Fast Compiled Logic Simulation Using Linear BDDs

Sudeep Gupta and Keshav Pingali
Department of Computer Science
Cornell University, Ithaca, New York 14853.

Abstract

This paper presents a new technique for compiled zero delay logic simulation, and includes extensive experiments that demonstrate its performance on standard benchmarks. Our compiler partitions the circuit into *fanout-free regions* (FFRs), transforms each FFR into a linear sized BDD, and converts each BDD into executable code. In our approach, the computation is sublinear in the number of variables within each partition because only one path, from root to leaf, of the BDD is executed; therefore in many cases, substantial computation is avoided. In this way, our approach gets some of the advantages of oblivious as well as demand-driven evaluation. We investigated the impact of various heuristics on performance, and based on this data, we recommend good values for design parameters. A performance improvement of up to 67% over oblivious simulation is observed for our benchmarks.

1 Introduction

The rapid growth of size and complexity of digital circuits has made logic simulation a bottleneck in circuit design. There are two major simulation strategies: *compiled simulation* and *interpreted simulation*. In compiled simulation, the circuit is transformed into executable code, and this code is executed directly; in interpreted simulation, the circuit netlist is read into a data structure, and simulation is performed by an interpreter which traverses this data structure. The execution of compiled code is usually faster than interpretation, so recent research has focused on compiled simulation. Compiled simulation is the focus of this paper as well.

The key issue in compiled simulation is the timing model — should the simulator assume that circuit elements have delays, or should it work with an idealized zero delay model? Recently, researchers have developed sophisticated algorithms for finding critical paths in the combinatorial sections of circuits[7, 12], which makes timing simulation within a clock cycle less important. Consequently, there is renewed demand for fast simulation with zero delay models. The choice of zero delay models permits flexible scheduling of circuit element evaluation; therefore, the circuit elements can be scheduled as needed to achieve faster simulation. There are three popular paradigms: *oblivious simulation*, *event-driven simulation* and *demand-driven simulation*. In oblivious simulation, each circuit element is evaluated at every time step. In event driven simulation, a circuit element is evaluated only if one of its inputs changes. In demand-driven simulation, a circuit element is evaluated only when its output is needed to produce the output of the whole circuit (for example, if one of the inputs to an AND gate is 0, the subcircuit computing the other inputs need not be evaluated)[1, 17].

For any simulation method, the time required for simulation is determined by the number of circuit evaluations, and the overhead of scheduling. Oblivious simulation performs much more evaluations than either demand-driven or event-driven simulation. However, demand-driven evaluation is usually implemented using procedure calls at every step of the evaluation, and this makes the scheduling overhead very high. Therefore, most of the work in simulation has focused on oblivious and event-driven simulation. Oblivious simulation has lower scheduling overhead, so it is clearly preferable if the level of circuit activity is high. However, there seems to some difference of opinion about the level of circuit activity above which oblivious simulation is preferable [19, 2, 15, 11]. To some extent, this is because event-driven simulation strategies have been improved continuously. For example, the overhead of scheduling has been reduced in COSMOS[5] and HSS [2] through the use of a central scheduler, while Tortle_c uses threaded code to avoid scheduling[11]. A specialized threaded stack based technique called Gateways has also been used[13]. The number of circuit evaluations has also been reduced through the use of levelization based techniques to avoid multiple evaluations of some elements[19, 14], and through the use of Maurer's *Inversion* and *Shadow* algorithms [15]. In spite of these improvements, most simulators used in the industry are based on oblivious simulation. Oblivious simulation permits prescheduling of circuit elements for evaluation, and this not only eliminates the overhead of dynamic scheduling of circuit elements, but it also makes it possible to use registers effectively in simulation. In the event-driven model, the values of all nets must be preserved for the next iteration of simulation; hence they have to be kept in memory. In contrast, in oblivious simulation, these values are needed only for that particular iteration; hence they can be stored temporarily in registers, avoiding load and store instructions. This appears to be very important for achieving good performance (note that comparing event-driven simulation with C code for oblivious simulation can be misleading because most C compilers do not do a good job of register allocation on global variables). In this paper, we use oblivious simulation as the base line for measuring performance improvements. However, our techniques are general and can be applied to improve performance in event-driven approaches as well.

In this paper, we describe a novel technique for circuit simulation which uses *binary decision diagrams* (BDDs) to represent portions of circuits. As we explain below, we use BDDs to eliminate the overhead of dynamic scheduling of circuit elements (an advantage shared by oblivious simulators), while retaining the ability to avoid some unnecessary computation (an advantage shared by event-driven and demand-driven simulators). The use of BDDs is not new, but previous efforts at using BDDs for simulation have converted the entire circuit to a BDD and this can be very expensive. In contrast, we build BDDs only for *fanout free regions* of the circuit, which ensures that all BDDs have linear size. These regions themselves are treated as complex elements that are nodes in a Graph of Regions (GoRe), and we use a flavor of oblivious simulation to evaluate a GoRe. An overview of our approach is shown in Figure 1.

The main issues addressed in this paper are the following.

- How are fanout free regions determined?
- How do we compile code for each region?
- What code is generated to evaluate the graph of regions?

To evaluate the performance of our approach, we use ten benchmarks. These circuits have been obtained from the ACM/SIGDA benchmarks suite archived at *mcnc.org*. Table 1 describes these benchmarks in details. In circuit `Diffeq`, all 36 bidirectional

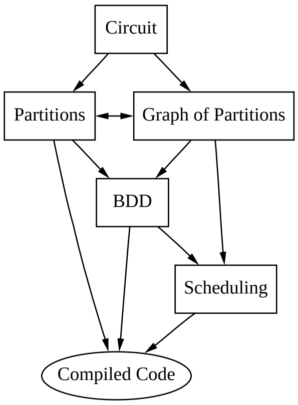


Figure 1: An overview of our approach

```
Input format:  EDIF
Platform:  SPARCsystem 600MP series with SunOS 5.3 (Solaris)
```

Circuit	Circuit type	Number of gates	Number of Inputs	Number of outputs
Cordic	Combinatorial	2056	23	2
Dsip	Sequential	3227	229	197
Misex3	Combinatorial	3624	14	14
Diffeq	Sequential	4346	$28 + 36$	$3 + 36$
Alu	Combinatorial	2381	14	8
Seq	Combinatorial	2875	41	35
Apex2	Combinatorial	3149	39	3
C6288	Combinatorial	4768	32	32
C7552	Combinatorial	4094	207	107
Bigkey	Seqential	2741	263	197

Table 1: The benchmarks

ports are counted as both input and output ports.

The rest of the paper is organized as follows. Section 2 describes BDDs. Section 3 describes how we determine fanout free regions. In general, there are many BDDs that correspond to a given combinatorial circuit. Therefore, Section 4 describes our heuristics for constructing a good BDD for a fanout free region. Section 5 describes how partitions are scheduled. Section 6 shows how C code is generated for the entire circuit. Section 7 compares the performance of our approach with that of oblivious simulation. Finally, Section 8 presents some conclusions and discusses future work.

2 Binary Decision Diagrams(BDDs)

A binary decision diagram is a representation of a boolean expression [3, 18]. A BDD `b` represents the boolean expression f if f is written as $f = \overline{x} f_0 + x f_1$ (called the Shannon

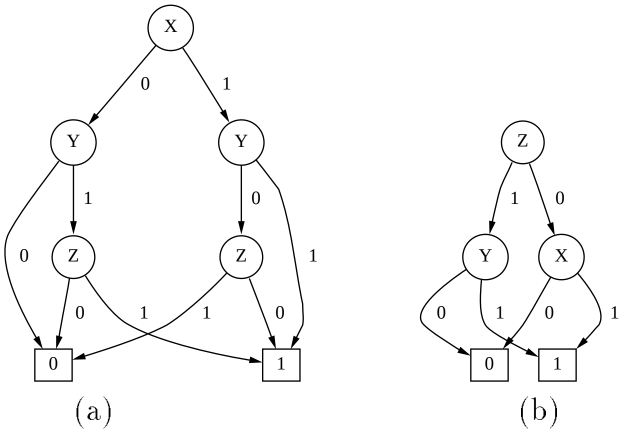


Figure 2: BDD for $f = \overline{z}x + zy$ with order of expansion (a) x, y, z and (b) z, x, y

expansion), where x is one of the variables in f and f_0 and f_1 do not contain x. The root of `b` is x and the left and right children are the BDDs of f_0 and f_1 respectively. Leaf nodes represent the constants `TRUE` and `FALSE`.

Any combinatorial circuit can be transformed into a BDD whose variables are the inputs to the circuit. Given the input values, the circuit can be simulated by traversing the BDD. BDDs are attractive for simulation for 2 reasons: (i) the complexity of evaluating a boolean expression is proportional to the length of the path of traversal in the BDD (which is bounded by the number of inputs in the circuit), and (ii) it is not necessary to evaluate circuit elements which produce values that are not on the path traversed during BDD computation, which gives a BDD-based approach the efficiencies of the demand-driven approach.

Unfortunately, the size of the BDD is highly sensitive to the order in which the variables are expanded. For example, the size of the BDD of $f = \overline{z}x + zy$ is much larger if the order of expansion is x, y, z (Figure 2(a)) than if the order of expansion is z, x, y (Figure 2(b)). Furthermore, finding an optimal ordering is NP-complete[4]. Even with optimal ordering, some circuits have very large BDDs. However, it can be shown that for some circuits, a linear sized BDD can always be obtained. In particular, circuits without fanout always have a linear BDD[6, 8, 12]. Our approach to circuit simulation involves determining such fanout free regions and using BDDs for simulating those regions. For future reference, we define these regions formally.

Definition 1

- *Any region without a fanout node is called a* fanout-free region *(FFR).*
- *An FFR which is not contained in another FFR is called a maximal FFR (MFFR).*

3 Partitioning

We now describe how we partition circuits into regions which we convert into BDD's. First, we focus on combinatorial circuits. Note that a naive partitioning can introduce cycles between partitions as shown in Figure 4. If A and C are kept in partition P_{AC} and B is kept in P_B then, a cycle is introduced between P_{AC} and P_B because of the

```
Procedure MFFR_Partition (Circuit)
{
    for each output o of circuit do
        current_partition = create_new_partition();
        let o be output of node n;
        Traverse_inputs_and_partition(n);
    endfor
}
Procedure Traverse_inputs_and_partition (Node n)
{
    if ((ALREADY_VISITED(n)) or (n==Circuit Input)) then
        return ;
    endif
    MARK_VISITED(n);
    if (fanout(n) > 1) then
        push(current_partition, traversal_stack);
        current_partition = create_new_partition();
    endif
    Add n to current partition;
    for all_inputs of n do
        Traverse_inputs_and_partition(input_node);
    endfor
    if (fanout(n) > 1) then
        current_partition = pop(traversal_stack);
    endif
}
```

Figure 3: Algorithm for MFFR partitioning

edges E_{AB} and E_{BC} in the GoRe. The introduction of cycles complicates scheduling, and should be avoided.

Figure 3 shows our partitioning algorithm for finding *Maximal Fanout Free Regions* (MFFR's). Intuitively, MFFR's are found by cutting all fan-out points in the circuit, and making each connected component of the resulting graph into a partition. The identification of fan-out points and the determination of partitions can be done by a depth-first traversal of the graph which starts at the outputs of the circuit and builds partitions on the fly, as shown in Figure 3.

Theorem 1 *If the circuit does not have cycles, then the graph of regions produced by the partitioning of Figure 3 does not have cycles,* i.e., *the GoRe is cycle free.*

Proof: Omitted.

Figure 5 shows partition size statistics for all benchmark circuits. The biggest partition had 2500 gates, but most partitions have less than 50 gates. Although we have described the partitioning for combinatorial circuits, it can be extended to more general circuits as follows.

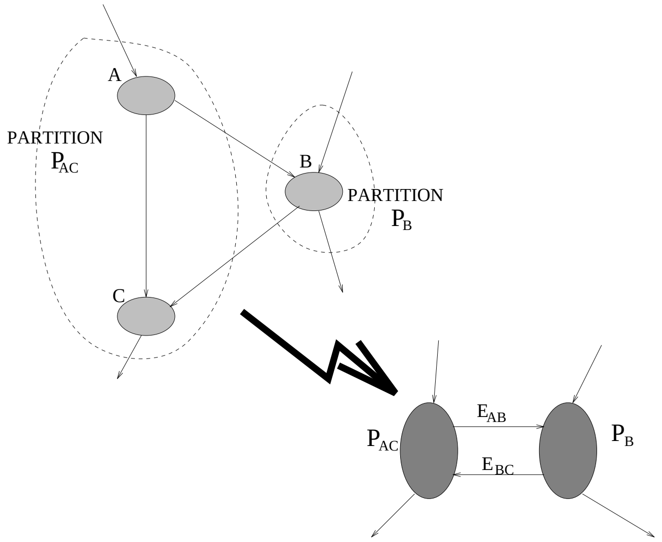


Figure 4: Naive partitioning can introduce cycles

Sequential circuits: Without loss of generality, it can be assumed that D flip-flops (DFFs) are the only sequential elements in the circuit. Since asynchronous sequential circuits are not considered, removal of all DFFs from the sequential circuit must leave a combinatorial circuit. This is illustrated in Figure 6 where the dotted node denotes a "removed" node. The simulation is done on the broken circuit with output D_o and input D_i. D_i is assigned the value D_o in a separate latching phase.

Fanin nodes: These nodes are introduced by wired and tristate logic. Fanin nodes can be treated as separate partitions with multiple inputs. Such partitions are converted into multi-terminal BDDs since values at such nodes can be 'X' or 'Z'. This approach illustrates how we deal with the problem of multi-valued logic. If the simulator finds a value other than 0/1 on a fanin net, it switches to code which performs vanilla interpretive simulation and comes back to the BDD based compiled code when the value becomes 0 or 1. Since the value on a net is different from 0 or 1 only in the beginning of the simulation in most circuits, the interpretive code is rarely needed. Thus, we can achieve efficient simulation without sacrificing the ability to handle multivalued logic.

Higher level abstractions: Like fanin nodes, these can be treated as separate partitions for which simulation code is provided by the circuit designer or from a library. Therefore, we can accommodate higher level abstractions with sacrificing efficient simulation.

4 BDD Construction

We now address the problem of generating a BDD from an FFR. The main problem is the design of heuristics for determining the order of selection of input variables for BDD construction. Each FFR is a tree, and a simple algorithm for building a BDD

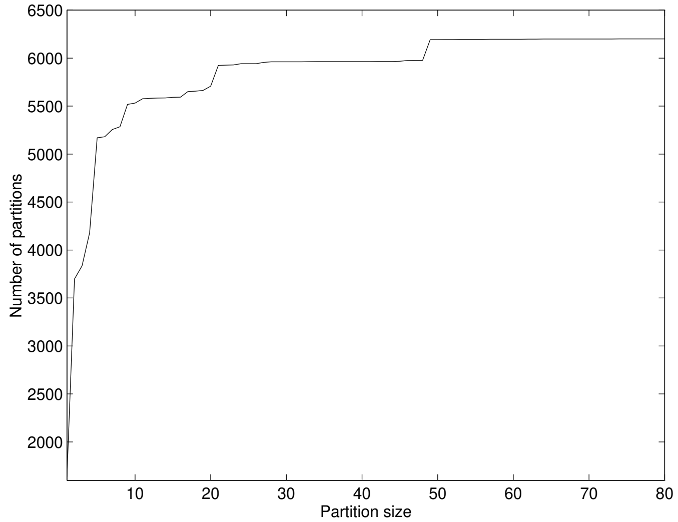


Figure 5: Number of partitions less than the partition size

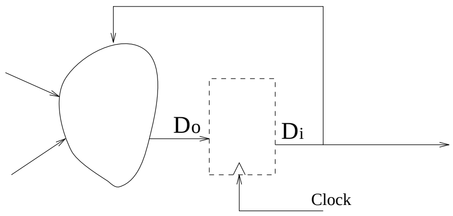


Figure 6: A sequential circuit

is to traverse the FFR recursively starting at its output, converting each sub-trees below a node into a BDD, and then combining these BDD's using the functionality of the node. It can be shown that the resulting BDD is always linear in the size of the circuit (the key property is 'contiguous' conversion — that is, at any node, the subtrees below the node are converted to BDDs separately, without any interleaving of variables from different subtrees). In general, a node has many subtrees below it, and the order in which these subtrees are chosen for conversion makes a difference in the efficiency of simulation. The goal in this section is to choose an order for subtrees so that we minimize the path in the BDD that is traversed during simulation, since this reduces execution time.

We have investigated a number of heuristics for the problem of ordering subtrees of a node for conversion to BDD form. These heuristics use the fanout and level of input variables. Intuitively, variables with high fanout are used in more places in a circuit than variables with low fanout, so it may be possible to examine these variables quickly and determine the output, without looking at other input variables. Expanding such variables earlier in the conversion process also creates opportunities for an optimization called variable merging that is described below. The level of a variable (how many gates away it is from the output of the FFR) can also be exploited — intuitively, a variable whose level is small is close to the output of the FFR, and may affect it more significantly than a variable whose level is large. Therefore, it is plausible that such variables should be examined first. We studied the following figures of merit for each input variable of an FFR.

- Fanout: The fanout of the FFR producing the variable.
- Fanout_local: The number of fanout edges of the FFR producing the variable, which are incident on the current partition.
- Level: The smallest number of gates on a path from the variable to the output of the FFR

In general, a subtree below a node has many input variables, so the numbers for each input variable of the subtree must be combined to generate one 'figure of merit' for the entire subtree. Some natural choices for combining functions are minimum, maximum, sum and average. We found that sum and average gave almost the same results, so we dropped average from consideration. If a heuristic gives equal priority to two variables, the ordering among them was chosen arbitrarily. This arbitrariness can affect performance, so we studied the effect of using three random orderings by specifying different seed values to a pseudo-random number generator. There was little change in performance, so in the rest of our experiments, variables with equal priority were chosen arbitrarily.

Figure 7 shows experimental results. It can be seen that the heuristics *Fanout_max* (fmax), *Fanout_sum* (fsum), *Fanout_local_max* (flmax), *Fanout_local_avg* (flsum) and *Level_min* (lmin) perform better than the other heuristics. For the later part of the study, we used only the *Fanout_max* and *Fanout_local_max* heuristics. There was a minor gain achieved by combining the heuristics, although, aditional research is required to study the effects of combining heuristics.

A key optimization in BDD construction is *variable merging*. If an FFR has multiple inputs being driven by the same net, there is no need in principle to consider these inputs separately. For example, in Figures 8(a), the partition obtained by clipping the fanout point will have both A and C as inputs, but since these are driven by the same net, these variables can be merged in the BDD. Figures 8(c) and (d) show the BDD for this example before and after merging of these variables (the variable Y in Figure 8(d) stands for A and C combined). Note that variables merging does not always give a smaller BDD. For example, if B and D came from the same net, they can be merged, but the BDD becomes larger, as shown in Figure 8(e).

We now describe our variable merging algorithm. The BDD's for the two inputs to the OR gate are shown in Figure 8(b) as I1 and I2. A simple-minded approach to building a BDD for the entire circuit is to identify all edges in I1 that point to 0, and redirect them to the root of I2. This results in the BDD shown in Figure 8(c). To obtain the advantages of variable merging, we first compute the value of each variable at every edge in I1 that points to 0 or to 1. Note that this value can be 0, 1, $\perp$ (undefined) or $\top$ (either 0 or 1 depending on the path). For example, for the edge from A to 0 in I1, the value of A is 0, but the value of all other variables is $\perp$. An example of the need for $\top$ is provided by the BDD of Figure 8(c) — if this BDD is itself used to build a larger BDD, the value of A on the edge from C to 0 will be $\top$ because the value of A is either 0 or 1 depending on the path from A to this edge. Figure 8(f) shows the lattice of values. Procedure *Propagate_Values* in Figure 9 propagates values of input variables along the edges of a BDD by visiting the edges in topological order. In the

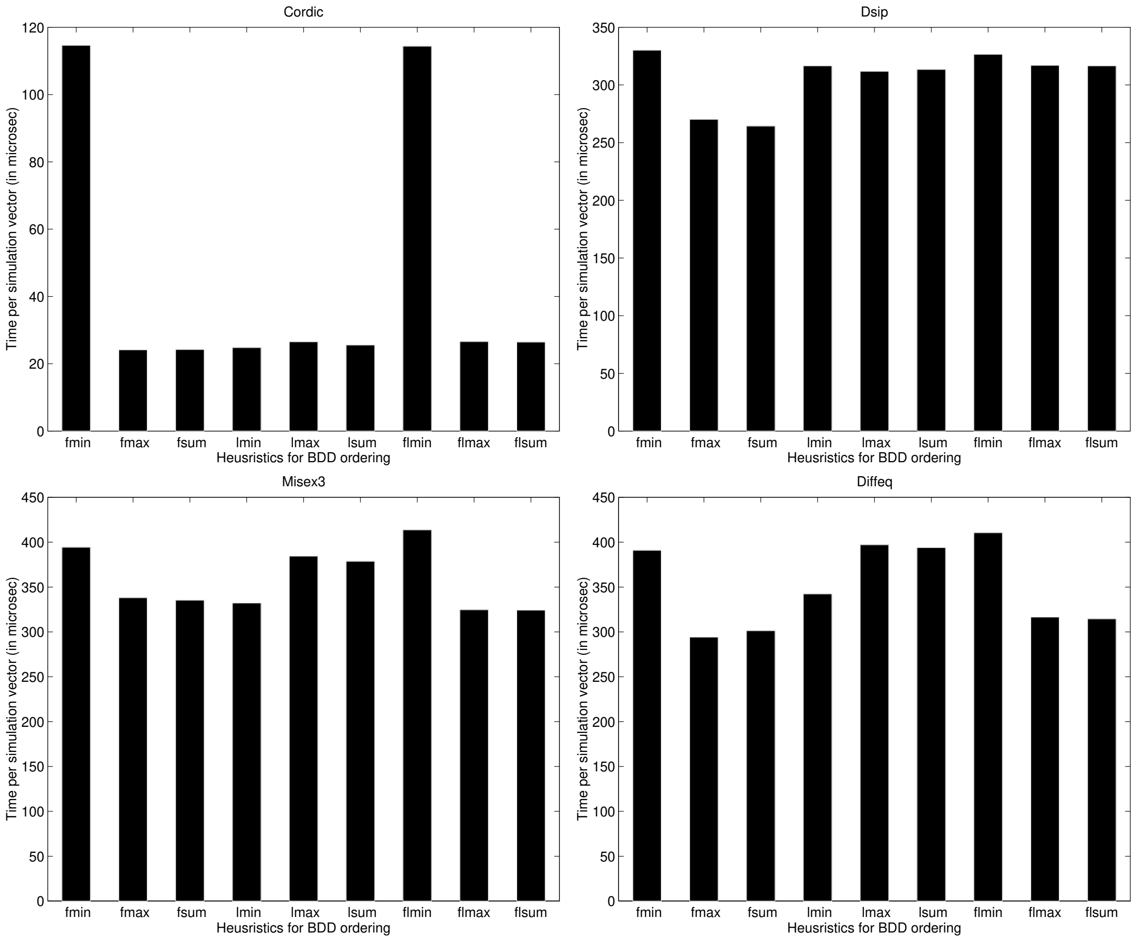


Figure 7: Simulation time *vs.* heuristics

actual implementation, value propagation is done together with BDD construction, but we have described these computations separately for clarity.

The values computed by the first phase are used in merging BDD I2 into BDD I1. Each edge in I1 that points to a 0 or to 1 is examined, and the vector of values computed for that edge is used to traverse the BDD of I2 till we reach a node whose value in the vector is either $\top$ or $\perp$. The edge in I1 is redirected to this node. This process can obviously be viewed as partial evaluation of the BDD of I2, using the vectors of values computed in the first step. Procedure **Partial_Evaluate** in Figure 9 shows the pseudocode. In the running example, for edge $\langle B, 0\rangle$ in I_1, the value of A (hence C) is known to be 1. So we traverse I_2 to D. Since the value of D in the vector is $\perp$, partial evaluation stops, node D is returned, and the outgoing edge from B is redirected to D. The BDD in Figure 8(d) shows the final result.

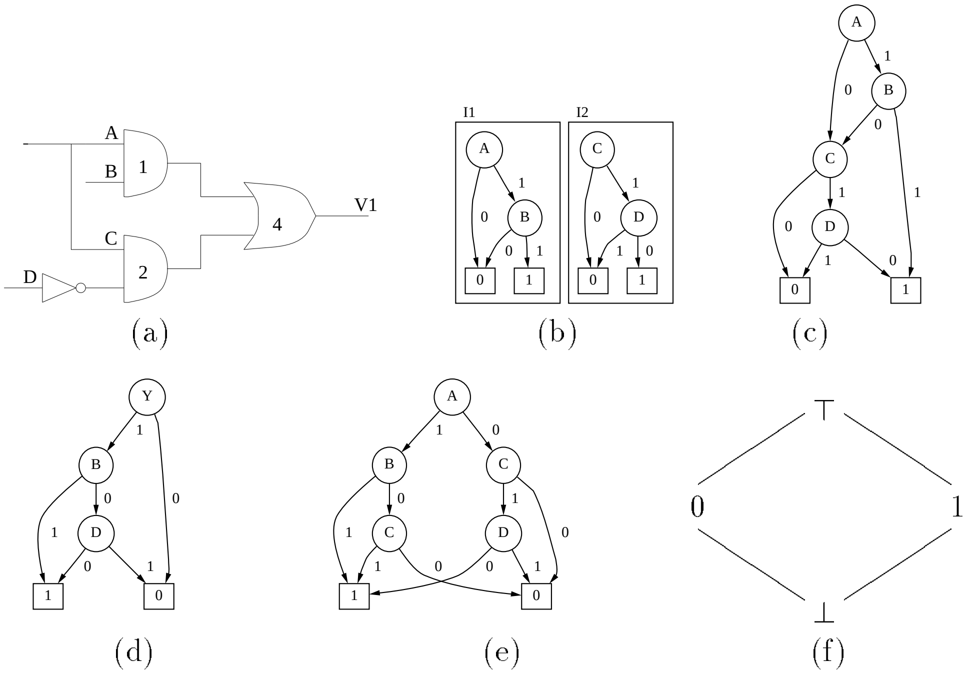


Figure 8: Merging variables

5 Scheduling of Partitions

We now describe how partitions are scheduled for execution. Figure 10 shows four FFRs (C,D,E and F), and their BDDs. The output of partition D is used in two different inputs of partition F; the output of partition C is also an input to partition F. One approach to scheduling partitions is to associate a function with each partition, and call that function when the output of the partition is desired. For example, if the output of partition F is the output of the circuit in Figure 10, we can traverse the BDD for F, and call the function associated with partition D if its output is needed. Since the output of a partition may be used in multiple places, we can avoid making repeated calls to the same function by storing values required across partitions in a data structure, and calling a function only if the corresponding value has not already been computed. However, in our experiments, we found that the overhead of function calls was substantial; moreover, inter-procedural register allocation is difficult for compilers.

A different approach is to use oblivious simulation across partitions. That is, all partitions are scheduled for execution, even if the outputs of some of them are not needed. However, in Figure 10, if the value of a is 0, partitions C, D and E do not need to be evaluated. To get some of the benefits of demand-driven evaluation, we can back away from oblivious simulation across partitions, and instead, schedule the execution of partition D just before the value of b is tested in the code for region F. This ensures that D is not evaluated if the value of a is 0. However, if a is 1, b is 0 and c is 1, the value of D is not required, but we would compute it anyway. In other words, this scheduling scheme has some but not necessarily all the computational savings of demand-driven evaluation.

We now describe our algorithm for determining where partitions must be scheduled.

```
Procedure Propagate_Values ()
{
    Initialize each element of each vector to ⊥
    for each node N in topological order  do
       % Let variable at node N be V, and input edges of N be E1,E2,...
       % Let F0 and F1 be the output edges of node N with labels 0 and 1 respectively
       % Propagate vectors from input edges to output edges
       ValueVector(F0) := ValueVector(E1) ∪ ValueVector(E2) ....
       ValueVector(F1) := ValueVector(F0)
       % Update values for variable V in output vectors
       ValueVector(F0)[V] := 0
       ValueVector(F1)[V] := 1
    endfor
}
Procedure Partial_Evaluate(ValueArray,BDD)
{
    R = Root(BDD)
    T = ValueArray[R]
    while ( T = 1 or T = 0)  do
       Let ⟨R,N0⟩,⟨R,N1⟩ be the outgoing edges of R,
          labeled 0 and 1 respectively;
       if (T = 1)  then
          R = N1;
       else R = N0;
       endif;
       T = ValueArray[R];
    endwhile
    return R
}
```

Figure 9: Algorithm for variable merging

First, the graph for the BDD for each partition is built. Next, if the output of partition Y is an input of partition X, an edge is inserted from the nodes in the BDD for X which use the output of Y to the root of partition Y, as shown in Figure 10. Finally, we introduce a node named START, and connect this node to the root of each BDD that computes an output of the program. In this graph, the problem of scheduling partitions is reduced to the problem of computing the *immediate dominator* of a node.

Definition 2 *A node w is said to* **dominate** *a node v if every path from START to v contains w.*

It can be shown that dominance is a transitive relation, and that its transitive reduction is a tree-structured relation called the *dominator tree*, rooted at START. In this tree, the dominators of a node v are all the nodes on the path from v to START. The *immediate dominator* of a node v (other than START) is its parent in this tree. The dominator tree of a program can be constructed in $O(|E|\alpha(|E|))$ time using an

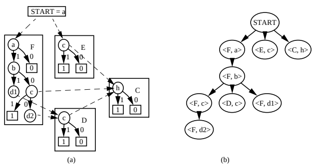


Figure 10: Part of a GoRe with BDDs and the corresponding dominator tree

algorithm due to Tarjan and Lengauer [10], or in $O(|E|)$ time by using a rather more complicated algorithm due to Harel [9].

Figure 10(b) shows the dominator tree for the running example. To find where D should be scheduled, we find the immediate dominator of the root of the BDD for D, which is seen to be the node labeled b in the BDD for partition F. For partition C, the immediate dominator of the root of its BDD is START, so it is always executed, just as in oblivious simulation.

6 Compiling into C Code

The BDD of each partition can be compiled into C code by creating an *if-then-else* statement for each node in the BDD. However, *if-then-else* statements introduce conditional branches in the compiled code, which is an undesirable feature for optimum utilization of modern pipelined processors. If the MFFR is small, the overhead of conditional branches can overwhelm the performance advantages of using the BDD to evaluate only a portion of the circuit. On the other hand, code for oblivious simulation is free of branches, but it has the disadvantage that it performs more circuit evaluations than the BDD based approach.

This suggests that rather than compile every BDD into code with conditionals, we should examine tradeoffs between oblivious and BDD based approaches. We examined two such tradeoffs.

In the first set of experiments, we compiled a partition using either *if-then-else* code or oblivious simulation code, using a parameter called *Threshold Size* (TS for short) to decide which compilation strategy should be used — partitions whose size was above TS were compiled using conditional branches, with oblivious simulation code being used for smaller partitions. By studying simulation performance for different values of TS, we determined a good value for TS.

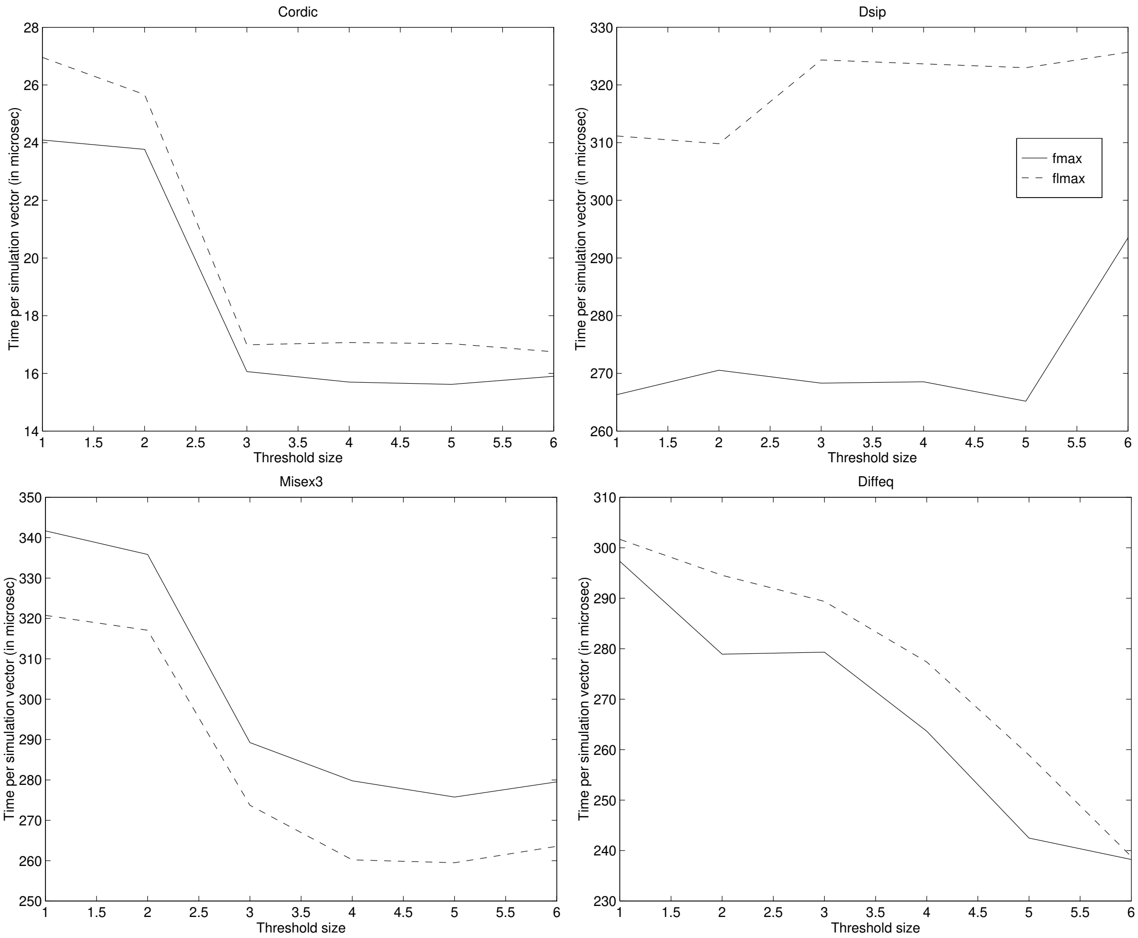
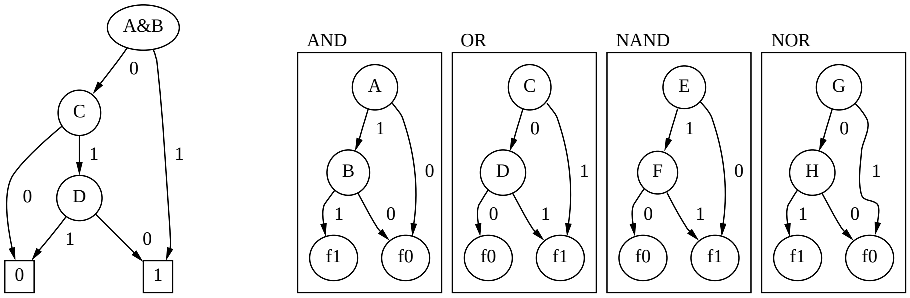


Figure 11: Simulation time *vs.* TS

Figure 11 shows how simulation time varies with TS. Intuitively, we would expect this graph to be U-shaped — for small values of TS, even small partitions are compiled using the BDD-based technology, which is inefficient, while for large values of TS, even large partitions cannot exploit our compilation technology. From these experiments, it appears that $TS = 5$ is a reasonable choice.

The idea of using threshold sizes is an 'all-or-nothing' approach to combining oblivious and BDD based approaches — that is, a BDD is either compiled entirely using conditionals, or it is compiled entirely into oblivious code. A more fine-grained trade-off is to compile some parts of a BDD into oblivious code, and use conditionals to evaluate the rest of it. We call this locally oblivious simulation. Figure 12(a) shows this symbolically by illustrating the result of using oblivious code to simulate the AND gate with inputs A and B in Figure 8(a). In effect, the nodes for A and B in the BDD of Figure 8(c) are collapsed together to form a compound node that is evaluated using boolean operations. In our compiler, we collapse parent-child pairs which have at least one child in common (in this case, C is a child of both A and B). The algorithm for collapsing nodes is shown in Figure 13, and is based upon pattern matching on one of the standard patterns of Figure 12(b). In these patterns, each node itself can be a collapsed node. To limit the number of nodes that are collapsed together, we set



(a) (b)

Figure 12: Collapsing nodes and standard patterns

```
Procedure Collapse (b : BDD)
{
    for each node n in topological order  do
        while ((number of Nodes in n < CP) ⋀ (n matches root in one standard pattern))
            Merge corresponding child into n;
        endwhile
    endfor
}
```

Figure 13: The collapsing algorithm

a limit called the *Collapsing Parameter* (CP), and explored performance for different values of this parameter.

As shown in Figure 14, the simulation time decreases initially upto a CP value of 3-4, after which it becomes nearly constant or increases. It appears that a value of 3 to 4 is a good choice for CP.

7 Comparison with oblivious simulation

Based on the experiments described so far, we recommend the following BDD-based compilation strategy:

- The circuit is partitioned into MFFRs, using Algorithm 3.
- The heuristics *Fanout_max* or *Fanout_local_max* should be used for ordering subtrees at a node during BDD construction. Subtrees ranked equally by these heuristics can be ordered arbitrarily.
- The threshold size TS should be set to 5. Partitions smaller than this should be compiled using oblivious simulation, while partitions larger than this should be compiled using conditional branches.
- The collapsing parameter CP should be set to 3 or 4. At most 3 or 4 variables should be aggregated in a BDD for locally oblivious simulation.

This simulation strategy is compared with oblivious simulation in Table 2. Our approach never does worse than oblivious simulation, and outperforms it in most cir-

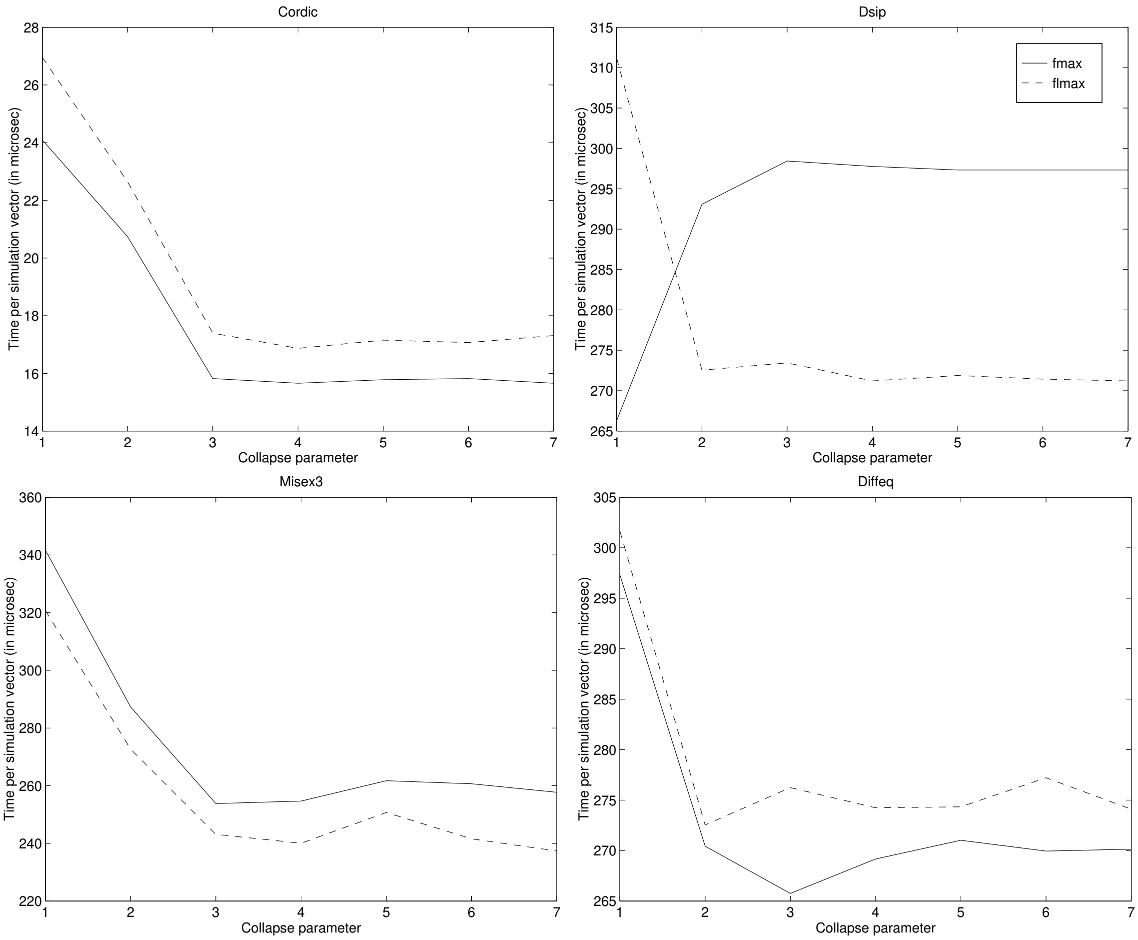


Figure 14: Simulation time *vs.* CP

cuits. The improvement in three of the circuits is more than 50%. Not surprisingly, the advantages of our approach are most evident in circuits in which average partition size is large, since large partitions expose more opportunities for avoiding computation in our approach. For example, Cordic has two partitions of 1194 nodes and 397 nodes, and shows good improvement in performance. In contrast, all partitions in Diffeq are smaller than 22 nodes.

8 Conclusions and future work

The BDD based approach presented in this paper was shown to be useful for generating efficient compiled simulation code. Extensive experiments were performed to determine suitable values of parameters for the compiler. On the benchmark circuits, our approach never did worse than oblivious simulation, and gave substantial performance improvements on others. Since an important issue for industrial simulators is that they must perform well for all circuits, we feel that our approach is particularly relevant in this context.

Timings are presented for simulation per test vector in μsec.

Circuits	Obl. Simulation Time	$TS = 5$ $CP = 4$	Improvement (%)
Cordic	47.62	16.99	64.32 %
Dsip	293.08	279.69	4.57 %
Misex3	374.31	224.84	39.93 %
Diffeq	237.56	234.49	1.29 %
Alu	195.30	127.35	34.79 %
Seq	323.16	224.78	30.44 %
Apex2	364.64	156.85	56.99 %
C6558	111.21	111.21	0.00 %
C7552	266.67	89.58	66.41 %
Bigkey	758.14	221.22	70.82 %

Table 2: Comparison between oblivious and our approach

In future work, we will examine the consequences of relaxing the FFR constraint to make partitions bigger. Since larger partition size gives better performance, this may result in better performance. Since we are generating C code, a number of issues in code optimization such as register allocation could not be studied. We plan to generate assembly code directly to study this issue. We will also study the incorporation of our compiling strategy into event-driven and demand-driven simulators. Compilation of simulation code from high level circuit specifications is another promising research area.

References

[1] W. Y. Au, D. Weise, and S. Seligman. Automatic Generation of Compiled Simulation though Program Specialization. In *28th ACM/IEEE Design Automation Conference*, pages 205–210, 1991.

[2] Z. Barzilai, J. L. Carter, B. K. Rosen, and J. D. Rutledge. HSS-A High Speed Simulator. *IEEE Transcations on Computer Aided Design*, 8(4):601–616, July 1987.

[3] R. E. Bryant. Graph Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[4] R. E. Bryant. On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. *IEEE Transcations on Computers*, 40-2:205–213, February 1991.

[5] R. E. Bryant, D. Beatty, K. Brace, K. Cho, and T. Sheffler. COSMOS: A Compiled Simulator for MOS Circuits. In *Proceedings of the 24th Design Automation Conference*, pages 9–16, 1987.

[6] S. Chakravarty. A Characterization of Binary Decision Diagrams. *IEEE Transactions on Computers*, 42(2):129–137, February 1993.

[7] S. Devadas, K. Keutzer, S. Malik, and A. R. Wang. Certified timing verification and the transition delay of a logic circuit. In *29th ACM/IEEE Design Automation Conference*, pages 549–555, 1992.

[8] M. Fujita, Y. Matsunaga, and T. Kakuda. On Variable Ordering of Binary Decision Diagrams for the Application of Multi-level Logic Synthesis. In *The European Conference on Design Automation*, pages 50–54, 1991.

[9] D. Harel. A linear time algorithm for finding dominators in flowgraphs and related problems. In *Proceedings of the 17th ACM Symposium on Theory of Computing*, pages 185–194, Providence, Rhode Island, May 6–8, 1985.

[10] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.

[11] D. M. Lewis. Hierarchial Compiled Event-Driven Logic Simulation. In *Proceedings of ICCAD*, pages 498–500, 89.

[12] S. Malik, A. R. Wang, R. Brayton, and A. Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. In *Proc. IEEE Int. Conf. Computer-Aided Design*, pages 6–9, November 1988.

[13] P. M. Maurer and Y. S. Lee. Gateways: A Technique for Adding Event-Driven Behavior to Compiled Simulation. *IEEE Transactions on Design of Int. Circuits and Systems*, 13(3):338–352, March 1994.

[14] P. M. Maurer and Z. Wang. Techniques for unit-delay compiled simulation. In *Proceedings of the 27th Design Automation Conference*, pages 480–484. ACM/IEEE, 1990.

[15] Peter M. Maurer. The Inversion Algorithm for Digital Simulation. In *Proc. IEEE Int. Conf. Computer-Aided Design*, pages 258–261, November 1994.

[16] K. Pingali and G. Bilardi. Apt: A data structure for optimal control dependence computation. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, 1995.

[17] S. P. Smith, M. R. Mercer, and B. Brock. Demand Driven Simulation:BACKSIM. In *Proceedings of the 24th Design Automation Conf.*, pages 181–187, 1987.

[18] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit State Enumeration of Finite State Machines using BDD's. In *IEEE International Conference on Computer Aided Design*. ACM/IEEE, November 1990.

[19] Z. Wang and P. M. Maurer. LECSIM: A Levelized Event Driven Compiled Logic Simulator. In *Proceedings of the 27th Design Automation Conference*. ACM/IEEE, 1990.

Appendix

A Are MFFRs the best partition?

We conducted experiments to ensure that MFFRs are the best choice for partitions rather than smaller FFRs.

Definition 3 *In a partition of depth n ($PD = n$), all circuit elements are at distance of n or less from the output of the partition.*

Partitions of varying sizes were created by changing PD. The performance effect is described in Figure 15. With increasing PD the simulation time decreases significantly at first, increases after that, and then decreases again, remaining almost flat afterwards. Initial decrease is obtained due to bigger partitions. After that, even though partitions continue to become bigger, the number of partitions increase. To explain this counter-intuitive phenomenon, let us consider an MFFR which is a perfectly balanced tree of height 6. If $PD = 4$, then each subtree at $PD = 4$ forms a separate partition requiring $2^5 + 1 = 33$ partitions. On the other hand, if $PD = 5$, then all the leaves form separate partitions, forming $2^6 + 1 = 65$ partitions. The increased number of partitions increases simulation time. However, eventually, increasing PD results in the MFFR and the best performance is obtained.

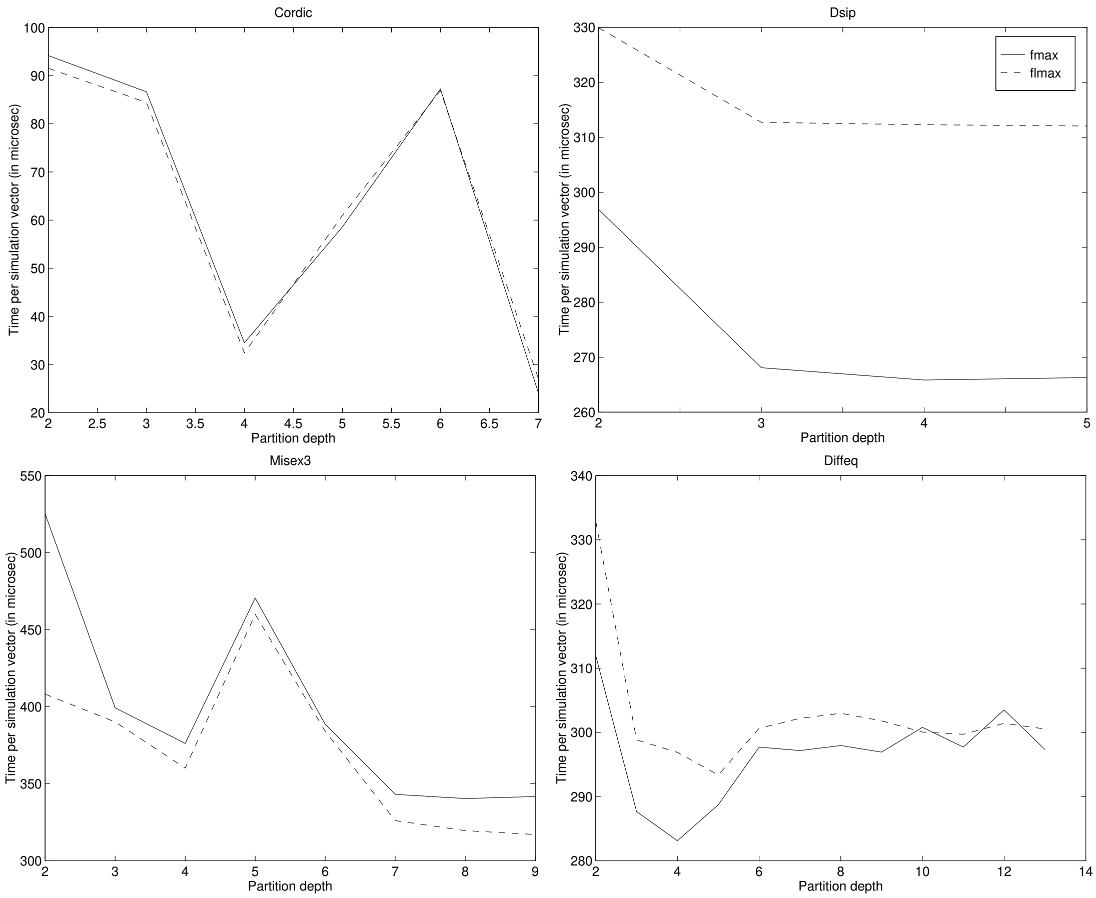


Figure 15: Simulation time *vs.* PD with $CP = 1, TS = 1, SeedValue = 0$