. Wang, Z. and Maurer, P. M., "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," *proceeding of ICCAD-89.*

8. Melvin A. Breuer, Arthur D. Friedman, "Diagnosis and Reliable Design of Digital Systems," Computer Science Press Inc., Rockville MD, 1976.

9. P. Maurer, Z. Wang, "Techniques for Unit-Delay Compiled Simulation,", *Proceedings of the 27th Design Automation Conference*, to appear June 1990.

10. Chappell, S. G., H. Y. Chang, C. H. Elmendorf and L. D. Schmidt, "Comparison of Parallel and Deductive Simulation Techniques," IEEE Transactions on Computers, Vol C-23, pp. 1132-1139.

11. F. Brglez, P. Pownall, R. Hum, "Accelerated ATPG and Fault Grading via Testability Analysis," Proc ISCAS-85, pp. 695-698.