A Multidimensional Study on Parallel Switch-Level Circuit Simulation*

Yu-an Chen and Rajive Bagrodia
Computer Science Department
University of California at Los Angeles
Los Angeles, CA90024

Abstract

Circuit Simulation is a critical bottleneck in VLSI design. This paper describes a parallel switch-level simulator, MIRSIM, for VLSI circuits and presents the results of an experimental study to evaluate the effectiveness of multiple synchronization protocols and partitioning algorithms in reducing the simulation time of VLSI circuits. Specific contributions of this paper include: (i) parallelizing an existing switch-level simulator such that the model can be executed using sequential, conservative, and optimistic simulation protocols with minor enhancements for efficiency; (ii) evaluating effectiveness of manual and automatic partitioning algorithms for parallel circuit simulation; and (iii) comparative performance evaluation of MIRSIM with three conservative and one optimistic simulation protocols for seven circuits, ranging in size from 3K transistors to about 87K transistors.

1 Introduction

VLSI circuit designs may be simulated at many levels of detail, ranging from abstract functional specifications to the more detailed gate, switch and circuit levels. In general, both the accuracy and the execution time of a simulation model increase with the level

*The paper extends its preliminary version published in PADS97 by including additional synchronization algorithms, detailed overhead analysis, and a detailed presentation of the circuit partitioning interface and results.

of detail. Parallel execution offers the tantalizing potential of providing a reasonable turnaround time even for detailed models.

Three primary mechanisms have been used for parallel circuit simulation: synchronous, conservative asynchronous, and optimistic asynchronous. Synchronous techniques attempt to execute events with the same timestamp in parallel. Although some performance benefits have been reported with synchronous and loosely synchronous parallel simulation [20, 24, 25], other studies have shown that these techniques do not appear to scale. Wong et al. [30] reported that in small circuits, ranging from 347 to 3,680 elements, the average activity level was only about 1.2% and the average concurrency level (i.e. average number of simultaneous events) was of between 18 to 30. Bailey et al. [4] also showed that this concurrency level does not scale linearly with circuit size.

In contrast, some analytical studies have shown that asynchronous parallel techniques have a significant potential to exploit the parallelism inherent in large circuits. Briner et al. [6] used the critical path analysis method to evaluate the parallelism in the circuit simulation. They measured potential speedups ranging from 117 to 337 for three circuits with sizes from 3,360 to 11,820 transistors. This was comparable to the average concurrency levels of 6 to 12 that were found in the same circuits using synchronous mechanisms.

Although asynchronous parallel execution appears to offer significant potential for performance improvements, as will be discussed in the next section, researchers have had mixed experiences in realizing this potential. A number of factors govern the performance of parallel circuit simulation [5]: circuit characteristics that determine the inherent parallelism in the design; the timing models and abstraction level used by the simulators; the partitioning methods used to decompose the circuit into multiple partitions; the architectural and operating system characteristics including processor speed, communication latency, context switching costs; and lastly, and perhaps most importantly, the overhead of the synchronization protocol. As the inherent parallelism in a circuit is fixed by its design and timing models, a simulation environment can be developed to explore the benefits of parallel execution along the remaining three degrees of freedom, as shown in

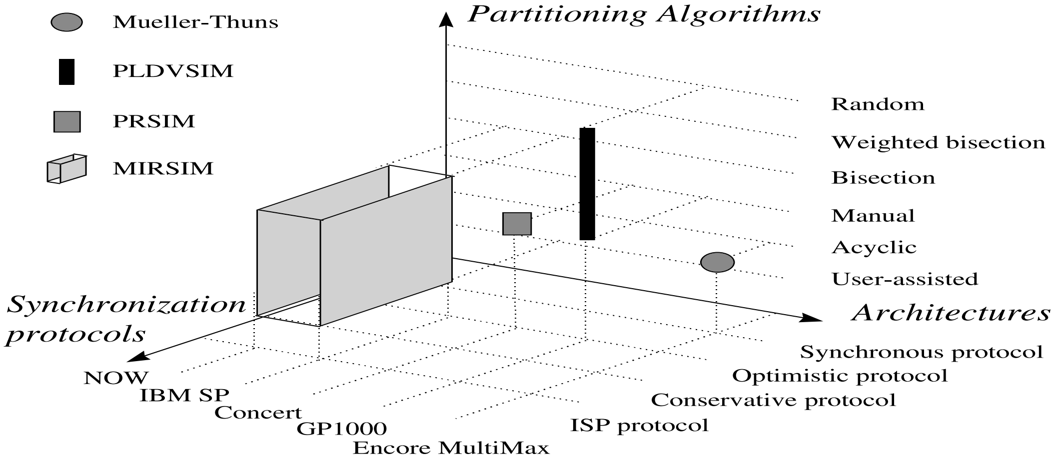

Figure 1: The related work in parallel switch-level circuit simulator.

Figure 1.

A parallel switch-level simulator, called MIRSIM, is described in this paper with precisely this goal. First, we demonstrate that conservative techniques can be useful for parallel switch-level simulation; all previous studies for switch-level simulation used optimistic synchronization protocols. Second, we demonstrate that neither conservative nor optimistic techniques are strictly better; the utility of a specific protocol is determined by the feasibility of generating acyclic partitions. Third, for a given partitioning scheme and parallel architecture, we derive the tight lower bounds on the execution time of the parallel implementation. Fourth, we corroborate previous findings on the superiority of manual partitioning for optimistic circuit simulation [15] and show that conservative techniques can also benefit from manual partitioning. Lastly, we demonstrate the importance of exploring the three-dimensional parameter space for parallel switch-level models by noting that no single partitioning mechanism or synchronization protocol dominates for even this relatively small set of benchmarks.

The next section surveys related work in parallel gate and switch-level simulation. Section 3 is a brief description of the sequential IRSIM simulator [29] and the modifications that were made to derive the parallel MIRSIM implementation. Section 4 describes the partitioning environment and the specific partitioning algorithms that have been investigated in this paper. Section 5 describes the results from five simulation protocols for a set of benchmark circuits tested on the IBM SP, a distributed memory multicomputer. Section 6 summarizes our contributions and outlines the ongoing work in porting the simulator to a shared memory environment.

2 Related work

A number of studies have explored the viability of parallel gate-level simulation using both conservative and optimistic techniques [21, 26, 27, 28]. For instance, Su & Seitz [28] studied the effectiveness of six variants of the conservative null message deadlock-avoidance mechanism on a iPSC/2 Hypercube machine, and reported that the null message overhead is an obstacle to good performance. Soule and Gupta [26] applied a conservative technique based on deadlock detection and resolution in their the gate-level simulator on a 16-processor Encore Multimax. Their measurements showed that 50% to 79% of execution time was spent in deadlock resolution - parallel execution did not offer any improvements in performance.

Sporrer et. al. [27] implemented a gate-level simulator using the Time Warp synchronization technique and tested it on a network of workstations with six ISCAS89 benchmarks. They used a hierarchical, cluster-based partitioning algorithm and reported a speedup of about 8 using 20 workstations. Wisley and McBrayer [22] also used optimistic techniques for parallel simulation of VHDL models and investigated combination of processes to increase the computation granularity.

Relatively few research efforts have compared performance of conservative and optimistic algorithms on a common set of circuits. Chung and Chung[10, 11] implemented four kinds of synchronization algorithms for gate-level simulation on the Connection Machine CM-2, a massively parallel SIMD architecture. These four algorithms utilized are synchronous simulation technique, conservative simulation using a global clock, conservative with lookahead technique, and Time warp technique. Two circuits, a 32-bit array multiplier with about 8000 gates and an ISCAS/C6288 benchmark with 2400 gates were used as benchmarks. They found the Time Warp technique performed worst and the conservative technique with global clock was the fastest.

Figure 1 gives a visual description of how existing switch-level simulators span in the three dimensional problem space. Arnold [2] implemented a parallel RSIM [29] simulator called PRSIM using an optimistic asynchronous technique. PRSIM was tested on a multiprocessor machine, *Concert*, with a single hand-partitioned 64-bit adder. A speedup of 3.77 was reported using 6 processors compared with one-node execution of PRSIM. Mueller-Thuns et al. [24] partitioned a "flat" transistor netlists into an acyclic graph of strongly connected components (SCCs) and simulated it using their loosely synchronous algorithm. They obtained speedups ranging from 2.21 to 7.56 with five benchmarks on 8 processors of an Encore MultiMax machine. However, some of the SCCs might have contained a large number of transistors which unbalanced the computation. The SCCs had to be split over processors which increased synchronization overhead.

Briner [15] described a parallel implementation of LDVSIM, called PLDVSIM, using Time Warp with incremental state saving technique. PLDVSIM supports both high level C models and RSIM switch-level models. Prior to partitioning, a circuit was first clustered into a set of "stages," where each stage, also called *dc-connected component*, was defined as a maximal set of transistors connected through their sources and drains. Two synchronization schemes were implemented for PLDVSIM:

1. interprocessor synchronization where all stages on the same processor were contained in a single simulation process, and
2. interstage synchronization where each stage was treated as a simulation process with its own simulation clock and event queue.

PLDVSIM was implemented on a GP1000 shared memory machine with five different benchmarks. The researchers compared the performance of random partitioning, bisection partitioning, and bisection weighted by activity partitioning. They found that the random partitioning performed better than both forms of bisection when using interstage synchronization. However, manual partitioning tended to be almost three times better than bisection partitioning. In general, the average speedups measured by Briner were approximately 4 to 6 on 16 processors and 7 to 10 on 32 processors. However, a symmetric CC array circuit, an array of classification combiners with 31,680 gates, reached speedup of 22 when hand partitioned and simulated on 32 processors.

3 Switch-level simulators

IRSIM, an extended version of RSIM, is an event-driven, switch-level simulator and incorporates a simple linear model of MOS transistors. In IRSIM, MOS transistors are modeled as voltage controlled switches in series with fixed resistances, while gates and interconnects are modeled as fixed capacitances. Standard RC network techniques are used to compute the final logic state of each node, as well as the transition delay. Further improvement on the timing models was provided by C. Y. Chu [9].

IRSIM uses three kinds of major data structures: nodes, transistors and events. A node represents the connection point between transistors and stores the capacitance and associated potential during simulation. Each event schedules a change in the state of a node at current time. The transistor structure contains pointers to gate, source, and drain nodes. Additional fields are defined to store the resistance and state of the transistors. The transistor structure is declared as follows:

```
struct Transistor {
  Node    *gate, *source,
          *drain;      /* nodes to which trans is connected */
  char    state;       /* state of this transistor          */
  char    ttype;       /* type of transistor                */
  Resists *r;          /* transistor resistances            */
```

```
  char    tflags;
} Transistor;
```

In IRSIM, events are generated by new signals from input vectors or due to a change in the potential of a node. In the beginning, a new event with the earliest time step is dequeued from the global event queue and delivered to the associated node. The state of transistors which connect to the affected node is changed too. The new potential of source and drain of those transistors and the delay time is computed by standard RC network technique. If the new values are different from the previous ones, new events are generated and inserted to the global event queue. The simulator repeatedly executes events from the global event list until there are no more events or the event time of the earliest event exceeds the maximum simulation time.

3.1 Parallel implementation

A parallel implementation of IRSIM, called MIRSIM [1], was designed by implementing the IRSIM simulator in Maisie [3], a parallel simulation language. The basic simulation object in a Maisie model is called an entity. Entities communicate with each other via time stamped messages which can be used to simulate the events in a DES model. Maisie supports the execution of a discrete-event simulation model with multiple synchronization algorithms which include a sequential global event-list algorithm running on a uni-processor machine, as well as a conservative and optimistic parallel algorithm. The conservative algorithm has three different variants including the null message protocol [23], the conditional event protocol [7], and an asynchronous combination of the null message and the conditional event protocols [17]. The optimistic algorithm uses the space-time protocol [8] with lazy cancelation. Besides these protocols, Maisie also provides an Ideal Simulation Protocol(ISP) [16] runtime which provides a reference point to estimate the overhead of different synchronization protocols. ISP computes the execution time for a given model on a given architecture by executing the model using a complete message trace for that model. Using the message trace, ISP pre-computes the sequence in which incoming messages are accepted by the entity, thus allowing it to process the "next" message in the sequence as soon as it arrives. In this manner, ISP excludes overheads that are due solely to the synchronization protocol but includes all overheads that are due to the model partitioning, message transmission and buffering, and other factors related to the execution of a parallel program. This allows an analyst to determine if the observed inefficiencies in the execution of a parallel simulation model are primarily due to implementation inefficiencies in the simulation protocol or to the inherent lack of concurrency in the parallelized model.

IRSIM had to be modified significantly to incorporate it within Maisie. We first describe the modifications necessary to support parallel execution and subsequently address those modifications needed to utilize Maisie's multiple synchronization protocols. As described in section 5, the modifications did not degrade the sequential performance of MIRSIM as compared to IRSIM.

For parallel execution, the circuit to be simulated is first partitioned into a number of subcircuits (as discussed in section 4), each of which may potentially be simulated on a separate processor. A Maisie entity, called $irsim$, was defined to capture the functionality of IRSIM. A new instance of this entity is created to simulate each partition. A message is sent from one entity to another if the state of a node in the subcircuit changes and this information is required by another partition. The runtime systems under these entities are responsible for delivering the messages and scheduling the execution of entities. In the conservative runtime, all messages are delivered in time order and an entity is activated only when the runtime system can make sure there will be no straggler messages later. In the optimistic runtime, an entity keeps executing as long as it has a message in its buffer. The entity's state is checkpointed by the runtime periodically to recover from any causality errors.

IRSIM was originally designed for a sequential implementation and relies heavily on global data structures. As Maisie entities communicate only via messages (to facilitate their distribution among multiple processors) it was first necessary to replace the global data structures by local data structures declared within an entity. Likewise, all read and write references to the global data were implemented via message communication between entities.

3.2 Modifications to support optimistic protocol

Optimistic execution of a model requires periodic checkpointing of the state of each entity to correct for potentially erroneous computations. Researchers have pointed out the checkpointing overhead is a key issue for optimistic protocols. The state of a Maisie entity is defined by a collection of data structures which are either declared locally, or dynamically allocated by the entity. The state of an *irsim* entity contains: transistors and nodes which construct the electronic net of a sub-circuit, events which describe the activity in the circuit, and the current simulation clock. Since these data structures are allocated dynamically, checkpointing becomes a time-consuming job because it needs to trace down pointers to traverse the entire network. To facilitate checkpointing of dynamic memory functions *mayc_malloc()* and *mayc_free()* were added to Maisie. These functions are similar the standard C memory allocation functions, but support efficient checkpointing by allowing each entity to manage its own dynamic memory pool from a list of contiguous memory blocks.

As circuits with many thousands of transistors are not unusual, checkpointing the entire circuit may still be prohibitive due to memory constraints. To reduce this cost, the state of an *irsim* entity is subdivided into two parts: *static-valued* data structures whose values are initialized once and subsequently remain unchanged during the simulation, and *dynamic-valued* data structures whose values change as necessary. For example, the transistor data structure has been split into two parts: $Transistor$ and $DynTrans$.

```
struct Transistor {
  Node     *gate, *source, *drain;
  char     ttype;
  Resists  *r;
```

```
  DynTrans *dyn_tran;
};
struct DynTrans {
  char        state;
  long        tflags;
};
```

The *Transistor* structure remains static after the circuit network is constructed. Therefore, it is allocated through standard C malloc() function and will not be checkpointed; only the *DynTrans* structure needs to be checkpointed by the runtime. Similar splitting is applied to other data structures also. As a result of these optimizations, the checkpointing overheads were reduced dramatically. As shown in Table 3, the checkpointing overheads consumed only 5%, or less, of the total simulation time.

3.3 Modifications to support conservative protocol

In general, the performance of a conservative algorithm is heavily influenced by the lookahead properties and communication topology of the model. Maisie provides a set of modular interfaces that may be used to specify the lookahead and connectivity for each entity. As IRSIM uses a static 0.1 ns timing model, this must be used as the lookahead for each *irsim* entity. The static communication topology is extracted from the partitioning interface and is specified in the initialization phase of the conservative execution using the Maisie facilities provided for this purpose. No other modifications are necessary to support conservative synchronization.

4 Circuit partitioning

As discussed before, to support parallel execution, the circuit must be divided into a number of partitions, such that each partition can potentially be simulated on a separate processor. We have implemented three kinds of partitioning methodologies: manual/structure partitioning, automatic partitioning, and user-assisted automatic partitioning which is a combination of the first two methods. MIRSIM provides an interactive partitioning interface called Interactive Circuit Partitioning and Simulation Environment (ICPSE) [19] that allows users to generate a partition and to evaluate the quality of the partitioning by providing information on the number of connections being cut, number of transistors in each partition, and the communication topology among partitions.

A number of terms used in this section are defined as follows: Given a circuit containing N connected components, a partitioning algorithm is used to decompose the circuit into K disjoint but complete partitions. The circuit is modeled as a graph $G(C, E)$, where C is a set of components and E is a set of nets. A *component* may have multiple input and output ports. A *net* connects one output port of a component to the input ports of one or multiple components. A net is *cut* if the components connected by the net are put in different partitions. The set of nets which have been cut is called the *cut-set* of this partitions. The objective function used by most algorithms is to minimize the size of the cut-set, while ensuring that each partition has an approximately equal area. The *area* may be represented either by the number of components, number of nets, or a combination of the two metrics.

Rather than partitioning the original transistor-level netlists directly, MIRSIM first clusters together transistors on the same dc-connected component. Thus the charge-sharing events in the IRSIM model will not span across partition boundaries. Each set of dc-connected transistors is treated as a basic component in the partitioning algorithm. A net connects two components if there are connections from the sources or drains of transistors in one component to the gates of the transistors in the corresponding component. Use of the preceding clustering technique generally results in clusters that contain a widely differing number of transistors. To facilitate the generation of balanced partitions, the clustering phase maintains two metrics with each cluster: the number of transistors and the number of output signals: The former represents the area of the cluster and is used in the generation of balanced partitions, and the latter is used to estimate the size of cut-set of this partitions.

4.1 Automatic partitioning

Two kinds of automatic partitioning algorithms have been implemented in our ICPSE environment to partition the clustered circuit. The first algorithm, referred to as K-FM, is the well-known K-way generalization of the 2-way iterative partitioning algorithm proposed by Fidducias-Mattheyeses [14]; the second algorithm, called K-AFM [12], is an extension of K-FM which generates acyclic partitions.

The K-FM algorithm starts with a randomly generated, balanced, initial partition and subsequently moves components (or clusters) among the partitions. At each step, feasible components are first identified. A component is feasible for moving, if moving it to another partition will not violate the balance constraint. Among all feasible components in a partition, the component with the highest gain is selected, where the *gain* of a component is defined to be the amount of reduction in the size of the cut-set that results from moving the component to the other partition. When all feasible moves have been inspected, the best partition encountered during the current iteration is saved as the initial partition for the next pass. The algorithm terminates either after a specified number of iterations have been executed or when an iteration makes no improvement to the partitioning solution.

The K-AFM algorithm is derived from the K-FM algorithm by imposing an additional acyclicity constraint on partitions. In order to define an acyclic partition, the direction of a net must be defined first. A *directed net* exists from a component C_i to another component C_j if there is a connection between the source or drain of a transistor in the component C_i to the gates of transistors in the cluster C_j. A *directed edge* exists from partition P_i to partition P_j if component C_i is in P_i and component C_j is in P_j and there is a directed net from C_i to C_j. For an acyclic partitioning, it must be the case that given any two partitions, say P_i and P_j, there is no such path from P_j to P_i, if a path exists in the partitioning from P_i to P_j.

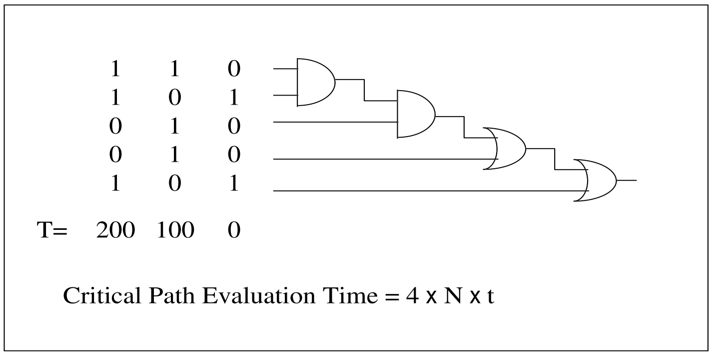

Figure 2: Uniprocessor Circuit Simulation

The K-AFM algorithm first sorts the component in topological order. If some components form a cycle, these components are collapsed into a super component before the topological sorting. An acyclic initial partition can be obtained from the order information. The criteria for choosing a feasible move in the K-AFM algorithm are similar to the K-FM with the additional constraint that the acyclicity condition is not violated. For a combinational circuit, it is always possible to generate acyclic partitions.

As this algorithm imposes an additional constraint on the partitioning process, it typically yields partitions with a larger cut-set than the standard K-FM algorithm and therefore generally causes more messages to be exchanged between partitions. However, as seen from the experimental results in section 5, the performance of circuit simulation with K-AFM partitions is much better than that of K-FM even though K-AFM partitions require more communication overhead. The reason for this can be explained as follows:

Figure 2 shows a circuit with four gates connected serially. Assume that N input vectors are fed into the circuit at 100 ns (simulation time) intervals and the simulator takes t units of physical time to evaluate each gate, then it takes $4 \times N \times t$ for a uniprocessor simulator to evaluate events on the critical path.

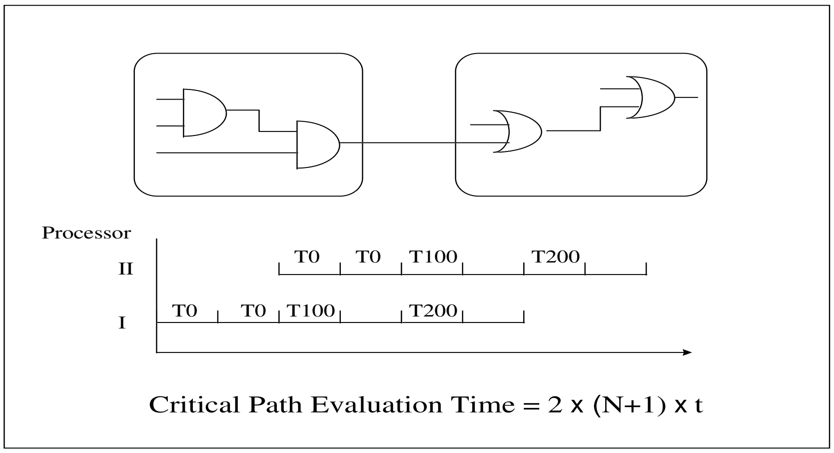

Figure 3: Acyclic Partition.

If we partition the circuit in a acyclic way as shown in Figure 3 and simulate each partition on different processors, the input vectors can be pipelined into the first processor, yielding a total execution time of $2 \times (N + 1) \times t$. A speedup 2 will be achieved if N is sufficiently large.

However, cyclic partitions like the one in Figure 4 do not support this form of pipelining. The first processor can compute the output values for a vector only after it receives the results from the second processor. From the time chart, we can easily determine total execution time of $4 \times N \times t$, indicating that the parallel execution has been sequentialized.

If a cycle among partitions does not exist in the original circuit and is introduced by a partitioning strategy, it is called a *partition-induced cycle*. As shown above, the partition-induced cycle tends to sequentialize the simulator and reduce the parallelism among partitions. Cycles also increase the synchronization overhead for conservative protocols as null messages traverse the cycle to advance the simulation time. In optimistic protocols, the first partition can process the second input vector immediately after the first input vector. However, straggler messages from the second partition will cause roll back. Experiments to corroborate this observation are presented in section 5.

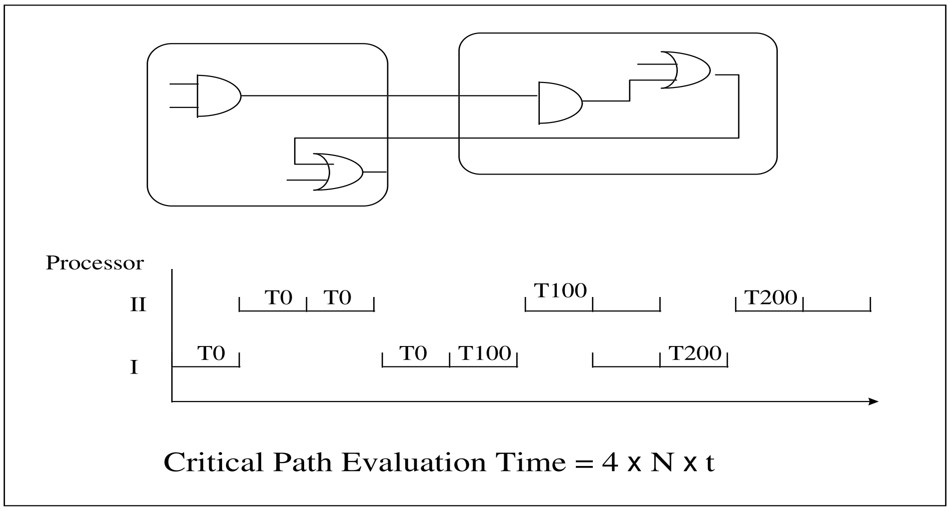

Figure 4: Cyclic Partition.

4.2 Manual partitioning

If the original circuit contains cycles, it may be possible to collapse those into a super component; however, it may not be possible to generate balanced, acyclic partitioning. Fortunately, with the help of a circuit designer, the circuit's structure information can be used to produce a balanced partition with low communication overhead. To facilitate the

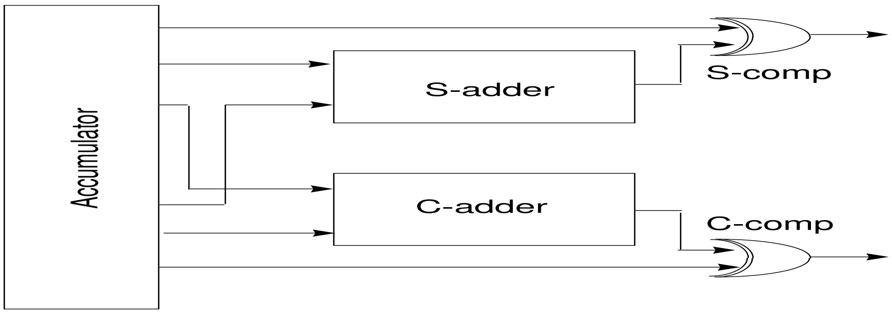

Figure 5: The scheme of DDFS circuit.

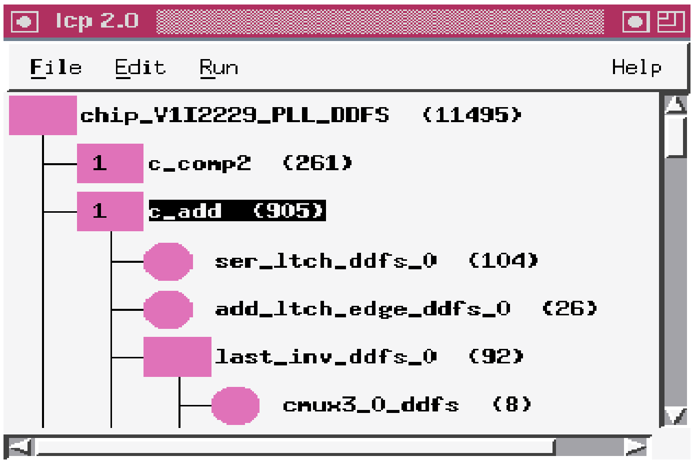

Figure 6: A graph interface environment to partition circuit.

manual partitioning, MIRSIM provides an interactive GUI, which allows users to manually partition circuits designed with the Magic tool. The hierarchic structure information exported by Magic is first read in and displayed as a tree structure. The users can travel up and down among different levels of the hierarchy and assign the partition numbers at any level. The assignment is subsequently used to generate netlists for each partition automatically. For example, Figure 5 shows the top most level of functional blocks of a Direct Digital Frequency Synthesizer (DDFS) [18] circuit. The circuit description is read by ICPSE and the hierarchical information is displayed as shown in Figure 6, where circles represent leaf nodes and rectangles represent higher level functional blocks.

Two heuristics were used to partition the circuits simulated by MIRSIM:

- Partition on the boundary of clocked latches. Mueller-Thuns et al. [13, 25] have suggested that sequential circuits can be partitioned on the boundaries of clocked latches. Therefore, messages are released at the clock rate which will reduce the communication overhead.

- Avoid partition-induced cycles. For example, the gate "s_comp" can be partitioned either with "accumulator" or "s_adder" without affecting the size of cut-set. However, if s_comp is in the same partition with accumulator, a communication link from the partition of s_adder to the partition of accumulator will be created which

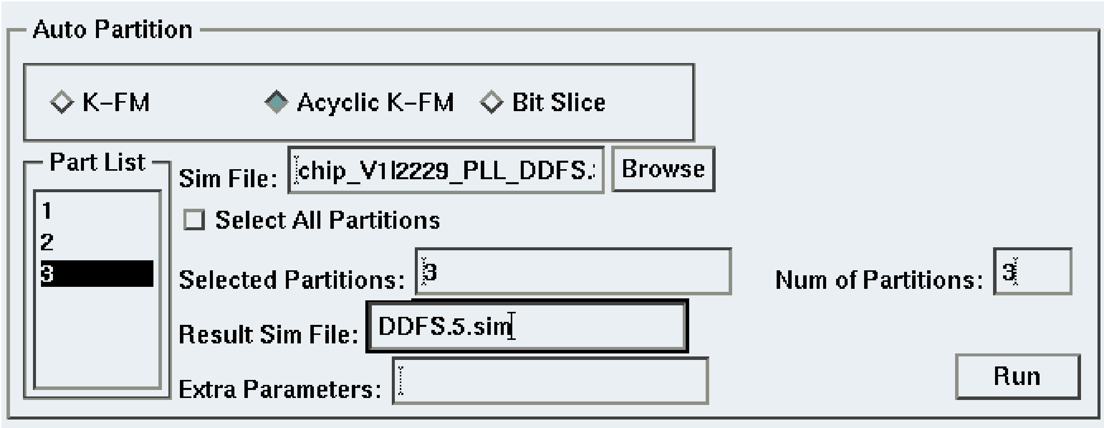

Figure 7: User-assisted partitioning.

doesn't exist in the original design. As discussed in the previous section, such partition-induced cycles will reduce the parallelism as well as increase the synchronization overhead. So it follows that s_comp should be in the same partition of s_adder.

The performance of the simulation for manual partitioning that employed these heuristics is reported in the next section; in general they yielded much better performance than the K-FM algorithm.

4.3 User-assisted partitioning

Although manual partitioning has its advantages, it is very hard for a human to keep abreast of a large number of partitions. Moreover, the design structure also limits the total number of partitions. Sometimes this results in an unbalanced partition because the sizes of modules are too dissimilar. For example, if we want to obtain 5 partitions from the DDFS circuit, the accumulator block could be a good candidate for further decomposition. Unfortunately, this sub-circuit is "flat" and structure partitioning is almost impossible. To solve these problems, we introduce a user-assisted automatic partitioning method. The basic concept is to allow users to manually specify an initial coarse partition. Selected partitions may be further decomposed into a specified number of partitions using a specific automatic partitioning algorithms. For example, as shown in Figure 7,

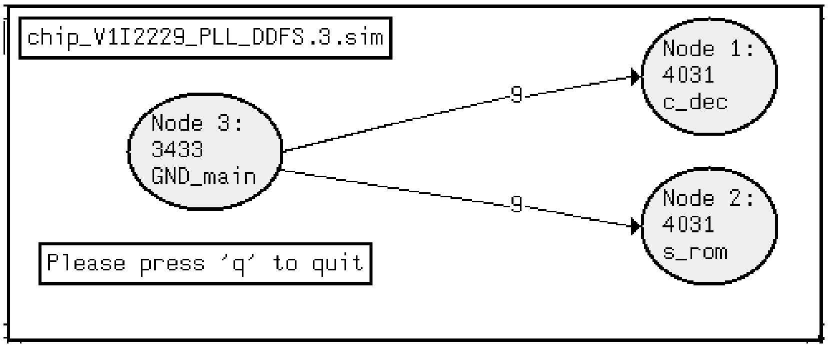

Figure 8: The partitioning result.

partition 3 (which represents the accumulator in the manual partition) is selected for decomposition into three partitions using K-AFM algorithm. By combining the automatic partitioning algorithms and manual partitioning, a circuit may be divided into a larger number of balanced partitions without loss of structure knowledge.

4.4 Partitioning quality inspection

After a circuit has been partitioned, ICPSE displays the result graphically, as shown in Figure 8. In the graph, each node represents a partitioned subcircuit where each subcircuit records its own total of number of transistors. The directed edges between nodes represent the communication direction, and are weighted by the number of cut nets. Thus, users can see the communication topology before simulation and repartition the circuit, if necessary. For instance, two partitions with a "communication-heavy" link can be collapsed, partitions that form a small cycle in the graph may be merged into a larger partition, or users can check if a partition-induced communication cycle has been created.

5 Experiments and results

Seven circuits were used to measure the performance of MIRSIM. The largest one, *Cdma*, is a chip which provides fine and coarse timing recovery for a CDMA radio. It consists of approximately 87K transistors. The smallest circuit, called *Chip2u*, is an