

Parallel Logic Level simulation of VLSI Circuits

Abstract

In this paper, we study parallel logic level simulation of combinational VLSI Boolean networks. The simulator is written in Maisie, a simulation language developed at UCLA. The unique feature of it is that it allows a simulation model to be executed in either conservative or optimistic modes with minimal changes to the model. It also supports a number of optimizations to reduce both checkpointing and blocking overheads for the corresponding simulation protocol. Maisie also supports a variety of parallel and distributed architectures including a network of workstations. We also have implemented our efficient acyclic multi-way network partitioning algorithm named K-MAFM algorithm, which consistently outperforms conventional undirected methods. We have evaluated the impact on parallel circuit simulation of different number of partitions with different parallel and sequential simulation algorithms on different number of processors and obtained very interesting and encouraging experimental results on various Boolean networks from the ISCAS85 benchmark circuits. Significant speed-ups are achieved when either optimistic or conservative algorithm is used. We have evaluated and understood the relationship among several important facts of the parallel language system and performance of simulations.

1. Introduction

Parallel computation can achieve significant reductions in simulation execution time. It is especially useful and important in VLSI design to correctly and efficiently simulate and verify behavior of digital designs.

In recent years, there has been progress in software techniques for parallel and distributed computer simulation [(null)a]. A wide variety of parallel computer architectures exists. In the context of VLSI design and development, those systems may be used for various design and engineering tasks, particularly circuit simulation[(null)a]. Though the job is extremely time consuming and memory intensive. circuits usually have large amount of activities that can potentially be evaluated in parallel. Here, parallel computers and logic simulation seem like a nature fit. Few statictics have been published to exploit the parallelism and analyze performance in circuit simulation[(null)a].

From circuit simulation application perspective, people have reported

- (1) small circuits have enough parallelism. By using asynchronous control and virtual time synchronization with lazy cancellation, limited component sizes, special clock distribution and bounding windows, significant speedup is possible[(null)a].
- (2) the comparison of different parallel processing platforms using an VLSI simulation application to look at the issue of how to design the application to be easily portable[(null)a].
- (3) CMB algorithm [(null)a] exhibits an effective concurrency in logic simulation. But significant overheads avoid it to be a viable alternative to the event-driven algorithm for parallel logic simulation[(null)a].

In this paper, we study parallel gate level simulation for combinational circuits. We present an efficient acyclic multi-way partitioning algorithm to partition large circuit network to several processors under the criteria that each processor satisfies the memory requirement and network cut-size is minimized so that interprocessor communication is potentially reduced. We introduce our logic simulation algorithm which efficiently minimize communication overheads. Our simulator is written in Maisie, a simulation language developed at UCLA, which allows a simulation model to be executed in either sequential, parallel conservative, or parallel optimistic modes with minimal changes to the model[BaLi94].

2. Maisie: Overview

Maisie adopts the process interaction approach to discrete-event simulation. An object (also referred to as a PP for physical process) or set of objects in the physical system is represented by a logical process or LP[(null)]. Interactions among PPs (events) are modeled by message exchanges among the corresponding LPs. We first describe the process representation and communication primitives of Maisie and subsequently indicate how they are used to describe events. A resource manager is used as a running example to illustrate Maisie constructs. The manager manages two types of resources: channels and printers. The initial version of the resource manager only handles printer requests. It is subsequently extended to process channel requests.

A Maisie program is a collection of entity definitions and C functions. An entity definition (or an entity type) describes a class of objects. An entity instance, henceforth referred to simply as an entity, represents a specific object in the physical system. Maisie supports dynamic and recursive entity creation. An entity is created by the execution of a **new** statement which may optionally specify the processor on which the new entity will execute. An entity terminates itself by ‘falling off the end’ of the entity body.

Entities communicate with each other using buffered message passing. Every entity has a unique message buffer; asynchronous send and receive primitives are provided to respectively deposit and remove messages from the buffer.

Maisie uses typed messages. An entity type must define the types of messages that may be received by its instances. A message type consists of a name and a (possibly empty) parameter list.

An entity sends a message by executing an **invoke** statement. Each message is transparently timestamped with the current simulation time and is deposited in the destination buffer at the same (simulation) time at which it is sent.

An entity accepts messages from its buffer by executing a **wait** statement. This statement has the following form:

```
wait  $t_c$  until
{   declarations;
     $r_1$ ;
  or  $r_2$ ;
    ...
  or  $r_n$ ; }
```

where t_c is a numeric value called wait-time and each r_i is a *resume statement*. The most commonly used version of the resume statement references a single message type and has the

following form:

$$[mvar =] \mathbf{mtype}(m_t) [\mathbf{max} v_i] [\mathbf{st} b_i] \\ \text{statement};$$

where m_t is a message type, $mvar$ is a variable of type m_t , b_i is a boolean expression called a *guard*, v_i is a message parameter called a *ranker*, and *statement* is any Maisie statement. The guard is a side-effect free expression that may reference entity variables and message parameters. If omitted, it is assumed to be the boolean constant *true*. The guard is said to be *local*, if it references only entity variables. The message type, guard, and ranker are together referred to as a resume condition. A resume condition with message type m_t and guard b_i is said to be *enabled* if the message buffer contains a message of type m_t whose timestamp is at most equal to the simulation clock, and b_i evaluates to *true* (b_i is evaluated only if the buffer contains a message of type m_t); the corresponding message is referred to as an *enabling* message.

On execution of a wait statement, the message buffer is searched for an enabling message. If the buffer contains more than one enabling message of a given type, the ranker is used to select a unique enabling message: if keyword **max(min)** is used, the enabling message with the largest (smallest) value for parameter v_i is selected. If the ranker is omitted, the enabling message with the smallest timestamp is selected. If two or more resume conditions are *enabled*, the timestamps on the selected enabling message of each type are compared and the message with the earliest timestamp is selected. The selected message is removed from the buffer and delivered to the entity either in the corresponding variable $mvar$ or, if $mvar$ is omitted, in a system-defined variable called **msg**.

If no resume condition is enabled, the entity is suspended for a *maximum* duration equal to its wait-time t_c ; if omitted, t_c is set to a value that is larger than the maximum simulation time for the model. A suspended entity resumes execution *prior* to expiration of t_c , if it receives an *enabling* message; otherwise the entity is sent a special message called a **timeout** message, as discussed in the next subsection. An entity may implement a non-blocking receive by specifying $t_c = 0$.

Figure 1 describes an entity type to model a resource manager. The heading in lines 1 and 2 indicates that the entity type is called *manager* and has one integer parameter called *max_printers*. The wait statement in lines 7--12 contains two resume statements. The first resume statement (line 8) specifies *preq* as the message type and was discussed earlier. The resume condition in the second statement (line 11) does not include a guard. This condition is enabled whenever a *releas* message is available in the buffer. As neither resume condition specifies a message variable, the enabling message is returned in variable **msg**.

```
1  entity manager {max_printers}
2    int max_printers;
3    { int units = maxprinters;
4      message preq {ename hisid; } ;
5      message releas;
6      for (;;)
7        wait until
8          { mtype(preq) st (units>0)
9            { units --;
10             invoke msg.preq.hisid with done; }
11          or mtype(releas)
12            units ++; }
13    }
```

Figure 1: A Resource Manager: Single Resource

A resume condition may also reference message parameters. For instance, assume that the manager in our running example receives requests for one or more printer units and that incoming requests are serviced using the *first-fit* discipline. The following fragment shows how the resume condition is modified to ensure that a *preq* message is accepted only if the requested number of units are available.

```
5  message preq{ ename hisid; int count; } ;
...
7  wait until
8  { mtype(preq) st msg.preq.count <= ~units)
...

```

The wait statement may be used to schedule *definite* and *conditional* events. Consider an entity that models a priority preemptible server. The entity expects two types of requests, *low* and *high*, where the arrival of a *high* message can interrupt the processing of a *low* message. Assume that each message needs 10 units of service. Service of a *high* message is simulated by the following wait statement which schedules a definite timeout message:

```
wait 10 until mtype(timeout)
  invoke jobid with done;
```

Execution of the preceding wait statement suspends the entity for 10 time units because the wait-time is 10 and its single resume statement can be enabled only by a timeout message. A wait statement that schedules a definite timeout message may also be abbreviated by a **hold** statement. The following fragment is equivalent to the preceding wait statement:

```
hold(10);
invoke jobid with done;
```

A wait statement may also be used to schedule conditional events. The following statement uses a *conditional* time-out message to simulate service of a *low* message; *rtime* refers to the

remaining service time of the *low* message that is currently in service. The timeout message is rescheduled if a *high* message is received by the entity in the interim.

```
wait rtime until
{ mtype(high)
  preempt, recompute rtime, and
  serve high priority message;
or mtype(timeout)
  invoke jobid with done;
...

```

3. Parallel Execution

Sequential execution of a Maisie program is straightforward: future messages are stored in increasing order of their timestamps in a global event-list. At every step, the entry with the earliest timestamp is removed from the list and the corresponding message is delivered to the destination entity.¹ For parallel execution of the model, the event-list is physically distributed across the parallel architecture. While each node of the parallel architecture maintains its local simulation clock, messages must still be processed in the global order of their timestamps. This is guaranteed by the underlying distributed simulation algorithm. Maisie has been implemented using a number of parallel simulation algorithms; we briefly describe the two algorithms that were used to report the results described in this paper.

3.1. Null Message Algorithm

At any simulation instant, let n be the next message, with timestamp t_n , to be processed by an LP. In conservative protocols, n will have to wait for some time after its arrival, until the LP can make sure that there won't be any messages with smaller timestamps, before it can be processed. This waiting period, which is the main overhead in conservative protocols, can be reduced by estimating t_n in advance. **Earliest Input Time**(EIT) for an LP, at a given simulation instant, is a lower bound on t_n . Under conservative protocols, therefore, an LP can not *process* any messages with timestamp greater than EIT. Null messages may be used by an entity to locally compute its EIT.

We define **Earliest Output Time**(EOT), for an LP, at a given simulation instant, as the lower bound on the timestamp of the next message sent by the LP. It is equal to EIT plus the value of **lookahead** for the process at that simulation instant. Every LP uses *null* messages to inform the LPs, corresponding to all its *output* channels, of the value of EOT whenever it changes. The EIT of a process is simply equal to the *minimum of the last EOTs received on every input channel*.

¹For simplicity, we assume that all timestamps are unique; if not, alternative criteria must be used as suggested in [(null)a] to select the next message.

Note, therefore, that the knowledge of **communication topology** is crucial for the performance of null message based algorithms. Null message overhead can be reduced by piggybacking null messages with regular messages, and by requiring that the entities send null messages only when they have no regular messages to process. A non zero lookahead is required[(null)a] in every cycle of entities to ensure that the simulation model does not deadlock(i.e. EIT keeps advancing).

3.2. Optimistic Algorithms

In the space-time paradigm, multiple logical processes (LP) may be used to simultaneously compute the state of a physical process at different points in time. We use the terms LP and entity interchangeably. Let T be the upper bound on the time for which the system is to be modeled. Let $P_i^{x,y}$ refer to the LP responsible for the simulation of some physical process in the interval $[t_x, t_y)$, $t_x < t_y$; exactly one LP computes the behavior of a physical process for every t in $[0, T]$. A *precedence* relation, symbolized by \rightarrow , is defined between two LP, where $P_i^{x,y} \rightarrow P_j^{x,y}$ if and only if the state of $P_j^{x,y}$ depends on the state of $P_i^{x,y}$ or on some message received from $P_i^{x,y}$. If $P_i^{x,y} \rightarrow P_j^{x,y}$, we say that $P_i^{x,y}$ is a *predecessor* of $P_j^{x,y}$ and $P_j^{x,y}$ is a *successor* of $P_i^{x,y}$. Note that although the exact predecessor or successor set for an LP cannot be determined a priori, a loose upper bound on these sets can typically be determined (a trivial bound is the entire set of LP in the system).

Given that the preceding set of LP is executed on a distributed architecture, the simulation is executed as follows. The LP that are mapped to a common processor are simulated sequentially. For LP mapped to different architectures, the correct state of each LP is computed by using the following iterative strategy: given some state for its predecessor LP, an LP computes an *estimate* of its final state. During this computation, it generates a (possibly empty) sequence of messages for each of its successors. The message sequence is sent to each successor after a process has computed its final *estimate* state. When a process gets a message sequence from one of its predecessors that is different from the one it received in its previous iteration, the process recomputes its behavior. This procedure is repeated until eventually the computation reaches a fixed-point where further execution of any process does not change its state, and the computation is said to have converged.

We now consider parallel execution of Maisie using this algorithm. To implement this algorithm, the run-time system must perform the following major tasks:

- checkpointing and recomputation of an entity
- creation of multiple ‘incarnations’ of an entity to compute the state of the corresponding physical process at different points in time.

- synchronization of entity reincarnations
- convergence detection to determine the time up to which the simulation has been computed correctly

The preceding tasks may be performed by the run-time system in a variety of ways. For instance, the state of an entity may be checkpointed after every event or may be done less frequently. Convergence detection may be done synchronously or using an asynchronous algorithm. Each of these decisions impacts the execution efficiency of a Maisie program.

The run-time system checkpoints an entity after processing any message other than a timeout message. Recomputation is initiated whenever the timestamp on a message delivered to an entity's message-buffer is smaller than the timestamp on the last message processed by the entity. As explained in [(null)a], this recomputation strategy reduces rollbacks for some applications.

In the current implementation, at most two incarnations exist simultaneously for every entity. Let $P_i^{x,y}$ and $P_i^{y,z}$ be the two incarnations that respectively execute the entity over two successive time-periods, $[t_x, t_y)$ and $[t_y, t_z)$. The choice of a specific duration for each incarnation is a performance issue as discussed in [(null)a]. In a given iteration, both $P_i^{x,y}$ and $P_i^{y,z}$ are executed one after the other; however, the output message sequence generated by the execution of $P_i^{x,y}$ is transmitted before execution of $P_i^{y,z}$ is initiated. The two incarnations are executed repeatedly until $P_i^{x,y}$ converges, at which point, another incarnation of the entity is created. Let LP_a refer to an incarnation of some entity. Convergence detection is merged with message communication among the incarnations as follows:

- S_a^i be the sequence of messages generated by LP_a in its i^{th} iteration.
- R_a^i be the sequence of messages received by LP_a after executing its i^{th} iteration.

On receiving R_a^i from all its predecessors, LP_a executes its $(i+1)^{th}$ iteration. It then sends the suffix of $S_a^{(i+1)}$ that is different from S_a^i together with the time t_c upto which consecutive sequences were identical, to each of its successors. The timestamp t_c is simultaneously broadcast to other processes in the system and is used by every LP to compute the time upto which the entire simulation has converged (also referred to as GVT in the Time-Warp system). Note that each of the preceding tasks is transparent to the programmer.

4. Gate Level Simulation

An event-driven simulator uses a structural model of a circuit to propagate events. Primary input vectors are defined and events on the other signal lines are produced by the evaluations of the activated elements. The simulation processes events such that they will occur in a correct time

order. The future events are maintained in a data structure -- event-list.

The conceptual flow of event-driven simulation is as following: The simulation clock is advanced to the next time for which events are pending in the event-list. this then becomes the current simulation time. The simulator retrieves the scheduled events for the current time from the event-list and updates the values of the active signals. Then it activates the gate and evaluates it. The evaluation may result in new events. These are scheduled to occur in the future according to the delays associated with the operation of the activated gates. The simulator inserts the newly generated events in the event-list. The simulation continues as long as there is logic activity in the circuit or until simulation duration is over.

In our Maisie program, The *driver* entity is in charge of creating simulator entities and initiating primary inputs and coordinating the simulation jobs. Each partition is modeled as a separate Maisie simulator entity. Gates and lines are represented by appropriate data structures within the corresponding partitions. Each entity contains two event-lists, lists of internal and external events to the current partition. We denote them *in_event_list* and *out_event_list*, respectively. At each simulation instance, the entity processes those events with current timestamps in the *in_event_list* by accessing the local data structures. It also collects current events from *out_event_list* and propagates them to their destination entities. Usually the lists are long, if each event is sent as a single message, it will be too expensive in terms of communication overheads for the whole procedure. In attempting to largely reduce communication time, we pack all the outgoing event information that goes to the same entity into one message. So as to the primary inputs. they are predefined input vectors. We group those PI signals whose destination gates are in the same partition and record the line numbers and feed-in values pairwise in the message buffer and send it out as one single message at corresponding simulation clock.

Inside each entity, whenever an event is generated, we put it in either internal queue or external queue. Since we study circuits with minimal transition delay of 1 unit, we can guarantee that at each simulation instance, an activated gate can only produce a future event with clock greater than the current timestamp. Moreover, all the outgoing events are pending in the *out_event_list* because they must have been generated from previous time evaluation and scheduling. When the clock is advanced, we go through the *out_event_list*, pack the signals who go out to the same destination partition and send them out. This is done once for each activated time clock for each entity. we can not only save a lot of message buffering and passing time from this, but also a lot of state saving time when optimistic approach is used.

Entity is activated by receiving messages from other entities. It will detect both of the event lists at the end of each simulation clock, record the timestamp for the nearest future event. At that time, it either receives a message from the others to start the processing, or sends itself a **timeout** message to start the activity.

Overall, the implementation of the Maisie program can be summarized as the following:

```
entity driver { }
{
    establish PI data structures
    create simulator entities and allocate them to the processors
    send all entity IDs to entities
    while ( sim-clock < SIM_LENG ){
        invoke PI-message-pack to the simulator entities
        advance sim-clock
    }
}

entity simulator { }
{
    establish sub-circuit data structures
    initialize simulation data structures
    while ( sim-clock < SIM_LENG ){
        wait nearest-event-scheduled-time until {
            mtype (PI-message-pack) {
                decompose and put events into in-event-list
            }
            or mtype (regular-message-pack) {
                decompose and put events into in-event-list
            }
            or mtype (timeout) {
            }
        }
        invoke regular-message-pack from out-event-list
        process in-event-list until it is empty
        update nearest-event-scheduled-time
    }
}
```

The simulation program is very efficient because of its data structures for event activities such as evaluation, scheduling, and close control on memory size. Details are omitted due to the paper length limit.

5. MFFC Based Acyclic Multi-Way Partitioning

A combinational Boolean network N can be represented as a directed acyclic graph where each node represents a logic gate and a directed edge (i, j) exists if the output of gate i is an input of gate j . A primary input (PI) node has no incoming edge and a primary output (PO) node has no outgoing edge. Each node i in N has an area $a(i)$. A *balanced K-way partitioning solution* $S = (A_1, A_2, \dots, A_K)$ satisfies the following conditions:

- (i) $A_i \cap A_j = \emptyset$ for $i \neq j$ and $\bigcup_{i=1}^K A_i$ contains all the gates in the network;
- (ii) $(1 - \alpha) \cdot \frac{|A|}{K} \leq |A_i| \leq (1 + \alpha) \cdot \frac{|A|}{K}$ for each block A_i , where $|A|$ is the total area of all the gates in N , and α is a user-specified parameter controlling the slack of the area constraint.

Our objective is to minimize the total number of edges $\sum_{1 \leq i, j \leq K, i \neq j} c(A_i, A_j)$ between all the blocks, where $c(A_i, A_j)$ is the number of edges in N from a gate in block A_i to a gate in block A_j . Moreover, for a multi-way partitioning solution S , we can define a directed graph $D(S)$, called the *dependency graph* of S , such that each node in $D(S)$ represents a block in S , and there is a directed edge (A_i, A_j) in $D(S)$ if and only if there exists an edge (x, y) in N such that $x \in A_i$ and $y \in A_j$. We call S is an *acyclic K-way partitioning solution* if the dependency graph $D(S)$ is a directed acyclic graph. Since each signal line across the boundary represents a communication channel, cyclic dependency among the subnetworks on different processors can cause unnecessary roll-backs when optimistic simulation strategies are used or zero lookahead when Null message algorithm is used in our simulators.

Based on FM_algorithm [FiMa82], We have implemented efficient multi-way partitioning algorithms: generalized undirected K-FM, acyclic K-AFM and maximum fanout free cone (MFFC) decomposition based acyclic K-MAFM algorithm[CoLB94]. We have also observed that the K-MAFM algorithm not only guarantees an acyclic partitioning solution but also consistently outperforms the undirected K-FM algorithm, despite the acyclic constraint.

K-MAFM algorithm is an MFFC cluster-based iterative partitioning approach. First, We introduce a few definitions related to MFFC decomposition. We use $input(v)$ to denote the set of nodes which are the fanins of node v , and $output(v)$ to denote the set of nodes which are the fanouts of node v . For a node v in the network, a *cone of v* , denoted C_v , is a subgraph of logic gates (excluding PIs) consisting of v and its predecessors such that any path connecting a node in C_v and v lies entirely in C_v . We call v the *root* of C_v . A *fanout-free cone (FFC) at v* , denoted FFC_v , is a cone of v such that for any node $u \neq v$ in FFC_v , $output(u) \subseteq FFC_v$. The *maximum fanout free cone (MFFC) of v* , denoted $MFFC_v$, is an FFC of v such that for any non-PI node w , if $output(w) \subseteq MFFC_v$, then $w \in MFFC_v$.

It is not difficult to show that MFFC is unique for every node, and any FFC of v is contained in $MFFC_v$. Clearly, if a gate u is in $MFFC_v$, its value affects only gate v and the descendants for gate v , but not other PO nodes in the network. Therefore, it is very natural to cluster u and v together. In general, we can treat each MFFC as a cluster. The results in [CoDi93] showed that MFFCs have the following important properties.

Lemma 1 If $w \in MFFC_v$, then $MFFC_w \subseteq MFFC_v$.

Lemma 2 Two MFFCs are either disjoint or one must contain another.

Based on these results, we can decompose a given combinational network N into a set of MFFCs as follows: Let set Q , initialized to empty, store the list of MFFCs in the final decomposition. We shall repeat the following three steps to obtain the MFFC decomposition of network N : (i) Choose an arbitrary PO v in N ; (ii) Compute $MFFC_v$ and include it in Q ; (iii) Set $N = N - MFFC_v$ and update the PO list of N to include the gates with outputs to some gates in $MFFC_v$. We stop this process when N is empty. Figure 2 shows the MFFC decomposition of a network.

Our study shows another important property of $MFFC$ decomposition, which reveals the following important relationship between the minimum-cut partitioning and MFFC decomposition.

Theorem If we do not consider the area constraint, there is an optimal acyclic two-way partition (X, Y) of N , such that for each $MFFC_i$ in the MFFC decomposition of N , either $X \cap MFFC_i = \phi$ or $MFFC_i \subseteq X$.

Proof Omitted due to the length restriction.

This theorem tells us intuitively that there exists an optimal acyclic two-way partition that does not cut through any MFFC in the MFFC decomposition of N . This result further suggests that MFFC decomposition leads to a natural clustering solution.

Afer MFFC decomposition of the given network, each MFFC is treated as a cluster and is collapsed into a signal node. (If an MFFC is too large, further decomposition is necessary.) Then, we apply the K-AFM algorithm, a simple acyclic generalization of K-FM algorithm, to compute an acyclic K-way partition of the clustered network.

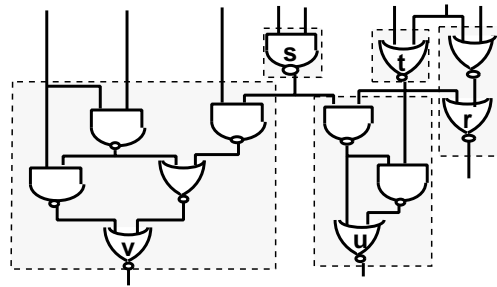


Figure 2 MFFC decomposition of a network

The algorithm starts with a balanced initial partition generated by a random topological sorting solution. A topological sorting solution of a given combinational network N is a linear ordering L of all nodes in N , such that node i appears before node j in L if the output of i is an input of j . Given a topological sorting solution L , we can partition the list L from left to right into K sub-lists such that the total node area in each sub-list does not violate the balance constraint. If we let the nodes in each sub-list to form a block in the partitioning solution, it is easy to show that the resulting partitioning solution is always acyclic.

To maintain the acyclic constraint, we shall check whether the dependency graph $D(S)$ has any directed cycle before moving a node from one block A_i to another block A_j , where S corresponds to the resulting partitioning solution after the proposed move. An efficient algorithm for checking if $D(S)$ has a directed cycle is to, again, use our topological ordering algorithm. There is a cycle if and only if it can not finish a topological sorting. Clearly the algorithm has time complexity $O(K^2)$ in the worst case. Since the value of K (number of partitions) is usually very small, the algorithm is very efficient.

The overall structure of the K-MAFM algorithm is as following: Select a *free* node feasible for move with maximum gain, move it to the destination partition and *lock* the node, and update the gain information of the neighbouring nodes. A move is *feasible* if it involves an unlocked node and if the move does not violate either the balance or the acyclic constraint. A bucket-array data-structure is used for selecting a feasible move with the maximum gain. If moving a node would increase the cut size, the gain of that node is negative. Moving node with negative gains is allowed by the algorithm in order to avoid stopping at local minima in the solution space. When all nodes are either locked or infeasible for moving, the current pass is completed and the best partition encountered during the pass is saved as the initial partition for the next pass. When a pass makes no improvement to the partitioning solution, the algorithm stops. An important contribution of FM based algorithm is that it can complete each pass in linear time with respect to the number of nodes and edges in the network, based on efficient gain computation and updating procedure and the use of bucket data-structure for selecting a node with the maximum gain.

We normally run the K-MAFM algorithm multiple times with different initial partitioning solutions and select the best partitioning solution from multiple runs as the final solution. As a result, it produces very good solution, compared to both K-FM and K-AFM algorithms.

6. Experimental Results

The benchmark examples are from the ISCAS85 suite, which is a set of large combinational circuits originally used to evaluate test generation algorithms. We choose this benchmark suite because it consists of combinational circuits with the signal direction information which is very

useful for the acyclic partitioning. Table 1 shows the characteristics of the benchmark circuits in terms of the number of gates, number of PIs, number of nets, and number of edges.

6.1. Sequential Simulation Experiments

We have run the simulator on Sun SPARC10 workstation using sequential algorithm. We have compared the execution time for different number of partitions for each circuit. In our experiments, the numbers of partitions were chosen to be 2, 4, 8 and 16. Simulation length is 250 time unit. We also include the simulation results on the original circuit for the comparison.

As we can see from the table 2, execution time almost monotonely decreases as the number of partitions increases. The speedup is up to 1.92 for 2 partitions, 3.08 for 4 partitions, 4.98 for 8 partitions, and 8.75 for 16 partitions.

6.2. Parallel Simulation Experiments

We have run the parallel gate level simulations on the IBM SP1 Parallel Computer. It is a distributed memory machine consisting of 24 RS/6000 workstation processors connected by a high speed switch. Each node has a main memory of 128 megabytes.

The simulation code was simulated using a sequential (Global Event List Algorithm[(null)a]), a parallel conservative (Null Message Algorithm), and a parallel optimistic

Circuit	No. of gates	No. of PIs	No. of nets	No. of edges
C2670	1193	233	1426	1983
C3540	1667	50	2167	2911
C5315	2307	178	2485	4331
C6288	2418	32	2450	4800

Table 1. Benchmark circuits.

Circuit	Unpartitioned	K-MAFM			
		k=2	k=4	k=8	k=16
C2670	8.15	5.70	3.65	3.70	3.53
C3540	13.47	9.50	6.35	4.88	7.22
C5315	38.13	24.00	15.23	12.87	9.52
C6288	85.07	44.12	27.65	17.07	9.75

Table 2. Execution Time (secs) (Sequential Algorithm)

(Space-Time Algorithm). Two benchmark circuits, namely, c6822 and c5315, were chosen to compare the execution times of the parallel simulations, executing on multiple nodes of SP1, to the those of the corresponding sequential ones (using the Global Event List algorithm with the highly efficient splay trees used for event list management), executing on one node of SP1. The circuit was partitioned using the K-MAFM algorithm. As noted before, the execution times of the sequential simulation varies with the number of partitions. We chose the number of partitions that minimized the sequential execution time, namely 16, for comparing the sequential and parallel execution times.

Table 3 summarizes the speed-ups obtained using the conservative algorithm(with each of the 16 partitions mapped to a different processor), with respect to the sequential algorithm.

There are many factors mentioned above will affect the performance of optimistic algorithms. In these experiments, we have adjusted two variables: the time-periods of each LP and the frequency of state-saving. We have chosen the best combination and compared the performance with that of conservative algorithm.

In the experiments, we have found that the simulator with short period has better performance. table 4 corresponds to a time period of 5 and 10 units for each LP. If the circuit, c5315, is partitioned and distributed to two or four processors, the performance is about the same. However, if the number of processors is increased, the performance of the simulator with larger time period degrades.

Another variable is the frequency of state saving. If we save state of an entity after each event, there is no unnecessary re-computation when a rollback happens, in other words, the rollback distance is minimum. On the other hand, we will sacrifice the memory and CPU time in saving states even though most of them are unnecessary. If the frequency of state saving is low, the entities will be forced to roll back to an earlier stage and extra recomputation and communication will be introduced. It may cause cyclic rollbacks and degrades the performance seriously. Table 5 shows the performance with different frequency of state saving. lines

Circuit	Time Units	Speed-up
C5315	100	2.79
C5315	250	2.80
C6288	200	3.02
C6288	250	3.33

Table 3. Speed-up obtained using Conservative Algorithm

Time Period	No. of Processors			
	k=2	k=4	k=8	k=16
5	21.85	11.35	8.18	9.90
10	22.36	11.51	11.09	12.18

Table 4. Performance with Different Time-Periods

correspond to state saving after every event, every 5 events, every 10 events, and every 20 events, respectively. In this case, the less frequently we save states, the better performance we get.

Figure 3 compares the execution times of the optimistic and conservative algorithms with the sequential execution times, for c5315. It plots the execution time with respect to the number of partitions into which the circuit is divided. For the parallel simulations (optimistic and conservative) each partition is mapped onto a different processor.

Frequency	No. of Processors			
	k=2	k=4	k=8	k=16
1	26.60	14.04	10.36	13.21
5	24.05	12.81	9.36	12.98
10	21.85	11.35	8.18	9.90
20	21.57	11.14	8.4	10.50

Table 5. Performance with Different Frequency of State Saving

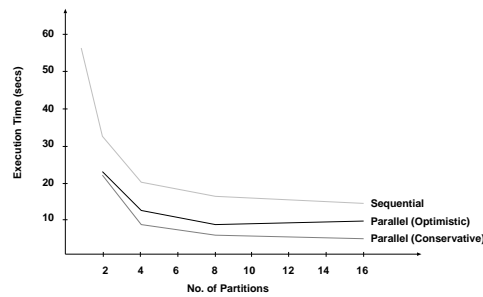


Figure 3 Execution Times for c5315

7. Conclusion and Future Research

8. Acknowledgments

This work is partially supported by the California MICRO program with Cadence, Hewlett-Packard, and Zycad, by ARPA/CSTO under Contract J-FBI-93-112, and by the NSF Young Investigator Awards under ASC-9157610 and MIP-9357582.

References

[(null)]

- [CoDi93] Cong, J. and Y. Ding, "On Area/Depth Trade-off in LUT-Based FPGA Technology Mapping," *Proc. 30th ACM/IEEE Design Automation Conf.*, pp. 213-218, June 1993.
- [CoLB94] Cong, J., Z. Li, and R. Bagrodia, "Acyclic Multi-Way Partitioning of Boolean Networks," *Proc. ACM/IEEE 31st Design Automation Conf.*, June 1994.
- [FiMa82] Fiduccia, C. and R. Mattheyses, "A Linear Time Heuristic for Improving Network Partitions," *ACM/IEEE Design Automation Conf.*, pp. 175-181, 1982.
- [BaLi94] Bagrodia, R. and W.-t. Liao, "A Language for design of Efficient Discret-Event Simulations," *IEEE Transactions on Software Engineering*, March, 1994.