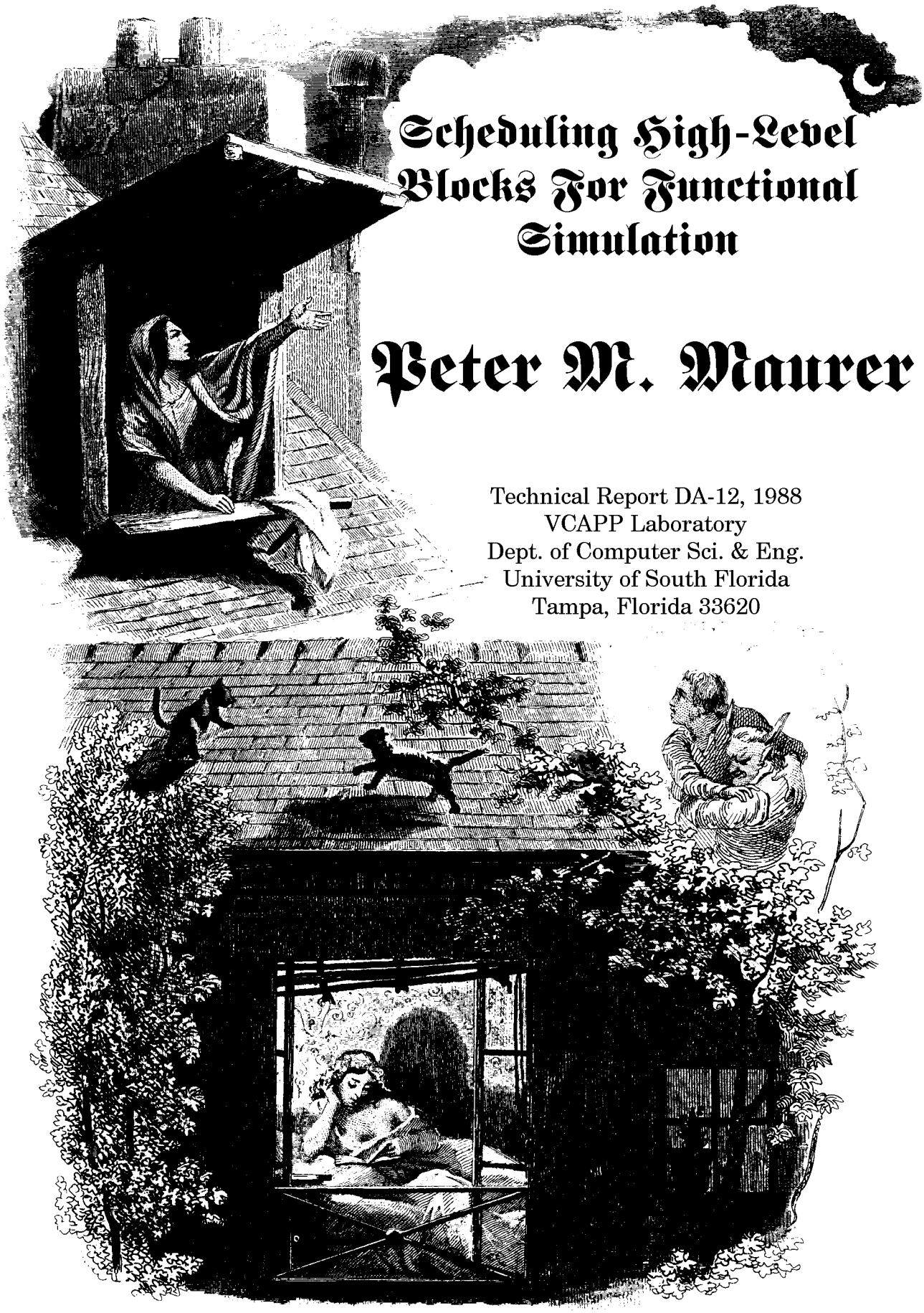
Scheduling High-Level Blocks For Functional Simulation

Peter M. Maurer

Technical Report DA-12, 1988
VCAPP Laboratory
Dept. of Computer Sci. & Eng.
University of South Florida
Tampa, Florida 33620



SCHEDULING HIGH-LEVEL BLOCKS FOR FUNCTIONAL SIMULATION*

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

ABSTRACT

This paper presents a method for scheduling high-level blocks for functional simulation under the assumptions that circuits may be cyclic (due to element grouping), and that blocks cannot be broken down into simpler elements. The solution presented here may simulate one block many times per clock period. Obtaining a minimal schedule for a cyclic circuit is shown to be NP-complete, and two approximation algorithms are presented, along with empirical data to evaluate their effectiveness.

1. Introduction

The aim of this research is to develop a functional-level simulation system for Wafer-Scale Integrated circuits (WSI), in support of a large project currently under way at the University of South Florida. We are especially interested in functional simulation because it is efficient and can be made accurate enough to detect most errors that are not electrical in nature. The various techniques used to construct functional models[1,2], give a wide range of trade-offs between speed and accuracy. Mixed-level simulation can be used to combine fast simulations of some components with accurate simulations of others [3].

Simulations can be categorized as interpreted or compiled depending on whether the circuit description is parsed into internal data structures and simulated by a separate program [4], or translated into a program that simulates the circuit directly [5]. Compiled simulations generally execute much faster than interpreted simulations, but require more compilation time.

Because WSI circuits can be many times larger than VLSI circuits, it is important that functional simulations be as efficient as possible. Thus we have concentrated on compiled simulations and HLL models. We are interested in circuits constructed from 1-5 block types of 10-20,000 gates each, with each block-type replicated many times. To minimize the size of the simulator, we implement each block type as a subroutine, which is called repeatedly during the simulation of the circuit. There are well-known methods for determining the order of function calls in acyclic circuits[5], but many of the circuits we wish to simulate contain cyclic dependencies of the type illustrated in Figure 1.

The usual practice is to remove these cycles by breaking one or more blocks into their individual components. However, this may not be possible for blocks implemented as HLL subroutines, and in any case can cause a huge increase storage requirements for WSI simulators. The remainder of this paper assumes that (1) circuits may be cyclic and (2) blocks may not be broken into individual components. All circuits are assumed to be purely combinational, but the results of this work may be easily extended to sequential circuits under the following assumptions. (1) Every cycle in the circuit contains at least one register or flip-flop. (2) The outputs of registers and flip-flops are treated as primary inputs, and their inputs are treated as primary outputs. (3) The state of each sequential element can change at most once per clock period. (4) Sequential elements are simulated in two phases, the first of which computes the next output value and the second of which makes the value visible to the rest of the circuit. Assumption number 4 is crucial, because a block may be simulated several times to obtain correct outputs. Phase 1 may be executed several times, after which phase 2 is executed once.

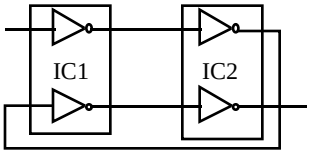


Figure 1. A Cyclic Purely Combinational Circuit.

2. Definitions and Preliminary Bounds.

A circuit consists of a set of *functional blocks*, and a set of *nets*. Each net is represented by a variable containing the value of the net, and each functional block is represented by a subroutine that simulates the block. A subroutine call is called a *simulation step*, and one simulation step for each block in the circuit 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.

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*. 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.

The following theorem shows that for acyclic circuits only *N* simulation steps are required to produce correct outputs, where *N* is the number of functional blocks in the circuit.

Theorem 1: *Let C be an acyclic circuit containing N functional blocks. The simulations of the N functional blocks can be scheduled so that only N simulation steps are required.*

The proofs of this and other theorems will be found in [6]. It can also be shown that the outputs of an acyclic circuit will be correct after N simulation *passes*, regardless of the sequence of the simulation steps within each pass. Theorem 2 obtains a similar result for (possibly) cyclic circuits.

Theorem 2: *Let C be a (possibly cyclic) combinational circuit containing N functional blocks. Let* n_i *be the number of outputs of the ith functional block,* $1<i<N$. *The outputs of C*

* This research was supported by the Defense Advanced Research Projects Agency under grant 2114-033-LO.

will be correct after $M=\sum_{i=1}^{N} n_i$ *simulation passes as long as the primary inputs are held constant.*

Although this theorem places an upper limit on the number of simulation steps required to obtain correct outputs for any combinational circuit, the limit is unacceptably large. Theorem 3, which is based on the concept of net-levels, shows that a much smaller number of simulation steps may be sufficient. If y is a primary input of a circuit C, then level(C)=0. If y is not a primary input, then level(y)=L+1 where L is the maximum level of the nets upon which y depends.

Theorem 3: *Let C be a (possibly cyclic) combinational circuit containing N functional blocks. Let* k_i *be the number of distinct values for level (x) where x is an output 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.*

One first schedules all blocks with level-1 outputs, then all those with level-2 outputs, and so forth. Blocks are scheduled in the same order as they are in event driven simulation.[4,7]. This procedure is not optimal as Figure 2 illustrates.

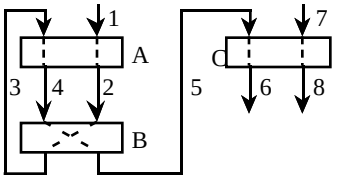


Figure 2. A Circuit for which Scheduling By Levels is Not Optimal.

For the circuit of Figure 2, scheduling by levels 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. The arcs of this dependency graph are labeled with the blocks that generate the immediate dependencies. It is obvious that the dependency between nets 7 and 8 will be resolved by the schedule ABABC.

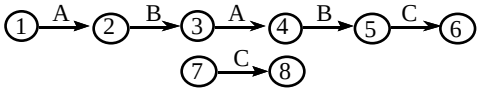


Figure 3. The dependency graph of a Circuit.

For purely combinational circuits the dependency graph is an acyclic directed graph with primary inputs as source nodes and primary outputs as sinks. Suppose x is a primary input and y is a primary output of some circuit C, and that there is a path from x to y in the dependency graph of C with arc labels A_1, A_2, ... , A_n. Now suppose B_1, B_2, ... , B_m is a schedule that correctly computes the value of y. The sequence A_1, A_2, ... , A_n must be a subsequence of the sequence B_1, B_2, ... , B_m. Observe also that we can treat any schedule for a circuit C as a string whose alphabet is the set of blocks of C. These observations lead to the following theorem.

Theorem 4: *Let C be a (possibly cyclic) purely combinational circuit containing N functional blocks. Let P be the set of all paths from primary inputs to primary outputs in the dependency graph of C, and let P' be the set of all schedules obtained by listing the labels on the arcs of the paths of P in source-to-sink order. The minimum number of simulation steps required to correctly compute the value of all primary outputs of C is* $||SCS(P')||$, *the length of Shortest Common Supersequence of the strings of P'.*

Given a set of strings Q on some alphabet S, it is a simple matter to construct a circuit C containing $||S||$ blocks and $||Q||$ paths from primary inputs to primary outputs, such that each path from primary inputs to primary output corresponds to a string of Q. Thus the problem of finding the Shortest Common Supersequence of a set of strings is reducible to the problem of finding an optimal schedule for an arbitrary circuit C. Since the Shortest Common Supersequence problem is known to be NP-complete[8], the problem of finding an optimal schedule for the blocks of a (possibly cyclic) circuit is also NP-complete. We state this result as Theorem 5.

Theorem 5: *Finding the optimal schedule for a (possibly cyclic) purely combinational circuit is NP-complete.*

This result shows that any algorithm capable of producing an optimal schedule for an arbitrary circuit C, will be intractable unless P=NP. Circuits with reconvergent fanout may have an exponential number of paths from primary inputs to primary outputs. However this theorem remains true even if the domain is restricted to circuits without such fanout. Therefore it is necessary to consider approximation algorithms for the scheduling problem.

3. Approximation Algorithms

Because finding an optimal schedule for a combinational circuit is equivalent to finding the Shortest Common Supersequence (SCS) of a set of strings, the algorithms presented in this section will be approximations for the SCS problem[8].

Two algorithms are studied in this paper. The first, SCSTEST1, is a "Breadth-first" local optimization algorithm. The way SCSTEST1 schedules items depends on the information in a two column "window", as shown in Figure 4.

	C0	C1	C2	C3
s1:	a	b	c	
s2:	b	c	e	g
s3:	a	d	f	
s4:	e	f		

Figure 4. The Matching Window.

A item in the current column (C0) is called *free item* if it does not appear in column (C1). The number of times an item appears in column C0 is called the *number of instances* of that item, while the number of times an item appears in all columns except C0 is called the *number of successors* of that item. In Figure 4, "a" and "e" are free items. Item "a" has two instances and zero successors, while items "b" and "e" both have one instance and one successor. The SCSTEST1 algorithm is illustrated in Figure 5.

The SCSTEST1 algorithm contains five heuristic rules:

H0: Select only one item per iteration to reduce the chance of dependency loss.

H1: If only one item is free, select that free item, preventing loss of local dependencies.

H2: If H1 fails, select the item with maximum instances, which will tend to bring in more dependency information for the next iteration.

H3: If H1 and H2 both fail, select the item with the minimum number of successors. It is more likely that item with large number of successors could be absorbed late without penalty.

H4: If none of above rules succeed, select the item in the longest string, which tends to reduce the depth of the circuit and improve the breadth-first locality.

When there are no free items, a cyclic dependency exists, and all items are treated as if they were free. Figure 6 gives an example SCSTEST1 and the heuristic rules used for each step.

```
SCS = the null string;
While at least one string is not empty
  compute free_item_count
  if free_item_count = 1
    current_item = single free item
  else
    if free_item_count = 0
      place all items in current_free_item_set
    else
      place free items in current_free_item_set
    endif
    for each item in current_free_item_set
      determine the number of instances
    eliminate items whose number of instances is < max;
    {skip the next four steps if only one item remains.}
    for each item remaining in current_free_item_set
      determine the number of successors;
    eliminate items with > min successors;
    {skip the next two steps if only one item remains.}
    for each item remaining in current_free_item_set
      determine the length of the string containing the item;
    eliminate all items whose length is less than maximum;
    current_item = choose an item from current_free_item_set;
  endif
  append current_item to SCS;
  for all strings whose first character matches current_item
    delete the first character of the string;
endwhile
```

Figure 5. The SCSTEST1 Algorithm.

SCSTEST1 is of complexity $O(N^2M^2)$, where N is the number of strings and M is the length of the longest string. As Figure 6 shows, it is not always necessary to count the number of successors, so the average complexity may be nearer $O(N^2M)$.

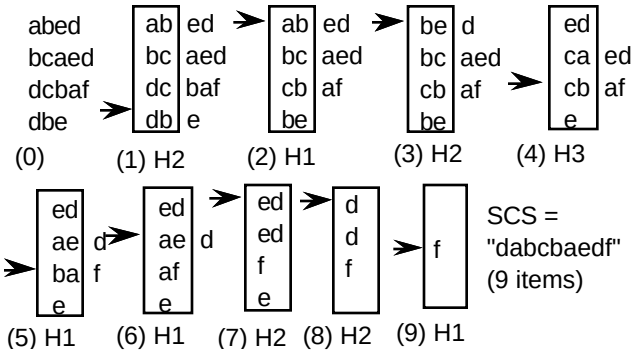


Figure 6. An Example of the SCSTEST1 Algorithm.

The second algorithm, SCSTEST2, is a "depth-first" local optimization heuristic algorithm (see Figure 7). It uses a fast LCS (Longest Common Subsequence) algorithm[9] to merge strings into an approximate SCS. It begins with longest string and checks each successive approximation against the remaining strings to select the best match and minimize the work required to produce the next approximation. When the selected string can be combined with the current approximation in more than one way, it is combined so as to maximize the potential for matching with the remaining strings. Figure 8 demonstrates the operation of SCSTEST2 using the strings from Figure 6.

SCSTEST2 is semi-optimal n the sense that it is similar to the optimal algorithm which exhaustively examines every possible way of merging strings. SCSTEST2 differs from the optimal algorithm in two ways. First, if there is more than one maximum LCS, SCSTEST2 arbitrarily picks one not necessarily the best. Second, the heuristic SCSTEST2 employs to choose the SCS candidates is not guaranteed to be optimal. The complexity of SCSTEST2 is dependent on the input pattern. The worst case occurs when there are no matched pairs among the input strings. For a set of N strings of length M without any matched pairs, SCSTEST2 is of complexity $O(\binom{M}{N*M}*(N*M \log (N*M)))$ which is intractable even for medium sized strings. On the other hand, SCSTEST2 is faster than SCSTEST1 when there are many matches between input strings.

```
begin
  list_of_strings STRING_LIST;
  string          SCS;
  sort strings descending by length assign to STRING_LIST;
  SCS = head of STRING_LIST;
  Delete the first element of STRING_LIST;
  while STRING_LIST is not empty
    for each string in STRING_LIST;
      {LCS = Longest Common Subsequence }
      determine LCS of string and SCS;
      determine ways string can be combined with SCS;
    endfor
    Select the string from STRING_LIST
      with minimum number of combinations
      and maximum LCS;
    delete the selected string from STRING_LIST;
    If there is more than one way to combine,
      create a list of candidate SCS strings;
      for each candidate
        determine the LCS of the candidate
                with each item in STRING_LIST
        compute the sum of the lengths of all LCS strings
      endfor
      select a candidate for which the LCS sum is maximal
        and assign to SCS;
    else
      combine selected string with SCS, assign to SCS;
    endif
  endwhile
end
```

Figure 7. The SCSTEST2 Algorithm.

4. Experimental Results

Both algorithms have been implemented in C and tested with random character strings. For each test, we are able to choose the number of strings(N), the maximum and minimum length of each string(M), and the alphabet from which the strings will be chosen(S). Within these constraints, each string consists of a sequence of randomly chosen characters. Two types of tests have been run and the results are shown in Figures 9 and 10. N_{in} is the number of input characters, N_{op} is the length of the optimal SCS obtained by manual inspection (for small input patterns only) and N_{s1} and N_{s2} are the lengths of the approximations obtained by SCSTEST1 and SCSTEST2. The test results are consistent with the calculated time bounds of the algorithms.

Figure 9 contains the results for small input patterns with N=4, M=2-6 and three types of match densities. For such input patterns, the optimal SCS for each test can be obtained by inspection. Both algorithms work well under all match densities. SCSTEST1 degrades slightly with increasing density. It tends to make most selections by rules H2, H3, and H4 under these conditions. This causes loss of dependency information hence reducing the effectiveness of the algorithm. SCSTEST2 works almost perfectly with all match densities, but it uses more time for sparse matches than for dense matches.

Input strings: "abed","bcaed","dcbaf","dbe"
Initialization: SCS = "bcaed" (first longest)

1st iteration:

Step1: b c a e d / a b e d — LCS=3 PC=0; d c b c a e d / b a f — LCS=2 PC=3; d b c a e d / b e — LCS=2 PC=0
Select "abed"

Step 2 Merge: SCS = bcabed

2nd iteration:

Step1: b c a b e d / d c b a f — LCS=2 PC=12; b c a b e d / d b e — LCS=2 PC=4
Select "dbe"

Step 2 Merge: SCS candidates:
"dbcabed","bdcabed","bcdabed","bcadbed"

Step3 SCS Selection LCS sizes: "dbcabed" 3,
"bdcabed" 3, "bcdabed" 2, "bcadbed" 2
Select "dbcabed"

3rd iteration:

Step1: Select "dcbaf"

Step 2: Merge. SCS is dbcbafbed (9 items)

Figure 8. An example of the SCSTEST2 Algorithm.

The second test examines the relative performance of two algorithms with various sizes of inputs. The accumulated test results are shown in Figure 10.

Match Density	N	M	S	N_{in}	N_{op}	N_{s1}	N_{s2}	$\frac{N_{s1}}{N_{op}}$	$\frac{N_{s2}}{N_{op}}$	$\frac{N_{s1}}{N_{s2}}$
Dense	4	2-6	[a-b]	485	212	220	213	1.04	1.00	1.03
Medium	4	2-6	[a-f]	483	322	330	326	1.02	1.01	1.01
Sparse	4	2-6	[a-z]	492	410	415	410	1.01	1.00	1.01

(Accumulation of 30 tests for each match density)

Figure 9. The Effect of Match Density on Performance.

Figure 10 shows that SCSTEST1 works better than SCSTEST2 when the input data has many short strings. This can be attributed to the "breadth-first" searching algorithm and two column "window" used by SCSTEST1. SCSTEST1 considers all input strings simultaneously for each selection. However with long strings the two column window prevents it from seeing the chances for "remote" matches. Since it is computationally infeasible to find the optimal SCS for even medium size input strings, we have no way to investigate the absolute performance of SCSTEST2. From the previous test, however, we expect SCSTEST2 to work well in most the cases.

Pattern	N	M	S	N_{in}	N_{op}	N_{s1}	N_{s2}	$\frac{N_{s1}}{N_{op}}$	$\frac{N_{s2}}{N_{op}}$	$\frac{N_{s1}}{N_{s2}}$
Short	20	5-10	[a-h]	1469	-	403	416	-	-	0.97
Medium	10	10-20	[a-h]	1515	-	652	630	-	-	1.03
Long	3	15-40	[a-h]	812	-	625	556	-	-	1.12

(Accumulation of 10 tests for each pattern)

Figure 10. The Effect of String Length on Performance.

5. Conclusions and future work

Two SCS approximation algorithms have been presented. Both algorithms can be used to schedule blocks for functional simulations based on the assumption that blocks may not be subdivided, and that cyclic references may exist between blocks. The SCSTEST2 algorithm produces better approximations than SCSTEST1, but SCSTEST1 gives better performance, particularly for sparse matches. We expect sparse matches to be the rule for scheduling functional blocks.

One area that needs further investigation is the problem of scheduling functional blocks when the dependency information between nets is unknown. Such information is usually not expicitly contained in user source files, and may be difficult to obtain in some circumstances. A scheduling algorithm that depends only on topological information is preferable in such situation. Another problem is finding techniques to automatically decompose blocks coded in high-level languages, to eliminate cyclic references.

REFERENCES

1. R. L. Wadsack, "Design Verification and Testing of the WE 32100 CPUs," *IEEE Design &Test of Computers*, Aug. 1984, pp. 66-75.
2. J. P. Hayes, "An Introduction to Switch-Level Modeling," *IEEE Design & Test of Computers*, Aug 1987, pp.18-25.
3 K. Tham, R. Willoner, D. Wimp, "Functional Design Verification by Multi-Level Simulation," *Proceedings of the 21st Design Automation Conference*, 1984, pp. 473-478.
4. M. A. Breuer, A. D. Friedman, *Diagnosis and Reliable Design of Digital Systems*, Computer Science Press, Woodland Hills, CA, 1976.
5. L. Wang, N. Hoover, E. Porter, J. Zasio, "SSIM: A Software Levelized Compiled-Code Simulator," *Proceedings of the 24th Design Automation Conference*, 1987, pp. 2-8.
6. Z Wang, P. Maurer, "Scheduling High-Level Blocks for Functional Simulation," University of South Florida Department of Computer Science and Engineering Technical Report Number CSE-88-24, 1988.
7. S.A. Szygenda, E. W. Thompson, "Digital Logic Simulation in a Time Based Table Driven Environment," Computer, March 1975.
8 D. Maier, "The Complexity of Some Problems on Subsequences and Supersequences," *Journal of the ACM*, Vol. 25, No. 2, April 1978, pp. 322-336.
9. D. Hirschberg, "Algorithms for the Longest Common Subsequence Problem," *Journal of the ACM*, Vol. 24, No. 4, Oct, 1977, pp. 664-675.