Scheduling Blocks for Hierarchical Compiled Simulation

Peter M. Maurer

Technical Report DA-23, 1991
VCAPP Laboratory
Dept. of Computer Sci & Eng.
University of South Florida
Tampa, Florida 33620

SCHEDULING BLOCKS FOR HIERARCHICAL COMPILED SIMULATION

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

ABSTRACT

Although preserving the hierarchy in compiled simulation can significantly reduce the compilation time for the code generated by the circuit compiler, the possibility of introducing pseudo-cycles due to element grouping can impair the performance of the generated code. A new approach to this problem is presented which uses dependency information to reduce the number of times a particular block must be simulated. The problem of determining the minimum schedule is shown to be NP-Complete, and a set of heuristics for the problem is presented. Experimental results for different combinations of these heuristics are presented. An algorithm for determining dependency information from the contents of a block is presented along with a new approach that can be used when the content of one or more blocks is unknown.

SCHEDULING BLOCKS FOR HIERARCHICAL COMPILED SIMULATION

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

1. Introduction.

Recently there has been a resurgence of interest in compiled simulation at various levels of the design hierarchy [1-6]. The primary motivation of this research has been to produce simulators that are much faster than interpretive event-driven simulators. However, a common problem in producing a compiled simulation is the amount of time required to compile the simulator once it has been generated. Typically 80% or more of the time required to create a simulator is spent compiling the generated code. One technique that has been used to reduce compilation time is to preserve the hierarchical structure of the circuit during the compilation process, by treating each sub-circuit as a sub-routine [3,4]. This technique can reduce the compilation time of circuits that have many replicated blocks of the same type, and can reduce recompilation time through incremental recompilation. Another benefit of this technique is that it simplifies the process of multi-level simulation by allowing subroutines produced by many different compilers to be combined into a single simulator [7].

Unfortunately, preserving the circuit hierarchy can complicate the process of code generation by introducing pseudo-cycles into circuits that would be acyclic if the hierarchy were removed. Figure 1 gives an example of such a circuit. For circuits such as that pictured in Figure 1, the levelized scheduling technique used by most compilers will fail even though the circuit is purely combinational.

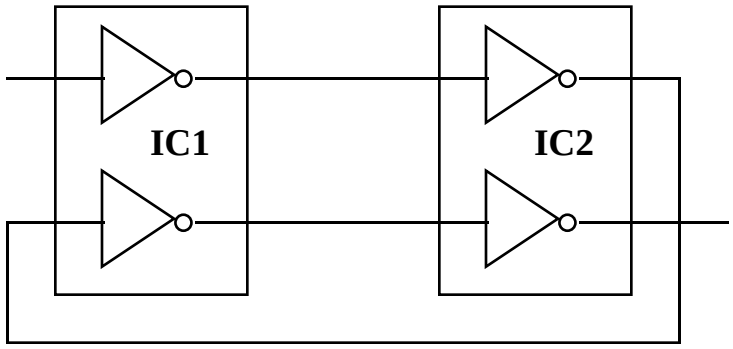

Figure 1. A Cyclic Purely Combinational Circuit.

One solution to the problem of pseudo-cycles was given in [4]. In this solution a circuit is simulated repeatedly until no net changes value. This solution is "blind" in the sense that it assumes no knowledge of the contents of a block, and has a worst-case running time of N^2, where N is the number of gates in the circuit. A more sophisticated solution was presented in [6], but it is applicable only to circuits without fanin or fanout, which makes it unusable in practice. The solutions presented here are adaptations and extensions of the work presented in [6]. They are applicable to all combinational circuits, and can easily be extended to handle synchronous sequential circuits. The most sophisticated technique presented here assumes some knowledge of a block's contents, but we also present a "blind" technique that improves on the N^2 worst-case time bound mentioned above.

Section 2 presents definitions and derives some preliminary bounds on schedule lengths. This section also proves that obtaining a minimum-length schedule for a circuit containing pseudo-cycles is NP-Complete. Section 3 introduces a collection of heuristics that can be used to attack the NP-Complete scheduling problem. Section 4 presents experimental results obtained from applying the heuristics of Section 3. Section 5 presents an algorithm for computing the dependency information used by the heuristics of Section 3. Section 6 presents techniques to be used when no knowledge of a block's contents is available, and Section 7 draws conclusions.

2. Definitions and Preliminary Bounds.

To simplify the presentation, a circuit is assumed to consist of a set of *blocks*, and a set of *nets*. Furthermore, each block is assumed to consist of a collection of ordinary gates. The techniques presented here can easily be

extended to handle circuits that contain a mixture of blocks and ordinary gates, and circuits that have a deeply nested hierarchy. Once the circuit has been compiled, each net will be represented by a variable containing the value of the net, and each functional block will represented by a subroutine that simulates the block. A single subroutine call is called a *simulation step*, and sequence of simulation steps that simulates each block in the circuit exactly once is called a *simulation pass* .

A block A is an *immediate predecessor* of block B if there is a net x which is an output of block A and an input of block B. Block A is a *predecessor* of block B if A is an immediate predecessor of B, or if A is an immediate predecessor of a block C which is a predecessor of B. A circuit is *acyclic* if no block is a predecessor of itself and *cyclic* otherwise.

Since in general we will be dealing with cyclic circuits, we introduce the concept of net dependencies to facilitate the handling of circuits containing pseudo-cycles. Let A be a functional block with an input x and an output y. The output y is *immediately dependent* on the input x, if the value of x is needed to compute the value of y. More formally, let A have n inputs and m outputs, and let x and y be the ith input and the jth output respectively. Suppose there exist two n-element vectors a and b such that $a_i \neq b_i$, but $a_k = b_k$ whenever $k \neq i$. Suppose further that c and d are the m-element vectors obtained from simulating block A with a and b respectively, and that $c_j \neq d_j$. Then y is *immediately dependent* on x.

For any two nets x and y, the net y is *dependent* on x if y is immediately dependent on x or if y is immediately dependent on some net z and z is dependent on x. A circuit is *purely combinational* if no net is dependent on itself. A circuit that is both cyclic and purely combinational is said to be *pseudo-cyclic*. A block A is (*immediately*) *dependent* on block B if x is an output of A and y is an output of B and x is (*immediately*) *dependent* on y. In this case, block A is (*immediately*) *dependent* on net y, and net x is (*immediately*) *dependent* on block B.

It is well known that an N-block acyclic circuit can be simulated using N simulation steps by processing the blocks in levelized order. It has also been shown in [4] that a circuit containing N *gates* can be simulated in N simulation *passes*, regardless of the order of the simulation steps. Even if the number of gates in each block is unknown, the maximum number of simulation passes can be calculated using the number of block outputs. Let C

be a pseudo-cyclic circuit containing N functional blocks. Let n_i be the number of outputs of the ith functional block, $1<i<N$. By using an argument similar to that given in [4] one can show that C can be simulated using $T=\sum_{i=1}^{N} n_i$ simulation passes.

Although this formula places an upper limit on the number of simulation steps required to obtain correct outputs for any combinational circuit, the worst-case time bound unacceptably large. Theorem 1, which is based on the concept of levelization with respect to dependencies, shows that a much smaller number of simulation steps may be sufficient. To levelize with respect to dependencies, we assign levels to nets in the following manner. If y is a primary input of a circuit C, then the level of y is 0. If y is not a primary input, then the level of y is L+1 where L is the maximum level of the nets upon which y depends. (This concept is essentially the same as the usual concept of level, except that within a functional block there may be many gates on the path from a level L input to a level L+1 output.)

<u>Theorem 1:</u> *Let C be a (possibly cyclic) purely combinational circuit containing N functional blocks. Let* k_i *be the number of distinct levels of the outputs of the ith functional block,* $1<i<N$. *The outputs of C will be correct after* $K=\sum_{i=1}^{N} k_i$ *simulation steps.*

<u>Proof:</u> Suppose that a block A has an output x, and that all inputs to A upon which x is immediately dependent are correct. Since the circuit is purely combinational, simulating A on these inputs will produce a correct value for x. This observation leads to the following procedure. First simulate all blocks that have outputs of level 1. This will cause all nets of level 1 to be correct. Next simulate all blocks that have outputs of level 2 which will cause all outputs of level 2 to be correct. Repeat this procedure until all outputs are correct. This procedure requires K simulation steps.■

This procedure is essentially the same as levelized scheduling, but a particular block may be scheduled many times. Unfortunately, this procedure is not always optimal, as the circuit of Figure 2 illustrates.

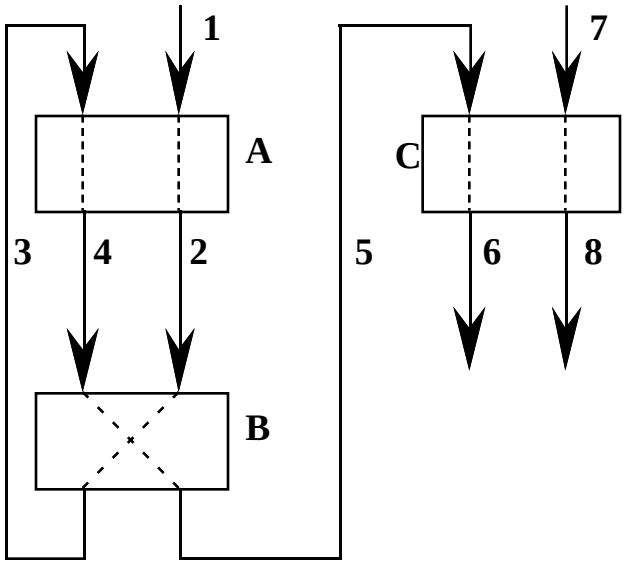

Figure 2. A Circuit for which Scheduling By Levels is Not Optimal.

For the circuit of Figure 2, levelized scheduling will produce the schedule ACBABC, but the optimal schedule is ABABC. It is easy to see why this schedule is optimal if we examine the graph of the relation "is immediately dependent on" as pictured in Figure 3. This graph is known as the *dependency graph* of the circuit. The vertices of this graph represent the nets of the circuit, while the arcs represent immediate dependencies between the nets. The arcs of the dependency graph are labeled with the blocks that generate the immediate dependencies. For purely combinational circuits the dependency graph is an acyclic directed graph with primary inputs as source nodes and primary outputs as sinks. It is obvious from Figure 3 that the dependency between nets 7 and 8 will be resolved by the schedule ABABC.

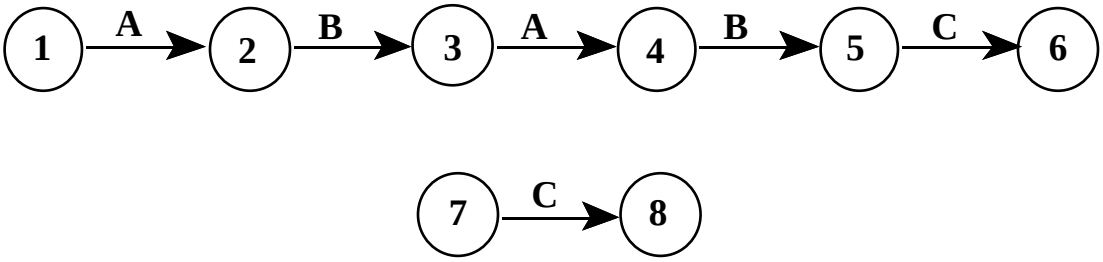

Figure 3. The dependency graph of a Circuit.

Although levelized scheduling does not produce an optimal schedule for this circuit, demand-driven scheduling, which proceeds backwards from the

primary outputs *will* produce the optimal schedule. However, for the circuit pictured in Figure 4, demand driven scheduling produces the schedule AABAC, while the optimal schedule is AABC.

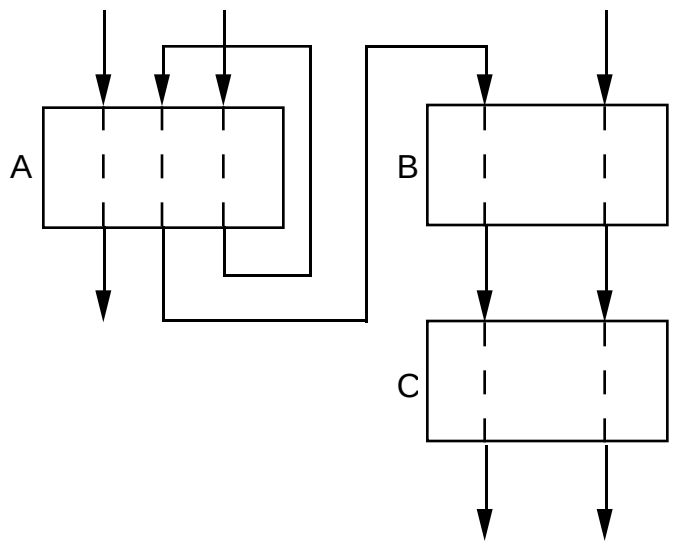

Figure 4. A Circuit for which Demand-Driven Scheduling is not Optimal.

Although the procedures for obtaining the minimal schedules for the circuits of Figures 2 and 4 are obvious, it is not immediately clear that these procedures will work efficiently for all circuits. To determine whether an efficient optimal algorithm exists for all circuits it is necessary to determine the computational complexity of obtaining a minimal schedule for an arbitrary circuit *C*. An analysis of the dependency graph will be used for this purpose.

We begin by establishing a relationship between the dependency graph, schedules, and sets of strings. For a given circuit *C*, the *alphabet of C* is taken to be the set of blocks of *C*. Suppose *x* is a primary input and *y* is a primary output of *C*, and that there is a path *p* from *x* to *y* in the dependency graph of *C* with arc labels $A_1, A_2, \ldots, A_n$. This sequence, which is a string on the alphabet of *C*, is called *the string generated by p*. Let Q be the set of all paths from primary inputs to primary outputs in the dependency graph of C. S(C) is the set of all strings generated by the paths of Q. Any schedule for simulating *C* is also a string on the alphabet of *C*. Let $A = A_1, A_2, \ldots, A_n$ and $B = B_1, B_2, \ldots, B_m$ be two strings. The string *B* is a *supersequence* of the string *A* if *B* contains all characters of *A* in the proper sequence. The characters need not

be contiguous in B. Thus "acacbca" is a supersequence of "aab", while "abaa" is not. If S is a set of strings, B is a *common supersequence* of S if B is a supersequence of every member of S. The following theorem shows the relationship between S(C) and the schedules that correctly compute the outputs of C.

<u>Theorem 2:</u> *Let C be a (possibly cyclic) purely combinational circuit containing N functional blocks. Let* $B=B_1, B_2, \ldots, B_m$ *be a schedule for simulating C. B correctly computes the value of all primary outputs of C if and only if B is a common supersequence of S(C).*

<u>Proof.</u> First, suppose B is a common supersequence of S(C). We will proceed by (complete) induction on the depth (maximum level) of C. If C is of level 1, S(C) is a set of single-character strings. Since every block must have at least one input and one output, S(C) contains every block of C. Since B is a common supersequence of S(C), B simulates every block of C at least once, which correctly computes the value of all level 1 outputs. Since all primary outputs are of level 1, B computes their values correctly. Now assume that C is of depth N, and let y be an arbitrary level-N primary output of C. Let y be an output of the block A, and let C_y be the circuit obtained by deleting from C the blocks and nets upon which y does not depend. Let k be the largest integer such that B_k is equal to A, and let $B' = B_1, B_2, \ldots, B_k$. Since B is a common supersequence of S(C), B' must be a common supersequence of S(C_y). Let C_y be the circuit obtained by deleting the output y from C_y. C_y is of depth N-1. Let $B'' = B_1, B_2, \ldots, B_{k-1}$. Since B' is a common supersequence of S(C_y), B'' must be a common supersequence of S(C_y). By the inductive hypothesis, B'' computes the values of the primary outputs of C_y correctly. But the primary outputs of C_y are precisely those nets of C upon which y is immediately dependent. Therefore, when the schedule B' is executed, the nets upon which y is immediately dependent will be correct before $B_k=A$ is executed, which implies that y will be correct after B_k is executed. Thus B' computes y correctly, and by inclusion, B computes y correctly.

To prove the converse, assume that C is of level 1, and that B computes C correctly. B must contain every block of C, and since S(C) is a set of

one-character strings, one for each block of C, B is a common supersequence of S(C). As before let C be of depth N, let y be an arbitrary primary output of level N, and let y be an output of block A. Since B correctly simulates C, B must contain at least one occurrence of A. Let C_y, C_y B' and B'' be as before. Since B' correctly simulates C_y, the nets upon which y is immediately dependent must be correct before $B_k = A$ is executed. This implies that B'' correctly simulates C_y. Since C_y is of level N-1 we conclude by induction that B'' is a common supersequence of S(C_y). One may obtain S(C_y) from S(C_y) by appending an A to every element of S(C_y). Since B' ends with A, and since B'' is a common supersequence of S(C_y), B' is a common supersequence of S(C_y). By inclusion, B is a common supersequence of S(C_y) for every output y. Since every element of S(C) is contained in S(C_y) for some output y, B is a common supersequence of S(C).■

<u>Corollary:</u> *The optimal schedule for a circuit C is the Shortest Common Supersequence (SCS) of S(C).*

Although Theorem 2 suggests a procedure for obtaining the minimal schedule of an arbitrary circuit, there are two difficulties. First, the Shortest Common Supersequence problem is known to be NP-Complete [8], and second, there exist circuits for which the size of S(C) is an exponential function of the number of blocks in C. The question yet to be resolved is whether there is a more efficient method for computing the minimal schedule of C than obtaining the SCS of S(C). Theorem 3 shows that the answer is no (unless P=NP).

<u>Theorem 3:</u> *Finding the optimal schedule for a (possibly cyclic) purely combinational circuit is NP-complete.*

<u>Proof</u>: Given a set of strings Q on some alphabet Γ, construct a circuit C containing at exactly P paths from primary inputs to primary outputs where P is the number of strings in Q. First, create one block for each distinct character of Γ that appears in an element of Q. For each block, count the number of occurrences of the corresponding character in all members of Q. Assume that the character a has n_a occurrences. Place

n_a inputs and n_a outputs on the block corresponding to a, and place n_a NOT gates inside of the block. Connect the ith input to the ith output through the ith NOT gate. For each string S in Q, traverse the string from left to right. When each character is encountered choose the first unused input and its corresponding output from the block corresponding to the character. If this is not the first character in the string, connect the output chosen in the previous step to the input chosen in this step. Then mark the input and output chosen in this step as "used." This creates a circuit C such that S(C)=Q. This shows that finding the optimal schedule is NP-Hard. Proving that the problem can be solved by a nondeterministic polynomial-time bounded algorithm is a simple exercise and left to the reader.■

This result shows that any algorithm capable of producing an optimal schedule for an arbitrary circuit C, will be intractable unless P=NP. Furthermore, the circuits created in the proof of Theorem 3 are extremely simple, so placing restrictions on circuit complexity will be of no help. Therefore, it will be necessary to consider approximation algorithms for the optimal scheduling problem.

3. The Approximation Algorithms.

The solutions to the scheduling problem that appear in [6] are approximation algorithms for the SCS problem. Most of the algorithms presented in this section are adaptations of one of these algorithms. Because a circuit may have an exponential number of paths from primary inputs to primary outputs, it is not possible to adapt all SCS approximation algorithms to general acyclic circuits. Generally speaking, those algorithms that operate on a fixed-size prefix of the string are adaptable, but those that operate on the entire string are not.

To see why this is so, we must first eliminate two possibilities. First, it may be possible that S(C) is of a reasonable size even though the number of paths through the network is exponential in size. That is, one string of S(C) may represent many paths. The second possibility is that there may be some subset Q of S(C) such that every common supersequence of Q is also a common supersequence of S(C). Neither of these statements is true for the circuit pictured in Figure 5. The size of S(C) is an exponential function of n,

whereas the number of blocks of C is a linear function of n. Furthermore, eliminating even one string from S(C) will produce a set that has at least one common supersequence that is not a common supersequence of S(C).

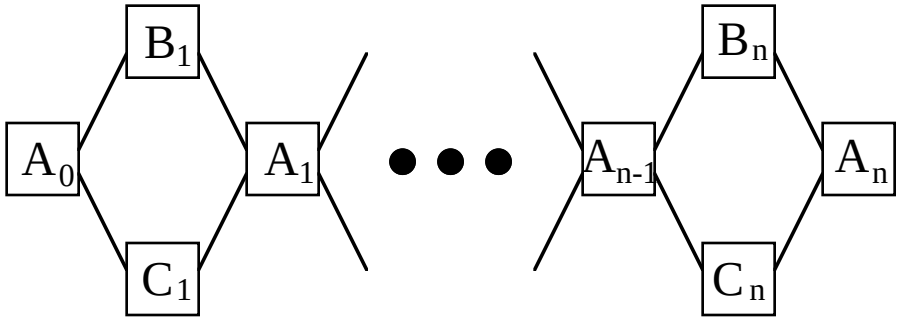

Figure 5. A Circuit with an Exponential Number of Paths.

Although the dependency graph presented in the previous section could be used to schedule the circuit, we have found it more convenient to use a special circuit diagram called the *partitioned* diagram of the circuit. In the partitioned diagram, each block A with n outputs is partitioned into several smaller blocks labeled A_1 through A_n. The inputs to A_i are precisely those inputs upon which output i depends. If two blocks A_i and A_j have precisely the same inputs, output j is added to block A_i, and block A_j is removed. The blocks A_1 through A_n are called the *sub-blocks* of A in the partitioned diagram of the circuit. Note that the outputs of a sub-block are all at the same level.

Since we are dealing only with purely combinational circuits, the partitioned diagram of the circuit is guaranteed to be acyclic and can be scheduled in a manner that is reminiscent of algorithms for generating levelized compiled code. Although our approximation algorithms operate on sub-blocks, blocks cannot be physically partitioned, and must be scheduled in their entirety. Blocks are scheduled one at a time using a list of candidates which initially contains all level-0 sub-blocks from the partitioned diagram. At any given time, a particular block may have many sub-blocks in the candidate list. When a block is scheduled, all sub-blocks for the block are removed from the candidate list and marked as scheduled. The algorithms maintain a second list, called the successor list, which contains those sub-blocks that will enter the candidate list after one of the blocks currently

represented in the candidate list has been scheduled. A sub-block is permitted to enter the candidate list only after all of its predecessor sub-blocks have been scheduled. A sub-block appears in the successor list only if all of its unscheduled predecessor sub-blocks appear in the candidate list *and* all of the unscheduled predecessors are sub-blocks of the same block. The algorithms also keep track of the number of unscheduled sub-blocks for each block.

A block is called a *prime candidate* if it contains a sub-blocks in the candidate list, but not in the successor list. The *candidate count* of a block is the number of sub-blocks of the block appearing in the candidate list. The *successor count* of a block is the number unscheduled sub-blocks, exclusive of those appearing in the candidate list.

The schedule will tend to be more efficient if blocks with high candidate counts are chosen, because this schedules many sub-blocks simultaneously. Everything else being equal, most of the approximation algorithms will schedule blocks with high candidate counts first. Rather than blindly choosing the block with the highest candidate count, most of the algorithms attempt to make its choices in such a way as to maximize the candidate counts for the blocks that have *not* been chosen. Prime candidates are scheduled first, because scheduling a single block cannot increase the candidate count of a prime candidate. If a non-prime candidate is chosen, then that block is *guaranteed* to appear at least twice in the final schedule.

Because it is generally impossible to predict how well an approximation algorithm will perform on real data without actually trying it, we have implemented 14 different algorithms with different combinations of heuristics. The most sophisticated of these are based on the following set of heuristics.

1. If there are no prime candidates, treat every block with sub-blocks in the candidate list as if it were a prime candidate. In this case, there is at least one block in the candidate list that will be scheduled twice.
2. If there is a single prime candidate, schedule it.
3. If there is a prime candidate with a successor count of zero, schedule it.
4. Select the prime candidate(s) with the maximum candidate count. This heuristic is applied only after applying heuristics 1-3.
5. Select the prime candidate(s) with the minimum successor count. This heuristic is usually applied to those candidates selected by heuristic 4.

6. Determine the effect that scheduling the candidate will have on the candidate counts of the other blocks. Select a block for which the increase in candidate counts is at a maximum. This heuristic is usually applied to those candidates selected by heuristic 5.

These heuristics were adapted from those for the algorithm SCSTEST1 presented in [6]. The effect of scheduling a candidate on the candidate counts of other blocks is determined as follows. For each block with representatives in the candidate list, the number of immediate successors of the block in the successor list is counted. This is the number of sub-blocks that will enter the candidate list if the block is scheduled. The block that has the most effect on the candidate counts of other blocks is the block for which this count is at a maximum.

4. Experimental Results.

Several experiments were run to determine the effectiveness of the heuristics described in the previous section. The basic circuit used for these experiments was an 8x8 array multiplier constructed from half and full adders. The half and full adders were expanded into AND, OR, NOT, and XOR gates before further processing. The circuit contained a total of 368 gates. In the first set of experiments, the gates were randomly partitioned into 2, 4, 8, 16, and 32 blocks of roughly equal size. Five different partitions were generated for each number of blocks, for a total of 25 test circuits. The procedure for generating the partition is as follows. This procedure does not create self loops, nor does it create more than one instance of any particular block.

1. For each gate in the original circuit assign a block-id at random from a set of appropriate size (2, 4, 8, 16, or 32).
2. For each block-id that has been assigned to some gate, create a subnetwork and place the gates with the corresponding id's into the subnetwork.
3. For each subnetwork, find all nets that are primary inputs of the original circuit, or used but not generated in the subnetwork, and place them in the primary input list. Also determine all nets that are primary outputs of the original circuit or generated in the subnetwork and used elsewhere and place them in the primary output list.

4. Once all gates have been removed from the original circuit, insert a subnetwork call to each of the new subnetworks.

Each of the circuits was scheduled using the 14 different algorithms to compare the effectiveness of the heuristics given in the last section, as well as other strategies. With the exception of the LIFO algorithm, all algorithms implement the candidate list as a first-in-first-out queue. The queue is initialized to those sub-blocks that have only level zero inputs. Each sub-block contains a count of all unscheduled immediate predecessors. When a sub-block is scheduled, the counts of all immediate successors are decremented and any block whose count has reached zero is added to the tail of the queue. Whenever a block is scheduled, all sub-blocks in the queue that correspond to the block are removed and marked as scheduled. Figure 6 lists the algorithms used and the scheduling strategy used by each. When the heuristics of an algorithm permit the choice of more than one block, the first applicable sub-block in the queue is chosen.

Algorithm Name	Strategy
FIQ	Select the first sub-block in the queue.
LIQ	Select the block with the largest candidate count.
PCSF	Use heuristics 1-4 *only*.
FS	Use heuristics 1-5 *only*.
FSME	Use all heuristics.
PCLF	Use heuristics 1-4, but place all immediate successors of the sub-blocks in the candidate list into the successor list.
XS	Use heuristics 1-5 but place all immediate successors of the sub-blocks in the candidate list into the successor list.
XSME	Use all heuristics but place all immediate successors of the blocks in the candidate queue into the successor list.
AS	Use heuristics 1-5 but treat all blocks as prime candidates.
ASME	Use all heuristics but treat all blocks as prime candidates.
FSO	Select the block with the least number of successors, ignoring all other heuristics.
MEO	Select the block with the largest effect on the candidate counts of other blocks, ignoring other heuristics.
LIFO	Same as FIQ, but use a first-in-last-out queue.

Figure 6. The Approximation Algorithms.

The rationale for testing each of these algorithms is as follows. First in Queue (FIQ) tends to schedule those blocks that are closest to the primary inputs first. This is contrasted with the Last In First Out algorithm (LIFO) which tends to trace paths from the primary inputs to the primary outputs. Largest in Queue (LIQ) demonstrates the relative effectiveness of simply choosing the block with the largest candidate count. The PCSF algorithm demonstrates the relative effectiveness of confining the search for the block with the largest candidate count to prime candidate counts. The FS and FSME algorithms demonstrate the effectiveness of adding heuristics 5 and 6 to the PCSF algorithm. The method used by algorithms PCSF, FS, and FSME to obtain the successor list is fairly complicated. The algorithms must pretend to schedule a block and then determine which new sub-blocks can be added to

the candidate list. The algorithms PCLF, XS, and XSME are identical to PCSF, FS, and FSME respectively, but they use a simpler, less accurate algorithm to obtain the successor list. This algorithm simply adds all immediate successors of the sub-blocks in the candidate list to the successor list, without regard to whether they could actually be placed in the candidate list after scheduling one block. These algorithms determine the relative effectiveness of the more complex algorithm.

The algorithms AS and ASME treat all blocks with representatives in the candidate list as prime candidates, but in all other respects are identical to FS and FSME. This has the effect of using heuristics 5 and 6 while ignoring heuristics 1-4. Thus these algorithms demonstrate the relative effectiveness of confining the search to prime candidates. Ignoring heuristics 1-4 in the PCSF algorithm results in the LIQ algorithm, hence no separate algorithm was constructed for this strategy. Finally the algorithms FSO and MEO demonstrate the relative effectiveness of heuristics 5 and 6 used in isolation.

To further determine the effectiveness of each algorithm, an estimate of the minimum number of blocks to be scheduled for each circuit was computed. This computation uses the concept of a block's *winding number*. The winding number for a block *A* is computed by assigning a winding number to every net in the partitioned diagram circuit. Primary inputs are assigned a winding number of zero, and sub-blocks are processed in levelized order. When a sub-block is processed, the maximum of the winding numbers of the inputs is obtained, and if the sub-block is a sub-block of block *A*, the number is incremented by 1, otherwise it is left unchanged. This new number is then assigned to all outputs of the sub-block as their winding numbers. Once all sub-blocks have been processed, the maximum of the winding numbers of the primary outputs is assigned to the block as its winding number. This computation must be repeated for each block in the circuit. The winding number of a block *A* is the maximum number of times *A* appears on any path between the primary inputs and the primary outputs. The block must appear at least this many times in any schedule for the circuit. An estimate of the minimum length schedule can be obtained by forming the sum of the winding numbers of all blocks. Any schedule for the circuit must be at least this long, but as Figure 7 illustrates, this is not necessarily the minimum length schedule. In Figure 7 the total of the winding numbers is 3, but the minimum length schedule is 4.

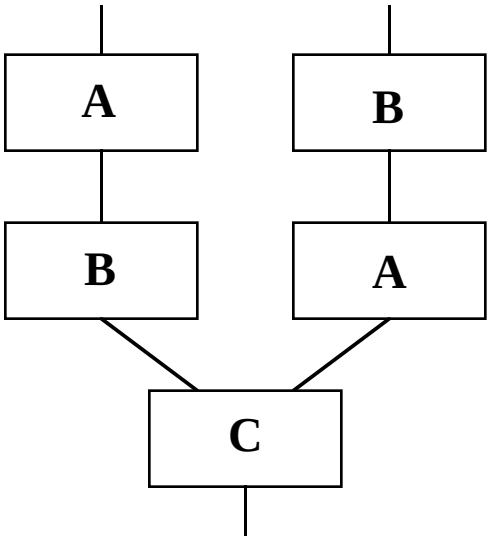

Figure 7. Winding Number ≠ Minimum Length Schedule

Figure 8 presents the results of the first set of experiments. To simplify the presentation, the total of the schedule lengths for all 25 circuits is shown, rather than the schedule length for each circuit. Results for the individual circuits can be found in [10]. The bottom item in Figure 8 is the total of the winding numbers for all 25 circuits.

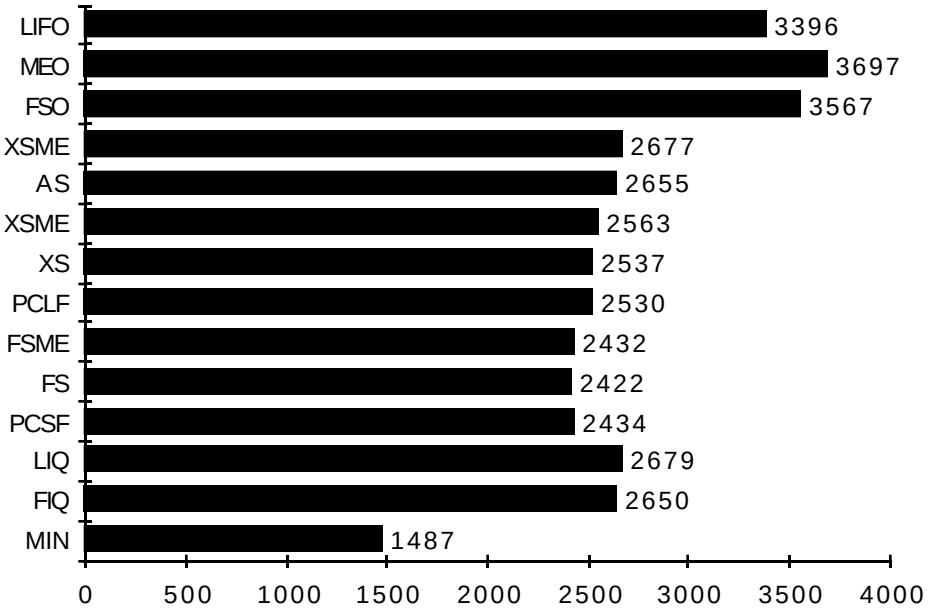

Figure 8. The Results from the First Set Of Experiments.

The second set of experiments uses the same base circuit, but with blocks of differing sizes. To create these circuits, the procedure for assigning block id's was altered to weight some choices more heavily than others. Again, the circuit was split into 2, 4, 8, 16, and 32 blocks, with five partitions being made for each number of blocks. For 2-block partitions, the weights for the two blocks were 1 and 2, creating one block approximately twice the size of the other. For 4-block partitions the weights were 1, 2, 4, and 8, while for 8-block partitions the weights were 1, 2, 4, 8, 16, 32, 64, and 128. The 16 and 32-block partitions used the same set of weights as the 8-block partitions, but with 2 and 4 blocks of each weight respectively. The results for the second set of experiments are given in Figure 9.

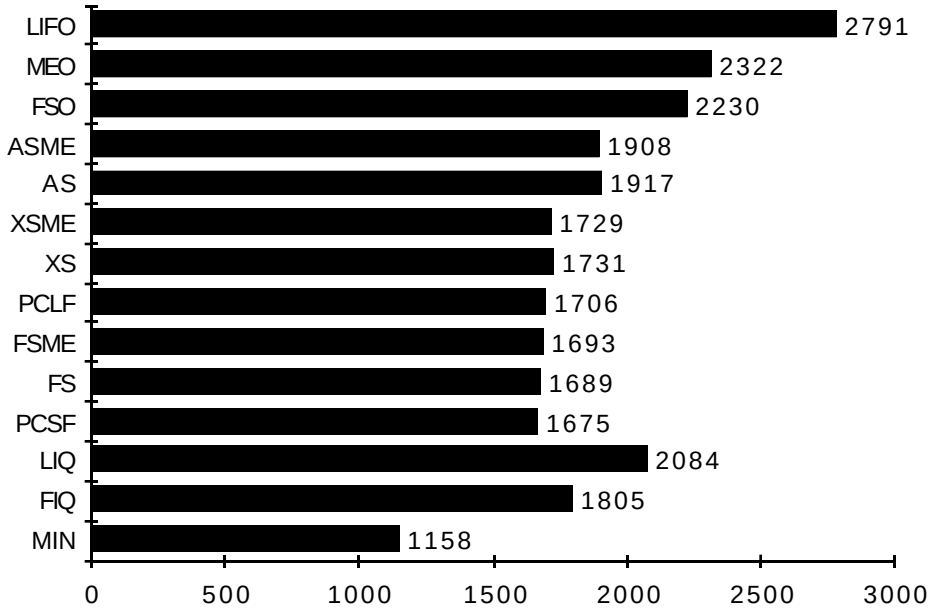

Figure 9. The Results from the Second Set of Experiments.

For the first set of experiments, FS (Heuristics 1-5) gave the best results, while for the second set, PCSF (Heuristics 1-4) gave the best results. In all cases, those heuristics that confined their selections to prime candidates outperformed those that did not. Furthermore, when computing prime candidates, restricting the successor list to only those items that could be scheduled after scheduling one prime candidate proved to be beneficial. Heuristic 6, selecting the block that had the most effect on the successor counts of other blocks in the candidate list, proved to be less beneficial than simply choosing the first item in the queue (after applying other heuristics). It is important to note that selecting the first item in the queue from a list of candidates is different from making a random choice from the set, since selecting the first item in the queue will tend to select items that are closer to the primary inputs. Finally, the algorithms FSO (least number of successors only), MEO (greatest effect on other candidate counts only) and LIFO (last in first out) performed significantly worse than the simplest strategy, FIQ (first in queue). Surprisingly, LIQ (largest candidate count in queue) also performed worse than FIQ.

It is important to note that the experimental data used in this section was designed to test the effectiveness of the algorithms, not to provide an accurate model of real circuits. In practice, most circuits will be far easier to schedule than the examples provided here.

5. Calculating Dependencies Within a Block.

The ability to determine the immediate dependencies of block outputs is central to the scheduling algorithms presented in the previous section. To determine these dependencies, it is necessary to have a knowledge of the contents of each block. This section presents an algorithm for determining dependencies, assuming that blocks are not nested. The dependencies for nested blocks can be obtained by applying this procedure recursively in an obvious way.

Given a block with *n* inputs, the algorithm begins by assigning an *n*-bit bit-field to each net in the block. Each bit in a particular field corresponds to a single primary input. The fields of the primary inputs are initialized with a one in the position that corresponds to the input and zeros in all other positions. The values of the remaining bit-fields (which are initially zero) are computed by processing gates in levelized order. (Note that the block can be assumed to be acyclic.) When a gate is processed, the bit-fields of the inputs are ORed together in bitwise fashion and the result is ORed into the bit-field of the gate's outputs. Once all gates have been processed, the bit-field of each primary output contains the dependency information for the output. This information is then used to compute the partitioned network of the circuit.

It is interesting to note the similarity between the algorithm for computing dependencies and algorithms used to certify the correctness of programs for handling classified information under a dynamic binding model [9]. Such algorithms compute the information flow through a high-level program and assign security classes based on a lattice-maximum operation. In the algorithm for computing dependencies, the bit-fields take the place of security classes and the bitwise OR operation takes the place of the "maximum" operation. It is probable that the dependencies for blocks coded in high level languages could be computed using procedures similar to those presented in [9]. More work is needed in this area.

6. Scheduling without Dependencies.

In the rare case where dependency information is unobtainable, a less efficient scheduling technique must be used. As was pointed out in [4] one can guarantee the correctness of the primary outputs of any N-gate circuit by completing N simulation passes. Because the time-bound of this algorithm is N^2, and because circuits seldom require this much computation time in practice, it is necessary to test for changes in the circuit after each pass. As noted above, the number of gates, N, can be replaced by the total number of block outputs, T. If T is small, this time bound may be acceptable, but by doing additional analysis of the circuit, it may be possible to reduce the time bound considerably.

First, the circuit is processed in depth-first order starting with the primary inputs. This procedure is used to identify and mark all pseudo-feedback arcs in the circuit. The circuit is then levelized, ignoring the pseudo-feedback arcs. Next we observe that there must be at least one feedback arc that does not depend on any other feedback arc (otherwise the circuit would not be purely combinational). If blocks are simulated in levelized order, this arc will be correct after the first simulation pass. We may continue this argument to show that if there are M feedback arcs, they will all be correct after M simulation passes. In the worst case, a final simulation pass is required to propagate the values of the last feedback arc to the primary outputs. Thus the maximum number of simulation passes required is $M+1$, where M is the number of feedback arcs. Since in most cases we would expect the number of feedback arcs to be small, this may significantly reduce the time bound of the algorithm. If the number of feedback arcs is large, it may still be beneficial to test the circuit for changes after each simulation pass, but in this case only the values of the feedback arcs need to be tested.

For many of the circuits we have studied, it is possible to arrange the dependencies in such a way that the worst-case number of simulation passes will be required to obtain correct outputs. This implies that, for these circuits, the worst-case time bound cannot be reduced based on topological information alone. However there are some circuits for which the worst-case number of simulation passes is less than or equal to the number of feedback arcs, regardless of how the dependencies have been arranged. Figure 10 gives an example of such a circuit.

The circuit of Figure 10 contains 3 feedback arcs, but no more than 3 simulation passes will be required to produce correct outputs.

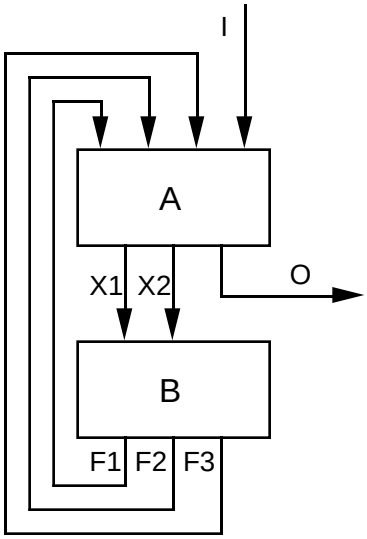

Figure 10. An Exception to The Feedback Rule.

In Figure 10, after the first simulation pass, at least one of X1 and X2 and at least one of F1 through F3 must be correct, otherwise there would be no way for all nets to obtain their correct values. Similarly, after the second pass, both X1 and X2 must be correct. Since all inputs to block B are correct, the outputs F1, F2, and F3 must also be correct. Therefore at the beginning of the third pass, all inputs to block A are correct and the output O will be correct after the third pass. To perform such analysis automatically may be quite complicated, and it is not clear that an efficient algorithm could be constructed to handle such cases. More work is needed in this area.

7. Conclusion.

This paper has presented several algorithms for scheduling high-level functional blocks under the assumptions that blocks may not be physically divided and that pseudo-cycles may be present due to the grouping of elements within blocks. These algorithms rely on dependency information derived from the block-definitions to create a logical (rather than physical) partitioning of the circuit. The partitioned network is then scheduled by an algorithm which is based on levelized scheduling with various heuristics added. Of these, the best algorithms appear to be those that are based on the

concept of prime candidates, which restricts the choice of the next scheduled block to those candidates that cannot be scheduled in the next round of candidate selection. This technique represents is an improvement over the technique of simulating blocks in random order, since no block will be scheduled unless there is a potential for performing useful work by simulating the block. The problem of finding the minimal schedule has been shown to be NP-Complete, hence a heuristic approach to the problem is justified.

A simple technique for computing the dependencies of a block has been presented, along with a set of techniques that can be applied when dependencies are impossible to compute. More work is needed in the area of computing dependencies for blocks written in high-level-languages, and in optimizing schedules when dependencies are not known. Nevertheless, the techniques presented here represent a significant improvement over existing techniques for hierarchical compiled simulation of function models.

8. Acknowledgement

Thanks are extended to Zhicheng Wang for pointing out the similarity between the problem of scheduling linear circuits and the Shortest Common Subsequence problem, and for providing the example of Figure 10.

REFERENCES

1. R. E. Bryant, D. Beatty, K. Brace, K. Cho, T. Sheffler, "COSMOS: A Compiled Simulator for MOS Circuits," *Proceedings of the 24th Design Automation Conference*, pp. 9-16, 1987.

2. L. Wang, N. Hoover, E. Porter, J. Zasio, "SSIM: A Software Levelized Compiled-Code Simulator," *Proceedings of the 24th Design Automation Conference*, pp. 2-8, 1987.

3. Z. Barzilai, J. L. Carter, B. K. Rosen J. D. Rutledge, "HSS–A High-Speed Simulator," *IEEE Transactions on Computer Aided Design*, Vol. CAD-6, No. 4, , pp. 601-617, July, 1987.

4. A. W. Appel, "Simulating Digital Circuits with One Bit Per Wire," *IEEE Transactions on Computer Aided Design*, Vol. CAD-7, pp. 987-993, Sept., 1988.

5. D. M. Lewis, "Hierarchical Compiled Event-Driven Logic Simulation," Proceedings of the 1989 International Conference on Computer Aided Design, to appear.

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

7. P. M. Maurer, Z. Wang, C. D. Morency, "Techniques for Multi-Level Compiled Simulation," Department of Computer Science and Engineering Technical Report CSE-89-04, University of South Florida, 1989.

8. D. Maier, "The Complexity of Some Problems on Subsequences and Supersequences," *Journal of the ACM*, Vol. 25, pp. 322-336, April, 1978.

9. D. E. R. Denning, *Secure Information Flow in Computer Systems*, Ph. D. Thesis, Purdue University, May 1975.

10. P. M. Maurer, "Raw Data for Experiments in Scheduling Blocks for Hierarchical Compiled Simulation," Department of Computer Science and Engineering, Technical Report CSE-89-21, University of South Florida, 1989.